Universität Leipzig
Wirtschaftswissenschaftliche Fakultät
Institut für Wirtschaftsinformatik
Professur Softwareentwicklung
Betreuender Hochschullehrer: Prof. Dr. Ulrich Eisenecker
Betreuender Assistent: Dr. Richard Müller

Thema

Visualisierung von Klassenbestandteilen am Beispiel der Stadtmetapher

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science – Wirtschaftsinformatik

vorgelegt von: Schulze, Christian

Matrikelnummer: 2890096

Email-Adresse: christian-schulze@gmx.net

Telefonnummer: 0341 58069772

Anschrift: Lessingstraße 8

04109 Leipzig

Leipzig, den 03.01.2017

Abstract

Die Stadtmetapher des Softwarevisualisierungsgenerators der Forschungsgruppe *Softwarevisualisierung in 3D und virtueller Realität* soll um zwei Varianten der Darstellung von Klassenattributen und -methoden erweitert werden. Nach Evaluation bereits existierender Stadtmetaphern mit dem Fokus auf die Anwendbarkeit verschiedener visualisierter Aspekte auf die Stadtmetapher des Softwarevisualisierungsgenerators folgt eine Definition der zu implementierenden Varianten zur Darstellung der Klassenbestandteile. Als Basis einer der umzusetzenden Varianten dient dabei der Ansatz von *CodeCity,* der die Zerlegung des Gebäudes in sogenannte *Bricks* vorsieht. Eine zweite Variante der Visualisierung soll das Gebäude ebenfalls in Segmente gliedern, jedoch konsequent aufeinander stapeln. Zusätzlich werden Zugriffsmodifikatoren beziehungsweise Methodentypen farblich verschieden gekennzeichnet sowie eine Sortierung der Elemente nach mehreren Kriterien vorgenommen.

Schlüsselwörter

generative Softwareentwicklung, modellgetriebene Softwareentwicklung, Softwarevisualisierung, Stadtmetapher, Xtext, Xtend, Java, Extensible 3D, Softwaremetriken

Gliederung

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

LOC	Lines of Code
NOA	Number of Attributes
NOM	Number of Methods
NOS	Number of Statements
X3D	Extensible 3D

1 Einleitung

1.1 Motivation und Problemstellung

Software gewann über die letzten Jahrzehnte mehr und mehr an Umfang und Komplexität, wodurch es immer schwerer wird, ihre Funktionsweise nachzuvollziehen und Fehler zu identifizieren, um den Quelltext zu optimieren oder weiterentwickeln zu können (vgl. [Caserta & Zendra 2011]). Diese Probleme treten insbesondere bei der Weiterentwicklung von bestehenden Softwaresystemen auf und sind nicht mehr ausschließlich durch die Historie des Systems, das heißt, die Änderungen der Softwareartefakte unter dem zeitlichen Aspekt, und dessen Dokumentation zu kompensieren. Dadurch entsteht die Notwendigkeit, neue Techniken zu entwickeln, um Software besser zu verstehen und analysieren zu können (vgl. [Wettel & Lanza 2008]). Die Softwarevisualisierung liefert neue Sichten auf Software und deren Bestandteile, um Probleme hinsichtlich der Struktur und des Verhaltens besser erkennen zu können. Neben der Visualisierung der Struktur und des Verhaltens bietet sie zusätzlich die Visualisierung der Historie von Softwareartefakten (vgl. [Diehl 2007, 3]).

Ein Projekt, das sich mit dieser Thematik beschäftigt, ist der Softwarevisualisierungsgenerator (folgend Generator genannt) [Müller u. a. 2011], welcher am Institut für Wirtschaftsinformatik der Universität Leipzig entwickelt wurde. Dieser Generator erzeugt aus Softwareartefakten automatisiert 2D-, 2,5D- und 3D-Softwarevisualisierungen und greift dabei auf verschiedene Metaphern zurück. Aufbauend auf der Arbeit von [Zilch 2015] soll nun die Stadtmetapher um eine Darstellung von Methoden und Attributen erweitert werden. Dazu wird dem bestehenden Modell ein neues Visualisierungselement hinzugefügt und als Teilelement des Quader-Klassenelementes beschrieben. Dieses Teilelement repräsentiert später im Einzelnen die Methoden und Attribute der Klasse, indem es die Klasse in der Visualisierung als Gebäude nicht als einzelnen Quader darstellt, sondern diesen in weitere geschachtelte Elemente aufteilt und eine neue Darstellungsebene einführt.

1.2 Zielstellung der Arbeit

Die Stadtmetapher des Softwarevisualisierungsgenerators hat das Potential, mehrere Visualisierungsebenen darzustellen. In der Stadtmetapher werden die strukturellen Aspekte bisher lediglich auf einer grobgranularen Ebene visualisiert, wobei Klassen als monolithische Blöcke ohne spezifische Details repräsentiert werden [Zilch 2015]. Eine feingranulare Ebene für die Darstellung von Methoden und Attributen soll dieser Visualisierung nun hinzugefügt werden, indem die Blöcke aus kleineren Teilelementen

aufgebaut werden. Da für Anordnung und Formgebung der Teilelemente mehrere Ansätze existieren, werden zunächst möglicher Repräsentationen der Teilelemente gegenüber gestellt und deren Eignung für die Stadtmetapher bewertet. Zusätzlich ist die Visualisierung bestimmter Metriken wie Quelltextumfang oder Zugriffsmodifikatoren durch Eigenschaften der Teilelemente wie Größeninformation, Anordnung und Farbgebung zu bestimmen. Nach der Analyse der zusammengetragenen Ansätze werden zwei für die Stadtmetapher des Generators geeignete Varianten der Darstellung der Teilelemente bestimmt und implementiert. Die Diskussion der Metaphern sowie die Implementierung der Darstellung von Klassenbestandteilen erfolgt hierbei lediglich für den statischen Aspekt von Software. Auf eine Betrachtung des dynamischen, sowie des historischen Aspekts wird verzichtet.

1.3 Methodisches Vorgehen

In dieser Arbeit kommen mehrere Forschungsmethoden der Wirtschaftsinformatik zum Einsatz [vgl. Wilde/Hess 2007]. Der erste Teil der Arbeit wird eine konzeptionell- und argumentativ-deduktive Analyse beinhalten, in der die verschiedenen Ansätze der Visualisierung von Klassenfunktionen und -attributen analysiert und gegenübergestellt werden. Im zweiten Teil werden nach der Evaluation der Visualisierungen zwei Darstellungsvarianten definiert und die Stadtmetapher des Generators wird um diese Darstellungen von Klassenbestandteilen prototypisch erweitert. Diese neuen Generatorkomponenten werden zum Abschluss der Arbeit evaluiert und eine Praxistauglichkeit des Konzepts anhand der Visualisierung des bereits bestehenden *FAMIX*-Modells *Freemind* nachgewiesen.

1.4 Aufbau der Arbeit

Nach der erfolgten Darlegung der Problemstellung sollen in Abschnitt 2.1 zunächst die Grundlagen der Softwarevisualisierung, sowie im Speziellen, die Stadtmetapher erläutert und ihre Zweckmäßigkeit beschrieben werden. Die darauf folgenden Abschnitte dieses Kapitels beinhalten eine Diskussion bestehender Stadtmetaphern hinsichtlich ihrer Visualisierung von Klassen und Klassenbestandteilen. Zusätzlich werden die Möglichkeiten der Implementierung einzelner Teilaspekte der Visualisierungen für den Generator Gegenstand der Diskussion sein. In Kapitel 3 wird zunächst der Generator vorgestellt und anschließend werden verschiedene Ansätze zur Darstellung von Klassenbestandteilen im Rahmen der Stadtmetapher des Generators definiert. Eine Beschreibung sowie die Evaluation der Implementierung zweier Varianten dieser Darstellungsmöglichkeiten folgen in den anschließenden Abschnitten. Abschließend werden in Kapitel 4 die Ergebnisse der Arbeit zusammengefasst und es wird ein Ausblick auf weiterführende Thematiken gegeben.

2 Darstellungen von Klassenbestandteilen

In den ersten beiden Abschnitten werden zunächst die Softwarevisualisierung und die Stadtmetapher als spezifische Visualisierungsform erläutert. In den darauffolgenden Abschnitten wird im Wesentlichen auf die verschiedenen Möglichkeiten der Methoden- und Attributdarstellung von Klassen eingegangen und die Unterschiede der Ansätze werden näher beleuchtet. Weitere Entitäten und Darstellungen wie beispielsweise Instanzen oder Makros seien hierbei nur am Rande erwähnt, da dies nicht Schwerpunkt der Arbeit ist. Aufgrund der Vielzahl der Projekte wird die Beschreibung der existierenden Projekte in drei Unterkapitel gegliedert und nach der Interpretation des Gebäudes im Kontext der Stadtmetapher differenziert.

2.1 Softwarevisualisierung

Software hat die Eigenschaft, abstrakt zu sein. Sie ist nicht greifbar, sichtbar oder auf andere Weise erfahrbar. Nur eine Repräsentation von Software macht es möglich, dieses Konstrukt zwischen Menschen kommunizieren zu können (vgl. [Chapin & Lau 1996]). Da sich die Blickwinkel auf Sachverhalte von Person zu Person unterschieden, nutzen Sie auch verschiedene Hilfsmittel, um diese zu verstehen. Einige Menschen brauchen Zahlen, andere nutzen abstrakte Formeln, aber die meisten von ihnen müssen die Informationen als Farben, Formen und Figuren visualisiert sehen, um ein Verständnis dafür entwickeln zu können (vgl. [Balogh & Beszédes 2013]). Diese visuellen Objekte stellen in der Visualisierung Attribute von Datensätzen dar und werden von [Borgo u. a. 2013] als Glyphen definiert. Dabei sollten nach [Charters u. a. 2002] die Glyphen möglichst das Wissen nutzen, dass Nutzer bereits besitzen. Die Verwendung von bekannten Formen und Figuren aus der erfahrbaren Welt erleichtern es, spezifische Sachverhalte in der Visualisierung zu identifizieren.

Die Abbildung eines Teiles, oder des gesamten Programmmodells mithilfe von Glyphen wurde von [Panas u. a. 2003] als Metapher definiert. Es existieren eine Reihe verschiedener Metaphern, die unterschiedliche Glyphen zur Veranschaulichung der Software nutzen. Die Vielfalt an bereits bestehenden Metaphern resultiert aus den unterschiedlichen Anforderungen der Interessengruppen wie beispielsweise Projektleiter, Designer oder Entwickler. Jede dieser Gruppen benötigt differenzierte, strukturierte Informationen und Perspektiven über Softwaresysteme, die sich teilweise deutlich voneinander unterscheiden. Die folgende Tabelle 1 soll einen Überblick der Interessenschwerpunkte für bestimmte beteiligte Personen geben:

Metrik/Analyse	Projekt-leiter	Architekt/ Designer	Entwickler	Wartung/ Umstruktu-rierung
Ausführungszeit			X	X
Ausführungshäufigkeit			X	X
Quelltextzeilen	X	X	X	X
Unsichere Funktionsaufrufe			X	X
Globale Variablen		X	X	X
new-delete-Anweisungen			X	X
Zyklomatische Komplexität	X		X	X
Arithmetische Komplexität			X	X
Mustervergleich		X	X	X
Klassenzugehörigkeit			X	X
Stark-verbundene-Komponenten		X	X	X
Arbeitsverteilung	X			
Häufige Änderungen	X		X	X
Fehlerhafte Abhängigkeiten	X			X

Tabelle 1: Korrelation Analyse-Projektbeteiligte [Panas 2007]

Bei diesen strukturierten Informationen handelt es sich überwiegend um sogenannte Metriken, die [Böhme & Freiling 2008] als mathematische Funktionen definiert, die ein System aus einer Menge von Systemen auf einen Wert aus dem möglichen Ergebnisraum abbilden. Konkret werden diese Werte aus statischen Analysen der Softwaresysteme ermittelt. Beispielsweise erhält man durch einfaches Zählen der Zeilen innerhalb einer Datei bereits die Quelltextzeilen-Metrik für diese Datei. Eine Möglichkeit, diese Metriken in die Visualisierung miteinzubeziehen, ist die sogenannte Identitätsabbildung (engl. *identity mapping*) [Wettel 2010, 31f]. Dabei werden Glypheneigenschaften, wie zum Beispiel Größe, Form und Farbe, definierten Metriken zugeordnet und durch Variieren eben dieser Eigenschaften die Werte der Metrik ausgedrückt. Eine häufig verwendete Zuordnung ist die Assoziation der Größe einer Glyphe mit der Anzahl an Quelltextzeilen (engl. *lines of code,* LOC). Je größer die Glyphe in der Visualisierung dargestellt wird, desto größer ist die repräsentierte LOC des Quelltextfragments.

Aus der Tabelle gehen nicht nur die Schwerpunkte der Interessengruppen hervor, sondern es werden auch vielfältige Anwendungsmöglichkeiten von Softwarevisualisierung deutlich. Die Reichweite als unterstützendes Instrument erstreckt sich vom generell besseren Verständnis des Softwaresystems durch eine konkretisierende Sichtweise, über die Aufdeckung von Flaschenhälsen bei der Datenausführung bis hin zur Identifizierung von schlechter Architektur oder Design (vgl. [Maletic u. a. 2002]).

2.2 Die Stadtmetapher

Die Stadtmetapher ist eine Visualisierungsform in der Softwarevisualisierung und kann als dreidimensionale Realwelt-Metapher klassifiziert werden. Das charakteristische und namengebende Merkmal bei dieser Metapher ist ihre Eigenschaft, Softwaresysteme als Städte darzustellen. Dafür werden bekannte Objekte aus Städten der erfahrbaren Welt in der Metapher als Glyphen verwendet, um damit Bestandteile des Softwaresystems und ihre Eigenschaften zu definieren und abzubilden. Innerhalb der Stadtmetapher bildet das Gebäude wohl die signifikanteste Glyphe und verkörpert in den existierenden Stadtmetaphern hauptsächlich entweder eine Klasse oder die Methode einer Klasse. Neben den Gebäuden existiert meist ein Bereich, der Gebäude voneinander abgrenzt und sich als Stadtteil, Distrikt oder in manchen Fällen auch als ein Vorgarten interpretieren lässt. Diese Bereiche stellen hierarchisch die übergeordnete Ebene der Datei- beziehungsweise Verzeichnisstruktur dar. In Szenarios, in denen das Gebäude mit einer Klasse assoziiert wird, ist es überwiegend der Fall, dass die Bereiche diejenigen Pakete repräsentieren, in denen die Klassen definiert sind. Somit entsteht durch die verschachtelte Darstellung der Glyphe eine Repräsentation der Datei- und Verzeichnisstrukturen mit ihren jeweiligen Inhalten.

Stadtmetaphern visualisieren stets mindestens zwei Softwareartefakte, wie Klassen oder Methoden. Dies charakterisiert sie als sogenannte Multiaspekt-Metapher (engl. *multi-aspect*), was sie von den Einzelaspekt-Metaphern abgrenzt. Einige in den folgenden Kapiteln und Abschnitten thematisierte Stadtmetaphern schaffen durch Erweiterung mit einer oder sogar mehreren zusätzlichen Ansichten zusätzliche Dimensionen der Visualisierung. Die wird meist dadurch bewerkstelligt, dass diese Ansichten erst bei näherem Heranzoomen und Interaktion mit einer Glyphe zugänglich werden. Die ist beispielsweise im Projekt *Software World* in Abschnitt 2.3.3 der Fall, worin eine Innenansicht von Gebäuden geöffnet werden kann, die wiederum neue Glyphen sichtbar macht. Diese Möglichkeiten der Erweiterung von Visualisierungen lässt die Metaphern in Einzelansicht- (engl. *single-view*) und Mehrfachansicht-Visualisierungen (engl. *multiple-view*) differenzieren.

Im Rahmen des in dieser Arbeit zu behandelnden statischen Aspekts sind die quelltext-, die klassen- und die architekturzentrierte Visualisierung zu unterscheiden, wobei sich letztere noch einmal in die drei Unterkategorien Softwareorganisation, Beziehungen in der Software und metrikzentrierte Visualisierung aufteilen lässt. Klassenebenen-Visualisierungen sind problemspezifisch und richten sich in erster Linie an Entwickler. Visualisierungen auf der Architekturebene sind abstrakter gefasst als auf Quelltext- und Klassenebene. Sie bieten einen Überblick, letztlich eine Zusammenfassung des gesamten

Softwaresystems, mithilfe dessen architektonische und designbezügliche Entscheidungen schneller kommuniziert werden können (vgl. [Panas 2007]).

Die Darstellung der Klassen und ihrer Metriken sowie deren Methoden unter Einbeziehung von Paketen und ihrer Hierarchie, ordnet die Stadtmetapher in die Kategorie der architekturzentrierten Visualisierung ein. Da die geordnete Darstellung von Methoden und Attributen innerhalb ihrer Klassen und die dazugehörigen Metriken den Hauptaspekt der in dieser Arbeit thematisierten Visualisierung der Stadtmetapher darstellen, lässt sich diese vorwiegend der softwareorganisatorischen und metrikzentrierten Visualisierung zuordnen. Die Stadtmetapher selbst kann jedoch potentiell auch unter anderem Methodenaufrufe durch eine zusätzliche Erweiterung darstellen und weist damit ebenfalls Eigenschaften der Unterkategorie Beziehungen in der Software auf (vgl. [Caserta & Zendra 2011]).

2.3 Methoden als Gebäude

Ein häufig anzutreffender Ansatz der Methodenvisualisierung ist der, die Methoden einer Klasse als Gebäude darzustellen. Einige dieser Konzepte werden in den folgenden Abschnitten erläutert.

2.3.1 ImSoVision

ImSoVision (*IMmmersive SOftware VISualizatION*) ist ein Framework, das eine virtuelle Umgebung nutzt, um natürliche Repräsentationen bestimmter Quelltextmetriken visualisieren zu können. Mittels einer speziellen auf *UML* (*Universal Modeling Language*) basierenden Beschreibungssprache *COOL* (*Language for Comprehending OO software*) werden heterogene Daten wie Klassen, Beziehungen und quantitative Informationen abgebildet und später in verschiedenen Metaphern angezeigt (vgl. [Maletic u. a. 2001] und [Malhan & Singh 2014]).

Das Basiskonstrukt einer objektorientierten Programmiersprache bildet die Klasse und wird in der Metapher als eine Plattform dargestellt, wobei deren visuelle Größe mit der physischen beziehungsweise metrischen Größe der Klasse korreliert (vgl. Abbildung 1). Die Berechnung der darzustellenden Größe erfolgt anhand der Summe von Anzahl an Attributen und Methoden einer Klasse. Somit ist die Größe der Plattform ein visuelles Maß für die Komplexität einer Klasse. Attribute und Methoden werden durch individuelle Glyphen visualisiert. Attribute haben die Form von Sphären, während Methoden durch rechteckige Säulen dargestellt werden. Die Größe der Sphären spiegelt den Speicherverbrauch der Attribute wieder, während die Höhe der Säulen die Anzahl an Quelltextzeilen der assoziierten Methoden darstellt. Die Säulen variieren sowohl in ihrer Farbgebung als auch

ihrer Positionierung auf der zugehörigen Plattform. Dabei wird zwischen den drei Arten der Konstruktor-Methoden, Getter- und Setter-Methoden unterschieden. Konstruktor-Methoden stehen im Zentrum der Plattform und werden weiß dargestellt. Getter-Methoden sind grün und umgeben die Konstruktor-Methoden. Den äußeren Rand der Plattform besetzten die lilagefärbten Setter-Methoden.

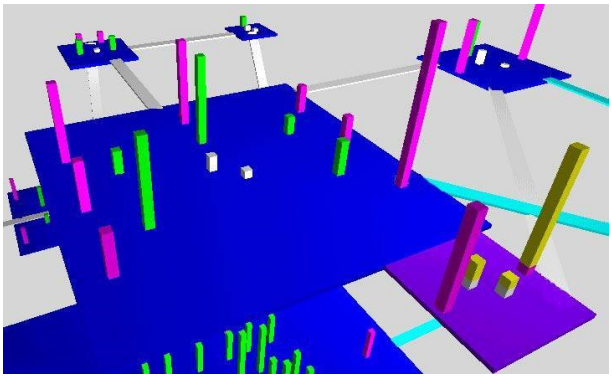

Abbildung 1: *ImSoVision* [Malhan & Singh 2014]

Um die Zugriffsmodifikatoren *private* und *public* mit in die Visualisierung einbeziehen zu können, werden beide flache Seiten der Plattform mit Sphären und Säulen besetzt. Für den Fall, dass Methoden oder auch Attribute als *public* definiert wurden, sind diese auf der Oberseite der Plattform angeordnet, für *private* auf der Unterseite. Es wird versucht, der daraus resultierenden Verdeckung der Elemente auf der gegenüberliegenden Seite, mit Semitransparenz der Plattform entgegenzuwirken.

ImSoVision bietet auch die Möglichkeit, Beziehungen zwischen den Elementen darzustellen. So wird beispielsweise bei überladenen Methoden der obere Teil der Säule gelb eingefärbt und Aggregationen werden mit einer weißen und Kompositionen mit einer hellblauen Verbindungslinie gekennzeichnet.

Die in *ImSoVision* abgebildeten Aspekte legen einen verstärkten Fokus auf die Darstellung der Methoden. Außerdem ist die Nutzung der Ober- und Unterseite der Plattformen eine interessante Möglichkeit, den Visualisierungsraum zu erweitern, was sich jedoch nachteilig auf die Orientierung innerhalb der Metapher bei großen Softwaresystemen auswirkt. Eine Wiederverwendung dieser Visualisierungstechnik wird daher für die spätere Erweiterung der Stadtmetapher des Generators nicht in Betracht gezogen.

2.3.2 Vizz3d - Unified City

Vizz3D [Vizz3D 2016] ist ein Framework, welches mehrere Metaphern unterstützt. Eine Variante der Visualisierung des Frameworks ist die Darstellung der Datengrundlage als Stadtmetapher. Diese Datengrundlage beschränkt sich hierbei nicht nur auf Metriken, sondern zusätzlich auch auf Analysen des Laufzeitverhaltens von Methoden und

Funktionen, um Programmabschnitte mit hohem Rechenaufwand identifizieren zu können. Programmiersprachen, wie C++, in denen die Arbeit mit Zeigern geläufig ist, sind für Speicherüberläufe (engl. *buffer overflow*) sowie im besonderen Maße auch Schutzverletzungen (engl. *segmentation fault*) anfälliger. Die Analyse des Quelltextes erfolgt ebenfalls im Hinblick auf diese Programmfehler. Der Aufbau der Stadt basierend auf den Analyseergebnissen gestaltet sich nach der folgenden Beschreibung von [Panas 2007].

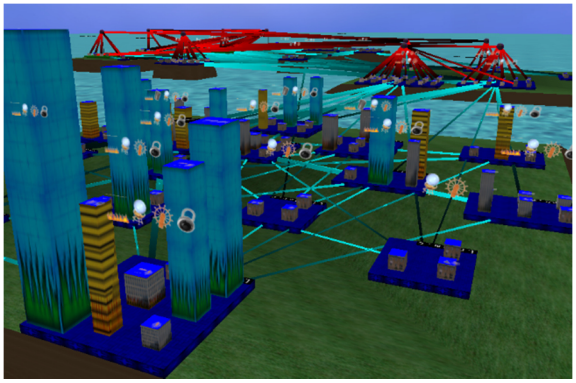

Abbildung 2: *Unified City*, Stadt [Panas u. a. 2007]

Abbildung 3: *Unified City*, Gebäude [Panas 2005]

Die Welt von *Unified City* besteht nicht wie in anderen Stadtmetaphern aus einer zusammenhängenden Ebene, die alle Elemente auf ihr verbindet (vgl. Abbildung 3). Stattdessen werden mehrere getrennte grüne Landstriche dargestellt, die Verzeichnisse abbilden sollen. Ausgehend von der Hierarchieebene der Verzeichnisse kann die Höhe der Platzierung dieser Landstriche variieren und somit die Tiefe der Verzeichnishierarchie optisch wiedergegeben werden. Auf den Landstrichen platzierte blaue Ebenen, repräsentieren Städte als Quelltextdateien. Deren Größe wird anhand der Anzahl der in ihr definierten Funktionen festgelegt. Innerhalb der Stadt befindliche Gebäude sind, im Unterschied zu vielen anderen Stadtmetaphern, nicht einfarbig, sondern mit Texturen überzogen. Diese Texturen geben Auskunft über definierte Metriken, wie zum Beispiel blaue Texturen im Falle von Funktionen mit einer hohen Anzahl von Anweisungen, oder eine Feuertextur für komplexe Bereiche (siehe Abbildung 3, rechts unten). Die Höhe der Gebäude kann ebenso unterschiedliche Werte annehmen.

Metriken werden nicht nur über Größeninformationen von Glyphen wiedergegeben. *Unified City* hat zusätzlich die Möglichkeit, auch Informationen durch projizierte 2D-Icons zu veranschaulichen. Unsichere Funktionen haben dabei ein Schlosssymbol oberhalb ihrer Gebäudemetapher (siehe Abbildung 2, links). Auch globale Variablen werden mit dieser Visualisierungsmethode als Globus dargestellt.

Im Fall von Programmiersprachen wie C++, liegen die Deklaration und Definition von Klassen in separaten Dateien vor, die eine Variabilität der Implementierung ermöglicht. Ein

„guter Programmierstil“ ist im Allgemeinen erkennbar an der Deklarierung aller Klassenmethoden in einer einzigen Quelltextdatei. Aufgrund der Neugestaltung und Umstrukturierung (engl. *re-engineering*) von Softwaresystemen ist allerdings die Gefahr der Codefragmentierung gegeben. Dabei kann es vorkommen, dass sich Definitionen von Methoden auf mehrere Dateien verteilen, was bedeutet, dass sich die assoziierten Gebäude in unterschiedlichen Städten befinden würden. Um diesen Sachverhalt veranschaulichen zu können, werden in der Stadt zusätzlich die Deklarationsdateien, auch *Header*-Dateien genannt, in Form von Säulen abgebildet. Die Verbindung der Deklaration und Definition wird nun über gleichfarbige Verbindungslinien zwischen den zwei Dateiarten visualisiert (siehe Abbildung 2, oben). Somit können auch fragmentierte Bereiche erkannt werden. Weitere Verbindungslinien in anderen Farben stehen unter anderem für Funktionsaufrufe. Das dargestellte Wasser und der Himmel dienen nur dekorativen Zwecken und haben keinerlei strukturellen, metrischen oder sonstigen Informationsgehalt.

Die Nutzung von Texturen für die Gebäudeaußenwände bietet ein großes Potential zur zusätzlichen Darstellung von Informationen, ohne weitere Glyphen in die Metapher aufnehmen zu müssen. Durch geschickte Wahl der Texturen kann eine sichtbare Trennung einzelner Gebäudesegmente erzielt werden, was einen Ansatzpunkt für die Paneel-Variante der Generatorerweiterung bietet, welche in Abschnitt 3.4 genauer erläutert wird.

2.3.3 Software World

Es gibt mehrere Möglichkeiten, genauere Aussagen über ausgewählte Artefakte von Softwaresystemen innerhalb der Visualisierung treffen zu können. Eine Variante ist die Beschränkung der Metapher auf genau diese abgegrenzten Aspekte und die Verwendung von mehreren Glyphen oder vielfältigeren Glypheneigenschaften. Dieser Ansatz hat den Vorteil der reduzierten Anzeige weniger relevanter Aspekte bei gleichzeitig stärkerem Fokus auf ausgewählte Aspekte und verringert zusätzlich die Gefahr, die Metapher zu überladen und damit eine Navigation zu erschweren. *Software World* hat jedoch einen anderen Ansatz, die Metapher um Informationen zu erweitern, aber dennoch die Navigierbarkeit zu gewährleiten. Dazu werden die methoden- und attributrepräsentierenden Gebäudeglyphen mit einer Innenansicht versehen, wodurch das Gebäude begehbar wird und sich zusätzliche Informationen zu diesen Artefakten visualisieren lassen. Eine Beschreibung der visualisierten Welt nach [Knight 2000, 85ff] sollen die anschließenden Abschnitte liefern.

2.3.3.1 Allgemeiner Aufbau

Software World soll statische Visualisierungen von Quelltexten in objektorientierten und prozeduralen Hochsprachen wie C++ und Java abbilden und das Verständnis des Quelltextes verbessern, sowie die zukünftige Entwicklung und Wartung unterstützen. Auf Basis dieser Zielstellung wurde der folgende Aufbau der Stadt realisiert.

Die Metapher bietet sechs Detaillierungsebenen: Welt, Land, Stadt, Bezirk/Stadtteil, Straße/Gebäude/Garten/Monument und Gebäude- und Garteninneres. Auf diesen verschiedenen Ebenen werden Quelltexte mit Klassen, Methoden, Attributen, Paket- und Dateistruktur und weitere Informationen dargestellt. Nachfolgend wird der Aufbau der Welt zusammenfassend beschrieben, wobei auf einige Visualisierungsobjekte, die für das Verständnis der Thematik nicht relevant sind, verzichtet wird. Eine detaillierte Beschreibung der Objekte und Prinzipien kann in [Knight 2000, 85-104] nachgelesen werden.

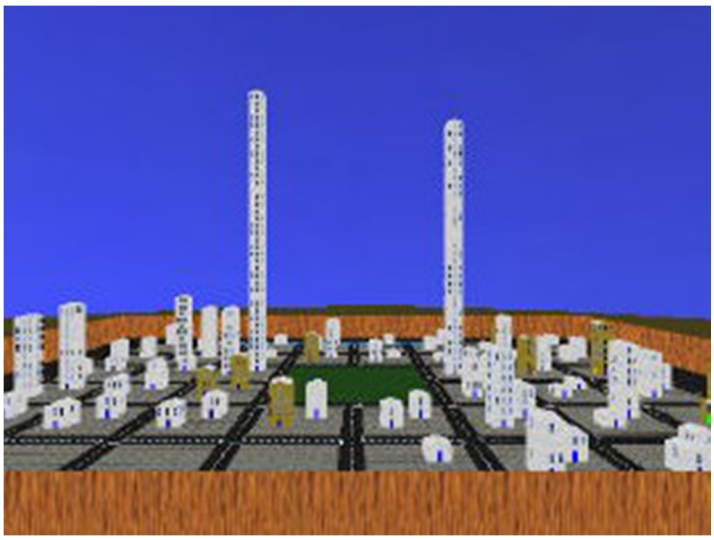

Abbildung 4: *Software World* [Knight & Munro 2001]

In der höchsten Darstellungsebene wird die Welt als eine flache zweidimensionale Übersichtskarte aus der Vogelperspektive dargestellt, die das Softwaresystem sehr grobgranular in seiner Paketstruktur, repräsentiert als Länder respektive Landmassen, wiedergibt. Die zweite Ebene bilden die einzelnen Länder/Landmassen, aufgebaut durch die Paket- und Verzeichnishierarchie des Softwaresystems. Anhand einer normalisierten Messzahl, die aus den in den Paketen enthaltenen Klassen gebildet wird, errechnet sich die Größe einer Landmasse. In der dritten Detaillierungsebene werden die Quelltextdateien als Städte abstrahiert (vgl. Abbildung 4). In den meisten Fällen kann in der Visualisierung eine Java-Datei direkt durch ihre beinhaltete Klasse abgebildet werden. Jedoch erlaubt es die Java-Spezifikation, mehrere Klassen in einer Datei zu definieren, wodurch es sinnvoll erscheint, zwei getrennte Abstraktionsebenen für Datei- und Klassenebene einzuführen. Zusätzlich zu den Klassen, die als Bezirk innerhalb einer Stadt visualisiert werden, sind in

jeder Stadt eine Stadthalle, die Dateiinformationen bereitstellt, und ein Flughafen vorhanden. Die vierte Ebene bilden die Bezirke. Diese sind zusätzlich durch Zäune innerhalb einer Stadt abgegrenzt und verkörpern die Klassen innerhalb einer Datei.

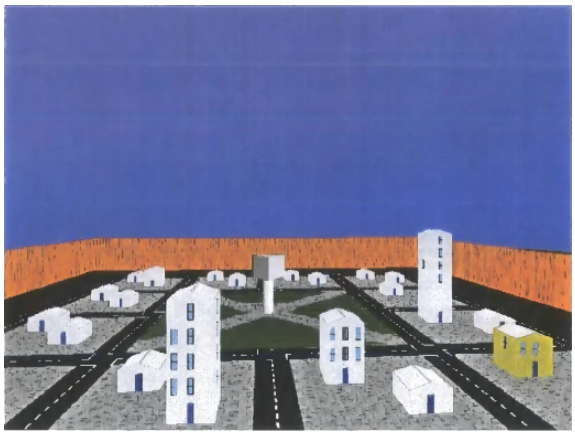

Abbildung 5: *Software World*, Stadtzentrum [Knight 2000, 161]

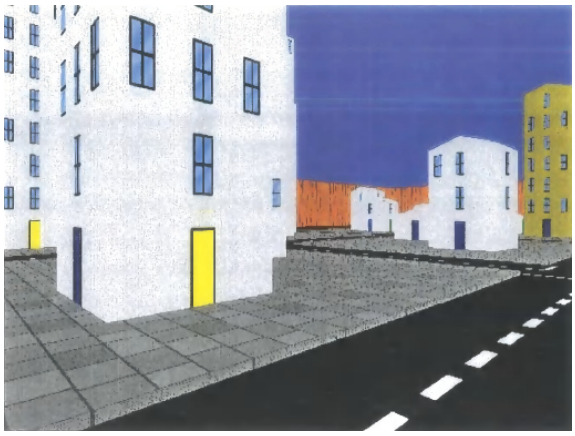

Abbildung 6: *Software World*, Gebäude [Knight 2000, 162]

Innerhalb eines Bezirks gibt es einen zentralen Garten, der sich durch zwei durchgehende diagonale Wege in vier Bereiche aufteilt. Im Schnittpunkt der Diagonalen steht ein Prisma mit dem Namen der Klassen (vgl. Abbildung 5, Mitte des Bildes). Die vier Bereiche, die durch die Diagonalen entstanden sind, geben Auskunft über Abhängigkeiten beziehungsweise importierte Klassen und welche Modifikatoren, wie beispielsweise *public*, *abstract* oder *static*, zur Definition genutzt wurden. Vererbungs- und Ableitungsinformationen können ebenfalls über dieses Element abgefragt werden. Für den Fall, dass Definitionen innerer Klassen existieren, wird dies durch zusätzliche Dollarzeichen im Namen der Klasse und das klassenrepräsentierende Prisma mithilfe von roten oberen und unteren Kanten hervorgehoben (vgl. [Knight 2000, 88-98]).

2.3.3.2 Visualisierung der Methoden

Ein weiteres signifikantes Objekt innerhalb der Bezirke bildet das Gebäude. Dieses spiegelt eine Methode der Klasse wieder und wird zur Differenzierung zwischen Methoden und Variablen als Quader dargestellt. Die Gebäude werden in einer Blockstruktur alphabetisch sortiert in ihrem jeweiligen Bezirk angeordnet. Alle Gebäude haben jeweils mindestens ein Stockwerk und einen Eingang, unabhängig davon, ob die Methode Parameter besitzt, um Zugang zu den Informationen über die Methode zu gewährleisten, die im Inneren bereitgestellt werden (vgl. Abbildung 6). Auf dem Erdgeschoss wird für den Fall, dass es mehrere Etagen gibt, ein Fahrstuhl zur Navigation in alle Stockwerke zur Verfügung gestellt. Eine Beschreibung verschiedener Eigenschaften und Informationen zu den Methoden und wie diese visualisiert werden, soll die folgende Auflistung verdeutlichen (vgl. [Knight 2000, 99-103]):

- **Methodenname:** Dieser ist neben dem Eingang auf einer Tafel vermerkt.
- **Parameter:** An den Außenwänden des Erdgeschosses verteilt werden Feuertüren angezeigt, deren Anzahl mit der der Parameter korreliert.
- **Parameternamen:** In einer frühen Version der Visualisierung, in der ausschließlich der statische Aspekt dargestellt wird, stehen die Namen der Parameter direkt im Inneren des Erdgeschosses.
- **Parametertypen:** Der Unterschied zwischen primitiven und zusammengesetzten Datentypen wird anhand der Farbe der Feuertür visualisiert; gelb für primitive Datentypen, grün für Zusammengesetzte. Eine detailliertere Typinformation steht auf einer Pinnwand im Foyer des Erdgeschosses.
- **Rückgabetyp:** Steht wie die Parametertypen auf der Pinnwand.
- **Ausnahmen** (engl. *exceptions*)**:** Außerhalb des Gebäudes sind Lampen angebracht, die geworfene Ausnahmen symbolisieren. Der Name dieser Ausnahme wird zusätzlich vom Lichtstrahl der Lampe auf dem Bürgersteig angezeigt und beleuchtet.
- **Zeilennummer:** Die Zeilennummer innerhalb der Datei, in der die Methode deklariert ist, wird auf der Tafel im Eingangsbereich wiedergegeben.
- **Zugriffsmodifikatoren:** Die Farbe des Gebäudes spiegelt den Zugriff auf die Methode wieder. Für den öffentlichen (engl. *public*) Zugriff werden das Gebäude und seine Stockwerke grau eingefärbt, im Falle des privaten (engl. *private*) Zugriffsmodifikators braun. Außerdem sind die Modifikatoren (engl. *modifier*) zusätzlich im Inneren des Gebäudes aufgeführt.
- **Quelltextzeilen** (engl. *lines of code,* LOC)**:** Ähnlich dem Projekt *Unified City* im Abschnitt 2.3.2 variiert das Gebäude in seiner Höhe abhängig vom Umfang an LOC. Alle Gebäude haben eine Mindesthöhe, welche sich pro weitere 10 LOC um ein weiteres *Stockwerk* erhöht. In der Außenansicht ist die Segmentierung der Gebäude in mehrere Stockwerke mit Hilfe von übereinanderliegenden Fenstern, ähnlich der realen Welt, dargestellt.

Darüber hinaus werden auch lokale Variablen, das heißt Variablen, die innerhalb einer Methode deklariert und verwendet werden, visualisiert und ausgewählte spezifische Informationen dargestellt. Hierzu wird für jede lokale Variable dem Gebäude ein Stockwerk hinzugefügt, welches diese Variable repräsentiert und durch den Fahrstuhl erreichbar ist. Die Bedeutung der Stockwerksglyphe unterscheidet sich somit zwischen der Außenansicht und der Innenansicht des Gebäudes. Jede Etage ist im Wesentlichen ein Büro, dessen

Informationen durch Schreibtische zugänglich sind. Durch die Nutzung dieser Untermetapher werden Quelltextkommentare und Annotationen über die Variable, wie Verwendung, beabsichtigte Funktion oder Ähnliches, bereitgestellt. In nachstehender Liste ist eine genauere Erläuterung der einzelnen visualisierten Merkmale beschrieben:

- **Name:** Dieser wird im Fahrstuhl als Navigationselement sowie auf dem Schreibtisch als Namensplakette angezeigt.
- **Typ:** Eine Schreibtischlampe dient als Indikator für den Typ der Variable. Für einen primitiven Datentyp wird ein Punktstrahler dargestellt, während für zusammengesetzte Datentypen Bankerlampen angezeigt werden. Ähnlich wie bei den Ausnahmen von Methoden, steht innerhalb des Lichtstrahls der Lampe der genaue Name des Datentyps.
- **Initialisierungswert:** Auf dem Schreibtisch existiert ein (Eingangs-) Ablagefach, auf dessen obersten vorhandenem Dokument der Wert bei Initialisierung der Variable vermerkt ist.
- **Verwendungszweck:** Informationen über die Nutzung der Variable werden dem Betrachter der Visualisierung anhand eines Buches präsentiert. Jeder dokumentierte Verwendungszweck wird einzeln auf einer eigenen Buchseite mit dazugehörigen Annotationen und einem Verweis auf die Stelle im Quelltext angezeigt. Zusätzlich existiert die Möglichkeit, den Verwendungszweck als Listenansicht in einer Inhaltsübersicht darzubieten.
- **Zeilennummer:** Die Quelltextzeile, in der die Variable deklariert wurde, wird mittels eines Abreißkalenders visualisiert.
- **Zugriffsmodifikatoren:** Der einzig zulässige Modifikator *final* wird durch ein aufgestempeltes *Final Version* auf dem obersten Dokument des Ablagefachs, auf dem auch der Initialisierungswert steht, angezeigt.
- **Felder** (engl. *array*)**:** Für den Fall, dass die Variable als Feld angelegt ist, wird an Stelle eines einzelnen Eingangsablagefachs ein Stapel Ablagefächer auf dem Schreibtisch angezeigt. Die Höhe dieses Stapels ist unabhängig von der Größe des Feldes. Genauere Spezifikationen des Feldes können dem Dokument im Eingangsablagefach, welche ganz oben auf dem Stapel an Ablagefächern positioniert wird, entnommen werden.

2.3.3.3 Visualisierung der Attribute

Zusätzlich zu den Methoden als Gebäude innerhalb der Stadt, existieren dreistöckige zylinderförmiger Monumente, die aufgrund ihrer Form mit Türmen assoziiert werden

können (vgl. Abbildung 7). Mithilfe dieser gesonderten Glyphe werden globale Variablen sowie Klassenvariablen dargestellt. Die Höhe eines Monumentes richtet sich nach der Häufigkeit der Nutzung dieser Variable. Um die Monumente bezüglich ihrer Höhe ins richtige Verhältnis setzen zu können, wird vorab eine Normalisierung über den Wert ihrer Nutzungshäufigkeit gebildet und die Höhe entsprechend gewählt. Somit ist die angezeigte Höhe der Monumente kein absolutes, sondern ein relatives Maß für diese repräsentierende Metrik.

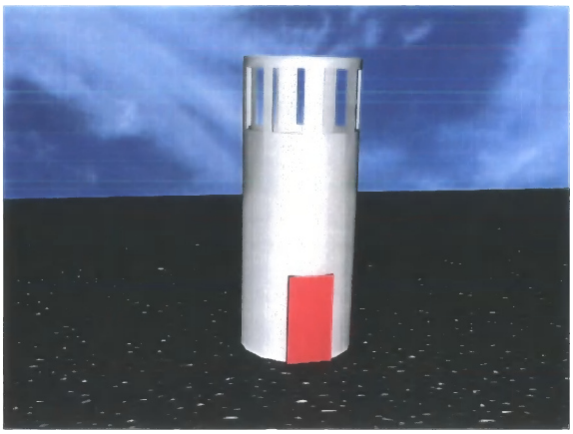

Abbildung 7: *Software World*, Monument [Knight 2000, 159]

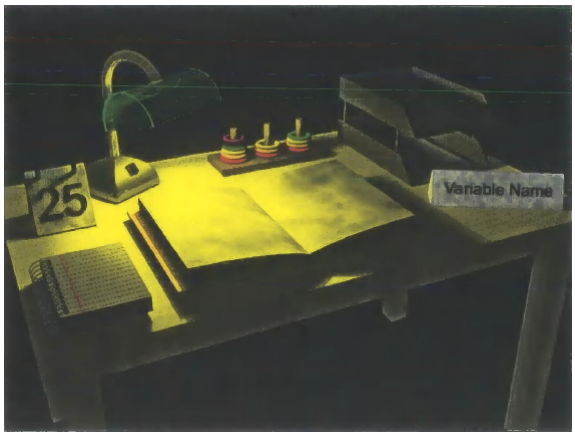

Abbildung 8: *Software World*, Schreibtisch [Knight 2000, 160]

Der Name der Variable steht auf dem oberen Ende des Monumentes, während sich anhand der Farbe des Mauerwerks zwischen primitivem und zusammengesetztem Datentyp unterscheiden lässt. Zwei weitere Eigenschaften sind zugleich außerhalb und innerhalb des Monuments erkennbar. Auf dem Dach des Monuments kennzeichnet eine Flagge den öffentlichen Zugriff auf die Variable und ein schwarzes Band rundum das Monument signalisiert die Definition der Variable als *statisch*. Zusätzlich ist die Eigenschaft als Konstante mithilfe eines Zauns, um das Monument herum, vermerkt. Innerhalb des Monumentes gibt es analog zu den Gebäuden wiederum Stockwerke, wobei es hier ein Limit von drei Etagen gibt. Im Eingangsbereich steht nochmals der Name der Variable geschrieben. In einem gesonderten Informationsraum ist ein Schreibtisch platziert, auf dem die Information zur Zeilennummer, wo die Variable innerhalb der Datei deklariert ist, angezeigt wird (vgl. Abbildung 8). Zusätzlich sind an diesem Ort Modifikatoren, Kommentare und weitere Informationen, wie die Definition als Feldvariable, hinterlegt (vgl. [Knight 2000, 98-99]).

Software World bietet unter diesen drei beschriebenen Projekten die detaillierteste Stadtmetapher mit einer Vielzahl an Glyphen und gleichzeitig darstellbaren Informationen. Die Visualisierung mehrerer Abstraktionsebenen, wie eine Außen- und Innenansicht der

Gebäude, ermöglicht eine tiefergreifende Analyse der Klassenbestandteile. Für eine schnelle Erfassung von Metriken beziehungsweise eine Reihe spezifischer Informationen bei großen Projekten scheint der Ansatz von *ImSoVision* jedoch intuitiver und übersichtlicher, da auf eine komplexe Ausgestaltung der Elemente zugunsten der Navigierbarkeit verzichtet wird. *Unified City* bietet unter anderem zusätzliche Informationen, wie beispielsweise Codefragmentierung, und eine direkte Darstellung von Funktionsaufrufen mittels Graphen, was besonders Entwickler in ihrer Arbeit unterstützen kann. Die Nutzung mehrerer zusätzlicher Ansichten für eine detailliertere Darstellung der Methoden und Attribute wird auch in anderen Metaphern genutzt, um die Möglichkeiten der gleichzeitigen Darstellung zu erweitern. Eine Implementierung einer Metapher innerhalb einer Metapher übersteigt jedoch den für die Generatorerweiterung vorgesehenen Aufwand.

2.4 Klassen als Gebäude

Der zweite, bei der Stadtmetapher oft vertretene Ansatz stellt die Klassen als Gebäude dar. Eine Auswahl an Projekten soll nun in den kommenden Unterkapiteln behandelt werden.

2.4.1 EvoSpaces / Software City

Das Gemeinschaftsprojekt *EvoSpaces* [EvoSpaces 2016] dreier Schweizer Universitäten zielt auf die Visualisierung von sich entwickelnden Softwaresystemen ab und der Möglichkeit, in dieser 3D-Umgebung multidimensional navigieren zu können. Ein Ergebnis dieses Projektes ist ein *eclipse*-PlugIn [Eclipse 2016] namens *Software City*, das die Stadtmetapher zur Repräsentation der Softwareentitäten nutzt (vgl. [Lanza u. a. 2009]).

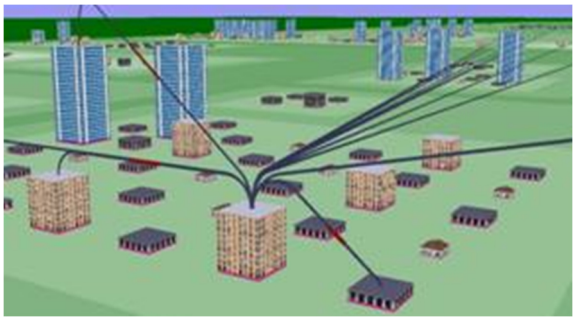

Abbildung 9: *EvoSpaces*, Außenansicht [Lanza u. a. 2009]

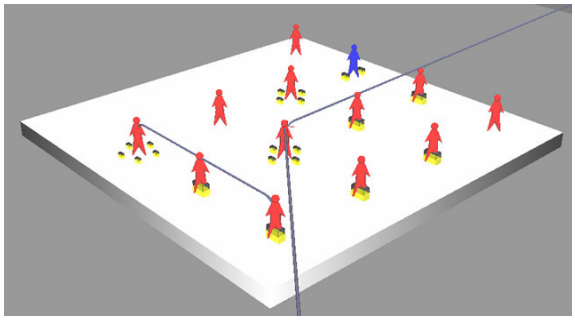

Abbildung 10: *EvoSpaces*, Innenansicht [Alam & Dugerdil 2007a]

Die folgende nähere Beschreibung basiert auf den Ausführungen von [Alam & Dugerdil 2007a] und [Alam & Dugerdil 2007b]. *Software City* stellt die Verzeichnis- beziehungsweise Paketstruktur als Distrikte dar, welche sich anhand der Untergrundfarbe unterscheiden, jedoch keine Tiefeninformation anhand einer Höhenabstufung vorsehen. Tiefere Hierarchien sind durch eine hellere Untergrundfarbe gekennzeichnet und heben sich von der umgebenden grünen Grundflächenfarbe ab. Innerhalb der Distrikte gibt es insgesamt vier verschiedene Arten von Gebäuden, wovon drei direkt Klassen respektive Dateien darstellen.

Abhängig von der ermittelten LOC in den Dateien werden diese entweder im Falle eines geringen Umfangs von 0 bis 50 LOC als Haus, zwischen 51 und 200 LOC als Apartment, und bei mehr als 200 LOC durch ein Bürogebäude dargestellt. Alle drei Gebäudetypen können sich nicht nur in ihrer Textur, sondern auch der Anzahl an Etagen unterscheiden, was die Anzahl an globalen Variablen wiederspiegelt. Dabei werden wiederum drei Kategorien differenziert: niedrig, mittel und hoch. Der vierte Gebäudetyp verkörpert eventuell vorhandene *Header*-Dateien und entspricht einem Rathaus mit Säulen und einem Strichmännchen. Die drei Größenkategorien gelten hierbei allerdings für die Anzahl an Methoden (engl. *number of methods*, NOM). Beziehungen zwischen den Klassen werden als gebäudeverbindende Rohrleitungen dargestellt, deren Richtung durch farbliche Segmente gekennzeichnet wird.

Eine detailliertere Ansicht der Inhalte einer Klasse respektive Datei bietet sich bei einem Hineinzoomen in ein Gebäude, welches eine neue Repräsentationsebene der Visualisierung öffnet. Hierbei werden drei übereinanderliegende Plattformen eingeblendet, die als Stockwerke innerhalb des Gebäudes fungieren und auf denen sich Strichmännchen befinden. Diese Strichmännchen visualisieren spezifische Inhalte einer Datei, wie Methoden oder Makros, und werden nach einem bestimmten Muster angeordnet. Auf dem untersten Stockwerk werden Methoden platziert, auf dem mittleren Funktionen und auf dem obersten alle übrigen Elemente der Datei wie Makros und Unterklassen. Die Strichmännchen unterscheiden sich zum einen in ihrer Färbung, zum anderen sind kreisförmig um die Stichmännchen herum gelbe Kisten zu finden, die jeweils einzelne lokale Variablen darstellen. Die zuvor bereits beschriebenen Rohrleitungen, welche Beziehungen zwischen den Gebäude wiederspiegeln sollen, sind auch in dieser Darstellung zu finden und verbinden sowohl die Objekte innerhalb der Klasse als auch außerhalb.

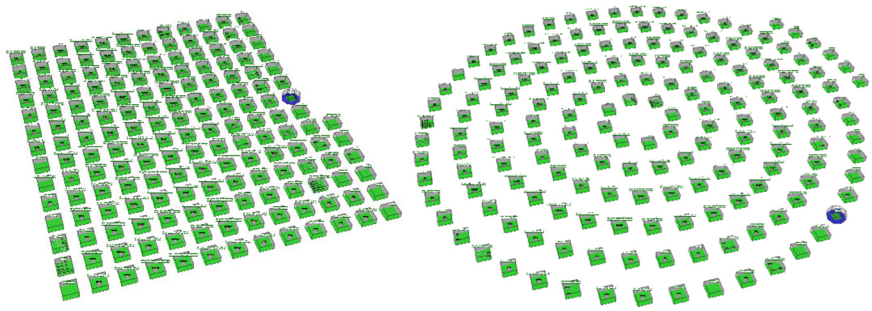

Abbildung 11: *EvoSpaces*, Layouts [Alam & Dugerdil 2007a]

Die Anordnung der Gebäude und der Strichmännchen in der Detailansicht der Gebäude entspricht wahlweise einem konsequenten Schachbrettmuster oder konzentrischen Kreisen (vgl. Abbildung 11), wodurch die Grundflächen der Elemente immer identisch sind und die Anordnung selbst keine Metrik oder Eigenschaft wiederspiegelt.

Strichmännchen als Glyphe für Methoden zu nutzen, die in einer separaten Ansicht angeordnet werden, ist ein signifikantes Merkmal in *Evospaces*. Allerdings übersteigt der Implementierungsumfang die Zielsetzung dieser Arbeit und wird daher nicht als mögliche Visualisierungsform eingesetzt.

2.4.2 SArF Map

Eine Stadtmetapher nach Paket- & Klassenhierarchien aufzubauen, ist zwar die am häufigsten vorzufindende, jedoch nicht die einzige Möglichkeit, Strukturen innerhalb des Quelltextes abzubilden. Losgelöst von einer solchen Hierarchie, existiert in Programmen oft ein hoher Grad an Kohäsion von Methoden, die im Zusammenwirken bestimmte Aufgaben bearbeiten. Solche klassenübergreifenden Interaktionen sind bei objektorientierter Programmierung häufig zu Modulen zusammengefasst und bilden Funktionseinheiten oder auch Funktionalitäten (engl. *features*) von Programmen (vgl. [Eisenbarth u. a. 2003; Kobayashi u. a. 2013]).

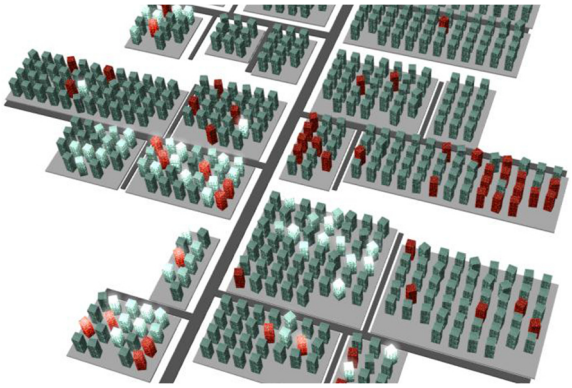

Abbildung 12: *SArF Map*, Gittermuster [Kobayashi u. a. 2013]

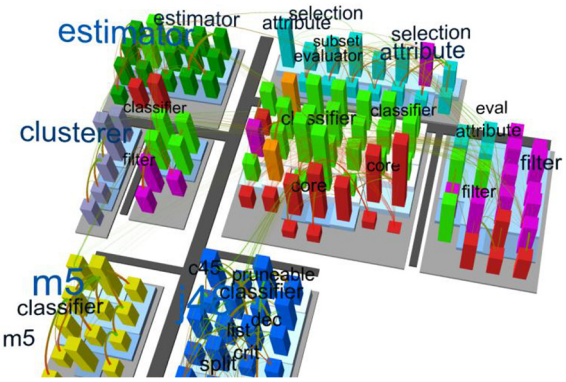

Abbildung 13: *SArF Map*, Layer [Kobayashi u. a. 2013]

Die *SArF Map*, Abkürzung für *Software Architecture Finder*, bildet statt einer Pakethierarchie eine Funktionalitätenhierarchie ab, in der einzelne Funktionskomponenten eines Programmes in durch Straßen abgegrenzte Bereiche, aufgeteilt werden. Die Visualisierung des Zusammenhangs der Komponenten bietet ein hohes Abstraktionsniveau (engl. *high-level abstraction*) von Softwaresystemen, von der ein besseres Verständnis der Systeme, besonders von Außenstehenden, erwartet wird.

Die Identifizierung der Funktionalitäten und Gruppierung (engl. *clustering*) der Klassen übernimmt der *SArF software clustering algorithm* [Kobayashi u. a. 2012]. Dieser besteht aus zwei verschiedenen Layouts, die zusammen das *Straßen-und-Block-Baum-Layout* (engl. *Street and block tree layout*) bilden. Das erste Layout ordnet die Gebäude beziehungsweise Klassen durch Straßen abgetrennten Stadtarealen zu (siehe Abbildung 12). Im zweiten Layout werden innerhalb der Stadtareale die Gebäude in unterschiedlichen Schichten (engl. *layers*) mittels Abschüssigkeiten (engl. *slopes*) repräsentiert. Zur besseren Übersicht liegt

der Gebäudeanordnung ein gleichmäßiges Gittermuster zugrunde. Die Herleitung der Schichten basiert auf Abhängigkeiten der Klassen untereinander (vgl. [Scanniello u. a. 2010]). Höher platzierte Klassen sind von darunter platzierten Klassen abhängig. Die Schichten werden in bis zu acht Abstufungen unterteilt, wobei auch ein Farbgradient zur besseren Unterscheidung Verwendung findet und mehrere Unterbereiche innerhalb des Stadtareals schafft (siehe Abbildung 13). Der Algorithmus gruppiert Gebäude in der gleichen Schicht nebeneinander. Die Identifizierung der Funktionseinheiten und ihrer Unterbereiche erfolgt über eingeblendete Wörter über den Gebäuden, welche die vorherrschenden Schlüsselwörter in den Funktionseinheiten darstellen.

Die Visualisierung setzt außer der Positionierung und Färbung der Schichten noch weitere optische Hilfsmittel zur Repräsentation von Informationen verschiedener Metriken des Softwaresystems ein. Darunter fallen beispielsweise die Verbindung der Höhe von Gebäuden mit deren Methodenanzahl oder auch Quelltextzeilenumfang, die Formgebung in Abhängigkeit des Datentyps, sowie Probleme im Quelltext der Klasse, die mithilfe von feuerähnlichen Texturen deutlich gemacht werden. Eine besondere Hervorhebung der Abhängigkeiten zwischen den Klassen ist mit der *hierarchischen-Kantenbündel*-Technik (engl. *hierarchical edge bundles*) [Holten 2006] ebenfalls möglich (vgl. [Kobayashi u. a. 2013]).

Der Ansatz, die Gebäude nicht nach Pakethierarchie, sondern nach Funktionalität zu gliedern, ist ein selten anzutreffender Ansatz zur Sortierung der Glyphen. Weiterhin bietet *SArF Map* allerdings keine neuen Visualisierungsformen, um speziell Klassenbestandteile darzustellen und hat daher für die Generatorerweiterung keine weitere Bedeutung.

2.4.3 Verso

Im Ansatz *Verso* (vgl. [Langelier u. a. 2005]) werden Pakethierarchien nicht etwa durch Höhenunterschiede von abgestuften Ebenen oder unterschiedlicher Färbung des Untergrundes differenziert wie bei anderen Beispielen in den vorherigen Kapiteln. Stattdessen werden gelbe Linien zur Abgrenzung der Gebäude eingefügt und die Hierarchien durch Gruppierung der Gebäude innerhalb der mittels dieser Linien begrenzten Areale hervorgehoben. Die Darstellung der Paketstruktur ist somit nur zweidimensional ausgeführt und lediglich die Gebäude haben eine zusätzliche Höheneigenschaft. Für die Anordnung der Gebäudeglyphen stehen wahlweise zwei verschiedene Layouts zur Verfügung.

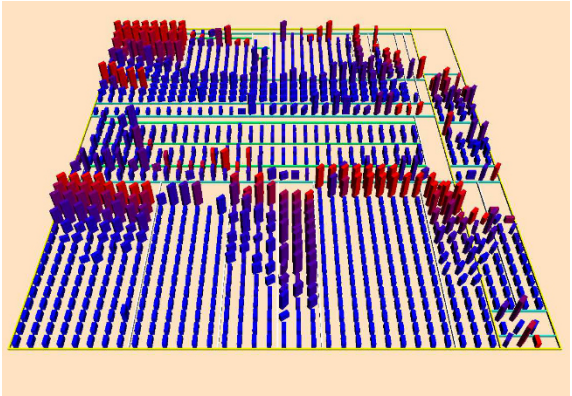

Abbildung 14: *Verso*, *Modified Treemap Layout* [Langelier u. a. 2005]

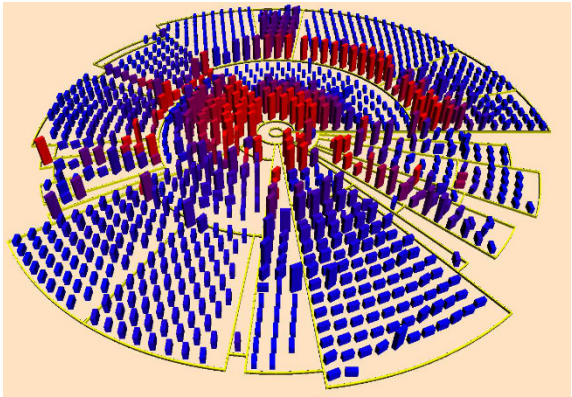

Abbildung 15: *Verso*, *Modified Sunburst Layout* [Langelier u. a. 2005]

Das erste Layout platziert die Gebäude nach einer modifizierten Variante des *Baumkarte*-Layouts (engl. t*reemap*) [Johnson & Shneiderman 1991] (siehe Abbildung 14). Das bedeutet, dass zwischen den Gebäuden hierbei ein gleichmäßiger Abstand existiert und die Anordnung einem Schachbrettmuster ähnelt. Ziel der Modifikation des *treemap*-Ansatzes ist es, die Grundfläche, auf der sich die Gebäude befinden, möglichst rechteckig formen zu können und damit eine bessere Übersicht in der Visualisierung zu erhalten.

Das *modifizierte-Sonnenausbruch*-Layout (engl. *sunburst*) basiert auf dem *sunburst*-Layout von [Zhang & Stasko 2000] (siehe Abbildung 15). Ausgehend vom Mittelpunkt, ordnen sich die Gebäude entlang konzentrisch verlaufender Kreise nach außen hin an. Radial angeordnete Separatoren trennen Nachbarpakete in derselben Hierarchieebene voneinander ab. Je weiter außen ein Gebäude platziert ist, desto tiefer geschachtelt ist seine Pakethierarchie. Anhand dieser Anordnung ist es möglich, Pakete und ihrer Unterelemente als eine Art Tortenstück eines Kuchens zu separieren.

Die Gebäude spiegeln drei verschiedene Metriken über eine jeweils andere Gebäudeeigenschaft wieder. Objektkopplungen (engl. *coupling*) sind durch einen von blau nach rot verlaufenden Farbgradienten dargestellt, wobei blau für eine geringe Kopplung steht. Eine Drehung des Gebäudes beschreibt den Grad der Methodenkohäsion während die Gebäudehöhe die Anzahl an Anweisungen (engl. *number of statements*, NOS) widerspiegelt.

Verso bietet ähnlich *SArF Map* keine neuen Visualisierungsformen, die detaillierte Eigenschaften von Klassenbestandteilen als Einzelelemente darstellen. Somit zeigt dieses Projekt keine Möglichkeiten für die in dieser Arbeit geplanten Generatorerweiterung auf.

2.5 Gebäude als ambivalente Glyphe

In den zwei nachfolgenden Abschnitten werden Projekte präsentiert, die die Gebäudeglyphe in einer Art und Weise nutzen, die nicht der Klassifizierung der vorherigen Abschnitte entspricht. Aus diesem Grund werden sie hier getrennt erläutert.

2.5.1 CodeMetropolis

Sämtliche in den vorhergehenden Kapiteln vorgestellten Projekte beziehungsweise Ansätze nutzen grundlegende 3D-Elemente und Einfärbungen eben dieser sowie teilweise einfach gehaltene Texturen zur Erzeugung der Visualisierungen. Mit dem technischen Fortschritt und der damit einhergehenden Erhöhung der Leistungsfähigkeit moderner Computersysteme können diese Beschränkungen überwunden werden und selbst fotorealistische Texturen zum Einsatz kommen. *CodeMetropolis* [CodeMetropolis 2016] nutzt die Spiel-Engine *Minecraft* [Minecraft 2016] zur Generierung einer 3D-Simulation, in der das Kernelement der Spielwelt ein Block ist. Datenvisualisierungstechniken werden oft in der Softwaretechnik genutzt, jedoch ist das relativ neue Gebiet der *Gamifizierung* (engl. *gamification*) noch weit von einer ähnlich häufigen Anwendung als Hilfsmittel entfernt (vgl. [Balogh u. a. 2016] und [Dubois & Tamburrelli 2013]). Die *Gamifizierung* verwendet Spielelemente außerhalb des Spielbereichs in einem anderen Kontext, um Lernkurven bei neuen Technologien zu steigern, und das Engagement von Nutzern sowie die organisatorische Produktivität zu verbessern (vgl. [Deterding 2012]). *Minecraft* wurde als Basis der Visualisierung aufgrund der geringen Einschränkungen hinsichtlich der Gestaltung und Navigation innerhalb der Spielwelt ausgewählt. Die nachfolgende Beschreibung der Visualisierung wurde [Balogh & Beszédes 2013] entnommen.

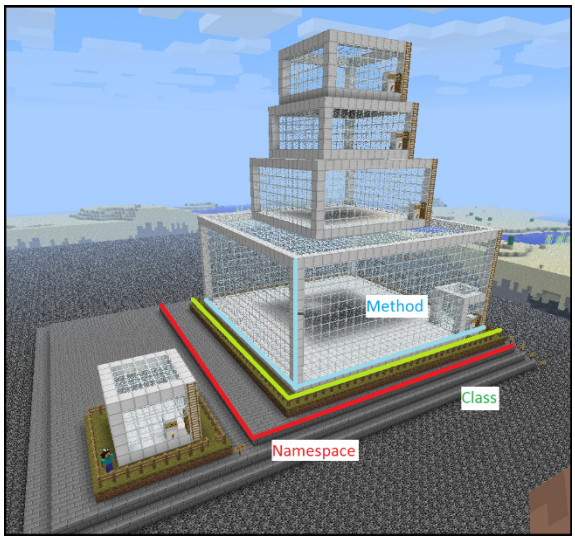

Abbildung 16: *CodeMetropolis*, Paket mit Klassen [Balogh & Beszédes 2013]

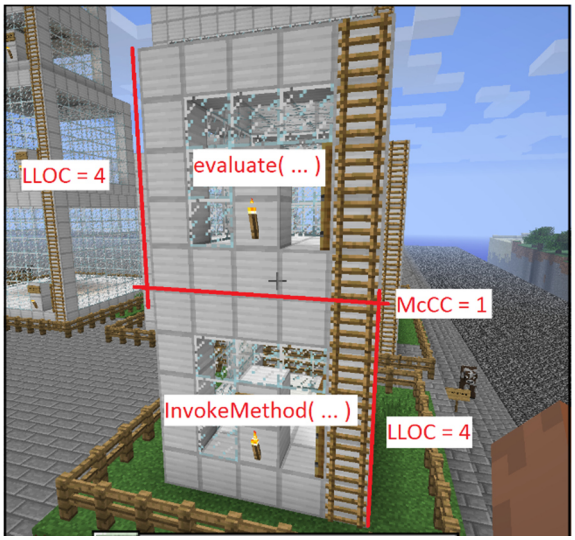

Abbildung 17: *CodeMetropolis*, Gebäudedetails [Balogh & Beszédes 2013]

Da, wie bereits erwähnt, der Block das Kernelement der Spielwelt darstellt, müssen alle Glyphen aus diesem Grundelement zusammengesetzt werden. Im Fall der Platte (engl. *plate*) als Glyphe des Verzeichnisses respektive Paketes, ist dies durch ein Rechteck aus Blöcken am Boden umgesetzt, welches gewissermaßen eine Art Fundament für alle Aufbauten wie Gebäude und weiterer Platten (siehe Abbildung 16, roter Rahmen) bildet. Die Hierarchie der Verzeichnisse spiegelt sich durch Übereinandersetzen mehrerer Platten wieder. Länge und Breite der Platten berechnen sich anhand der Abmessungen ihrer Aufbauten. Innerhalb der von den Platten begrenzten Bereiche sind Gärten ähnelnde grüne Distrikte zu finden, die zusätzlich durch Zäune abgegrenzt sind. Diese Distrikte symbolisieren Klassen. Gebäude befinden sich innerhalb von Distrikten und sind als Verbund von Bestandteilen der Klassen, wie beispielsweise Methoden, zu verstehen, die keine Bedeutung hinsichtlich des Quelltextes besitzen, sondern nur der Gruppierung der Elemente dienen.

Ähnlich wie bei *Software World* aus Abschnitt 2.3.3, bestehen die Gebäude aus aufeinander gestapelten Etagen, die hierbei jedoch von außen jederzeit ersichtlich sind. Es gibt keine gesonderte Innenansicht der Gebäude. Stattdessen werden von den einzelnen Etagen nur die Rahmen mittels grauer Blöcke angezeigt und als Wände gläserne Blöcke eingefügt. Lediglich der Boden der Etage ist mit grauen Blöcken als Fußboden ausgefüllt. Verschiedene Metriken der Methoden sind über die Abmessungen der Etagen abgebildet. Dabei gibt die Grundfläche, also Länge mal Breite, Auskunft über die zyklomatische Komplexität (engl. *cyclomatic complexity*), auch McCabe-Metrik genannt, während die Höhe die NOS abbildet (siehe Abbildung 17). Jede Entität ist zur Identifizierung mit einem Namensschild oder Beschriftung versehen.

Die Weiterentwicklung von *CodeMetropolis* bietet die Möglichkeit des Betretens der verschiedenen Etagen über eine Wendeltreppe innerhalb dieser Glyphen, die alle Etagen miteinander verbindet [Balogh u. a. 2015]. Dadurch entfällt die Notwendigkeit einer Leiter außerhalb des Gebäudes (siehe Abbildung 17). Eine jüngere Version erweitert die Welt um eine Darstellung sogenannter Außenposten, um Testfälle und Testsuiten hervorzuheben, welche die aus Methoden bestehenden Häuser beschützen (vgl. [Balogh u. a. 2016]). Außerdem wird die Repräsentation von Attributen in Form von Kellern unterhalb der Gebäude angedeutet.

CodeMetropolis hat durch seine Vielfalt an Glypheneigenschaften einige Ansätze zu bieten, die für die Generatorerweiterung interessante Möglichkeiten aufzeigen. Beispielsweise ist die Stapelung der Methoden aufeinander ein effizienter Weg, die Methodeneigenschaften darstellen zu können, ohne eine separate Ansicht dafür schaffen zu müssen. Das Modell ähnelt dabei dem von Software World aus Kapitel 2.3.3, jedoch wird

die Detailansicht der Methoden bei einer Interaktion mit der Glyphe nicht neu skaliert oder gar vollkommen neu gestaltet. Auch die unterschiedliche Dimensionierung der Etagen ist eine Eigenschaft, die in einem anderen Kontext in der Paneel-Variante der Generatorerweiterung weiterverwendet wird.

2.5.2 CodeCity

Ein auf dem *Moose Framework* [Nierstrasz u. a. 2005] aufbauendes Visualisierungstool ist das Projekt *CodeCity* [CodeCity 2016], dessen folgende Beschreibung sich auf [Wettel 2010] bezieht. Das Basiselement, wodurch der Großteil der Entitäten dargestellt wird, ist hierbei ein rechteckiger, einfarbiger Block. Auch bei dieser Metapher wird die Grundfläche der visualisierten Welt zuerst mit den Paketen beziehungsweise Verzeichnissen wiederspiegelnden Distrikten belegt. Die Anordnung der Distrikte entspricht keinem Raster, sondern wird unter Zuhilfenahme eines Algorithmus bestimmt, der die Abmessungen und Position der Distrikte anhand des Platzbedarfs der sich darauf befindenden Elementen wie Unterdistrikte und Gebäude rekursiv errechnet (vgl. [Wettel 2010, S. 33ff]). Damit wird eine Reduktion des Bedarfs der Grundfläche für die Stadt erreicht.

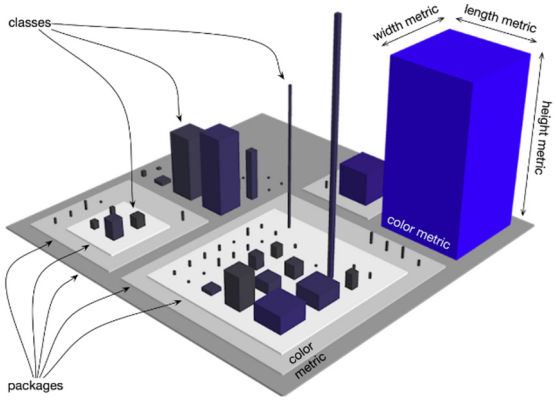

Abbildung 18: *CodeCity*, Klassen [Wettel 2010, 29]

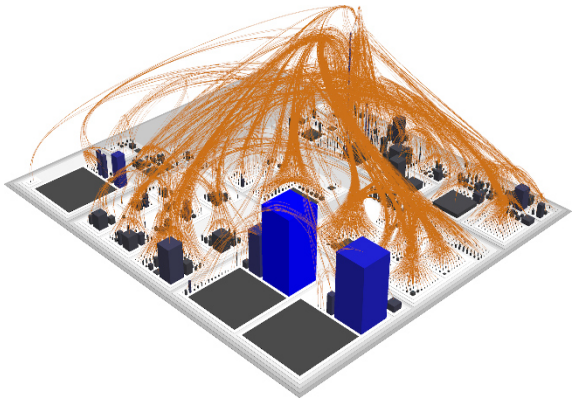

Abbildung 19: *CodeCity*, Kohäsion [Wettel 2010, 42]

Eine variable Zuordnung der Metriken zu den grafischen Attributen der Glyphen erlaubt verschiedene Konfigurationsmöglichkeiten, wobei die hier beschriebene Variante die am häufigsten genutzte Zuordnung veranschaulicht. Ferner ist *CodeCity* imstande, zwei unterschiedliche Varianten zu generieren, die sich in der Darstellung der Klassen und der Klassenbestandteile unterscheiden. In der grobgranularen Variante (siehe Abbildung 18) werden die Klassen als Gebäude dargestellt, ohne weitere Differenzierung ihrer Bestandteile wie Methoden oder Klassenattribute. Dabei ist die Grundfläche eines Gebäudes durch die Anzahl der Klassenattribute (NOA) definiert und die Gebäudehöhe über die Anzahl an Methoden (NOM). Zusätzlich färbt sich das Gebäude zunehmend von grau zu blau, je zahlreicher die LOC der Klassen sind. Eine Anzeige der Methodenaufrufe ist mithilfe der *hierarchical edge bundles* [Holten 2006] gelöst (siehe Abbildung 19, orangene Linien).

Im Kontext einer Analyse auf *Design-Disharmonien* (engl. *design disharmonies*) kann die Farbgebung der Gebäude variieren (vgl. [Lanza u. a. 2005]). So ist etwa beispielsweise vorgesehen, sogenannte *Gott-Klassen* (engl. *god classes*), *Gehirn-Klassen* (engl. *brain classes*) und *Daten-Klassen* (engl. *data classes*) farblich unterschiedlich hervorgehoben anzeigen lassen zu können.

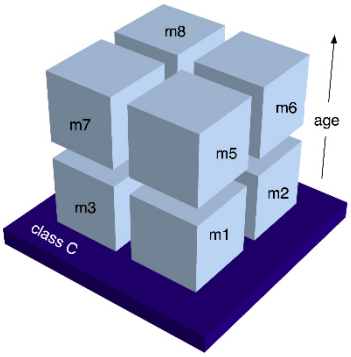

Abbildung 20: *CodeCity*, Backsteine [Wettel 2010, 38]

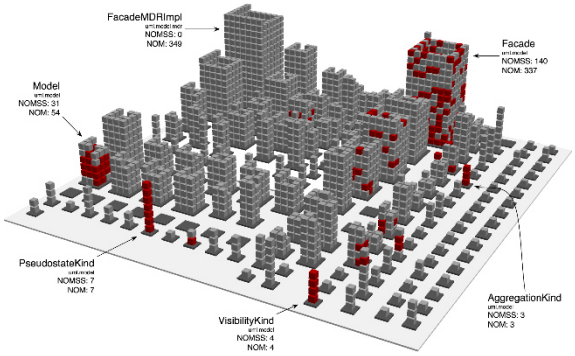

Abbildung 21: *CodeCity*, Stadt mit Backsteinen [Wettel 2010, 89]

Eine zweite, feingranulare Variante der Visualisierung spaltet das Gebäude in kleinere Untereinheiten auf (siehe Abbildung 20). Diese Backsteine (engl. *bricks*) stehen für die Methoden der Klasse und haben allesamt identische Abmessungen. Die Anordnung der Backsteine erfolgt von unten nach oben, angefangen mit den ältesten Methoden, ausgehend von ihrem Deklarationszeitpunkt. Es existieren wiederrum zwei unterschiedliche Unterlayouts für die Positionierung der Backsteine. Das *brick layout* definiert die Grundfläche fest auf 2x2 Backsteine und weitere Backsteine werden nach gleichem Schema weiter oben darauf gestapelt, während das *progressive brick layout* die Grundfläche bezogen auf die gesamte Backsteinanzahl vorberechnet. Je mehr Backsteine das Gebäude beinhaltet, desto größer wird auch die Grundfläche gewählt. Daraus resultiert ein proportioniertes Gebäude, das nicht übermäßig in seiner Höhe im Verhältnis zu seiner Breite und Länge wächst. Die einzelnen Backsteine können analog zur grobgranularen Variante ebenfalls definierte Disharmonien über ihre Färbung wiederspiegeln (siehe Abbildung 21). [Wettel 2010, 88] sieht beispielsweise die Markierung von Methoden vor, die besonders interessiert an Daten anderer Klassen sind (engl. *feature envy*), oder auch Methoden, deren Änderung zahlreiche weitere Änderungen an anderen Methoden und/oder Klassen erfordern würde (engl. *shotgun surgery*).

Aufbauend auf *CodeCity* existieren Forschungsarbeiten beziehungsweise Projekte, die den Ansatz weiterverfolgen und im Beispiel von *Manhattan* [Lanza u. a. 2013] ein *eclipse*-PlugIns realisieren, bei *SkyscrapAR* [SkyscrapAR 2016] und [Souza u. a. 2012] die Möglichkeiten der *Erweiterten Realität* (engl. *augmented reality*) nutzen, oder in *Software*

Map [Bohnet & Döllner 2011] speziell Managern Informationen über Änderungen im Code zur Verfügung stellen.

Da die Basis der Stadtmetapher des Generators auf *CodeCity* beruht, erscheint es praktisch, die Erweiterung der Darstellungen für die Methoden in *CodeCity* auch für den Generator umzusetzen. Dabei wird allerdings diese Visualisierung nicht nur für die Methoden, sondern zugleich auch für die Attribute von Klassen umgesetzt. Zusätzlich zu dem hier erwähnten Algorithmus zur Platzierung der Backsteine wird eine Variante implementiert, die die Basisfläche anhand anderer Metriken berechnet, um auch die Attribute miteinbeziehen zu können. Eine genaue Beschreibung der Generatorerweiterung folgt im Kapitel 3.

3 Erweiterung des Softwarevisualisierungsgenerators

In diesem Kapitel soll neben der Vorstellung des Softwarevisualisierungsgenerators die Definition der zu implementierenden Darstellungen der Klassenbestandteile erfolgen. Die darauf folgenden Abschnitte werden eine Beschreibung der Implementierung beinhalten, und eine Evaluation der Erweiterung schließt das Kapitel ab.

3.1 Der Softwarevisualisierungsgenerator

Der Generator ist ein Projekt der Forschungsgruppe *Softwarevisualisierung in 3D und Virtual Reality* zur interaktiven Darstellung von Softwareartefakten. Dieser wurde entwickelt, um mittels diverser Metaphern unterschiedliche Sichten auf die zu analysierende Software bieten zu können. Zur leichteren Erweiterung um neue Metaphern wurde der Generator nach dem generativen und modellgetriebenen Programmierparadigma implementiert und folgt einem vorgegebenen Ablauf von Transformationsschritten und Zwischenergebnissen (vgl. [Müller u. a. 2011]).

Die zu analysierenden und visualisierenden Softwareartefakte werden durch eine Reihe von Modell- und Texttransformationen in ein *Extensible 3D*-Modell (X3D) [X3D 2016] überführt. Ausgangspunkt für den Generator sind Daten aus formalen Quellen, welche durch einen *FAMIX*-Parser [Ducasse u. a. 2011] in das sprachunabhängige *FAMIX*-Format umgewandelt und je nach Konfiguration weiteren Analysen und Filterungen unterzogen werden. Für die zu visualisierende Metapher des Generators ist ein Metamodell notwendig, das durch das Framework *Xtext* [Xtext 2016] beschrieben wird. Zusätzlich wird der *Java*-Dialekt *Xtend* [Xtend 2016] verwendet, um die Datenmodelle zu modifizieren und die Transformation zu beschreiben. Den letzten Schritt stellt die Transformation in ein *X3D*-Modell dar, eine Beschreibungssprache für 3D-Inhalte, mit der nicht nur die Darstellung von

3D-Objekten ermöglicht wird, sondern zusätzlich auch Funktionen wie Navigation, Animation und Interaktion angeboten werden.

3.2 Anforderungsanalyse der Teilelemente

Die bisher implementierte Stadtmetapher wurde im Rahmen der Bachelorarbeit von Denise Zilch [Zilch 2015] nach der Vorlage *CodeCity*, welche im Abschnitt 2.5.2 beschrieben wurde, umgesetzt. Der Implementierungsumfang beinhaltet die Visualisierung der Klassen als Gebäude mit ihrer Anzahl an Attributen als Breite und Länge der Gebäudegrundfläche sowie der Methodenanzahl, welche als Gebäudehöhe repräsentiert wird. Die Klassen befinden sich auf abgestuften Flächen, welche die Pakethierarchie verkörpern, in der die Klassen definiert sind, wobei Ebenen sich erhöhen, je tiefer die Pakethierarchie reicht. Auf dieser Basis wird nun die Stadtmetapher um eine Darstellung von Methoden und Attributen der Klassen erweitert. Dabei erfolgt die Erweiterung in zwei getrennten Varianten, die die Klassenbestandteile mithilfe jeweils unterschiedlicher Glyphen visualisieren. Beide Varianten sind als eigenständige Metaphern zu verstehen und können nicht gleichzeitig visualisiert werden. Dem Nutzer des Generators bietet sich damit die Möglichkeit, zwischen beiden Darstellungsformen wählen zu können.

Die erste Variante übernimmt das Konzept der Darstellung von Methoden aus *CodeCity* und ist damit die logische Weiterführung der Stadtmetapher des Generators. Gebäude werden somit in einzelne Segmente, die Backsteine, unterteilt. Diese Backsteine repräsentieren wahlweise Methoden und/oder Attribute, je nachdem, welche Konfiguration[1] für den Generator gewählt wird. Der Algorithmus zur Positionierung der Backsteine wird ebenfalls von *CodeCity* adaptiert, jedoch im Detail variabel gestaltet, um auch für die getrennte oder gleichzeitige Anzeige von Attributen eine Skalierbarkeit zu gewährleisten.

Die zweite Variante nutzt für die Visualisierung der Klassenbestandteile ebenfalls den Ansatz der Segmentierung des Gebäudes. Allerdings werden die Segmente hierbei in variabler Größe gestaltet und lediglich eindimensional in die Höhe gestapelt. Dies stellt einen Unterschied zur Anordnung der Backsteine in der ersten Variante dar, die drei Dimensionen zur Anordnung nutzt. Impulsgeber dieses Ansatzes sind die in Kapitel 2 beschriebenen Projekte. In *Unified City* sind Gebäude mit Texturen belegt, die eine vertikale Segmentierung des Gebäudes suggerieren, während *Software World* fensterähnliche Formen an den Gebäuden erkennen lässt und dadurch der Eindruck entsteht, man könne Etagen des Gebäudes von außen erkennen. *CodeMetropolis* segmentiert die Gebäude von Beginn an in

[1] Eine genauere Beschreibung der Konfigurationsmöglichkeiten findet sich im Anhang.

einzelne Etagen mit durchsichtigem Mauerwerk und variiert deren Größe in allen Dimensionen.

Bestandteil der Erweiterung sollen neben der Anzeige der Elemente, auch eine Möglichkeit der Sortierung von Methoden und Attributen, sowie eine farbliche Hervorhebung bestimmter Eigenschaften sein. Eine Beschreibung der Implementierung beider Varianten folgt in den anschließenden Abschnitten.

3.3 Erweiterung des Metamodells

Das bereits bestehende Metamodell der Stadtmetapher kann für die Implementierung als Ausgangsbasis übernommen werden. Eine Ergänzung des Modells um einige Elemente ist jedoch nötig, um später die Klassenbestandteile visualisieren zu können. In Abbildung 22 ist das erweiterte Klassenmodell des Metamodells[2] dargestellt, wobei zugunsten eines besseren Verständnisses weniger relevante Definitionen ausgelassen wurden. Die Abbildung zeigt Elemente der bereits durch [Zilch 2015] umgesetzten Elemente weiß unterlegt. Neu hinzugefügte Definitionen, die für beide Varianten der Erweiterung genutzt werden, sind grün eingefärbt. Zusätzlich heben blaue Markierungen diejenigen Definitionen hervor, die von der Paneel-Variante benötigt werden.

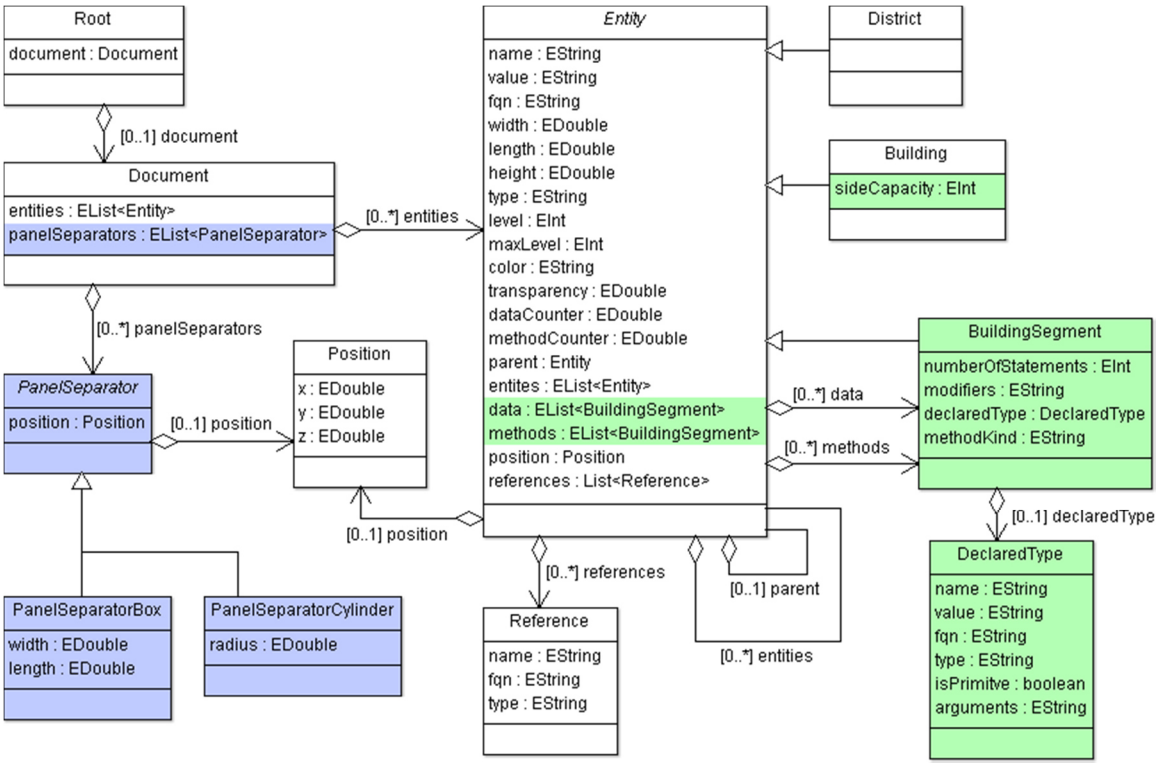

Abbildung 22: Klassendiagramm Stadtmetapher

[2] org.svis.xtext.city\src\org\svis\xtext\City.xtext

Die Stadtmetapher wurde bisher durch die zwei Elemente *Distrikt* (engl. *district)* und *Gebäude* (engl. *building)* aufgebaut, die von der Entitäten-Definition (engl. *entity)* abgeleitet wurden. Eine Hierarchie der Pakete und Klassen wurde außerdem dadurch erreicht, dass *Entity* weitere *Entity*-Instanzen enthalten konnte und somit einen rekursiven Aufbau der Struktur ermöglicht. Alle Entitäten, die weitere Entitäten beinhalten, sind Distrikte, während die (Blatt-)Knoten der Baumstruktur, bei denen dies nicht der Fall ist, als *Building* gelten. *Entity* erhält nun eine weitere abgeleitete Klasse namens *BuildingSegment*. Mithilfe dieses neu definierten Elementes werden alle Methoden und Attribute von Klassen dargestellt. Sowohl die Backstein-Variante als auch die Paneel-Variante nutzen *BuildingSegment*-Instanzen zur Repräsentation der Klassenbestandteile. Eine Differenzierung zwischen den Varianten ist nicht nötig, da sich die Elemente nur durch Form, Größe und Farbe unterscheiden, was bereits in der Definition für *BuildingSegment* vorgesehen ist. *Entity* verfügt bereits über Zählervariablen für Attribute und Methoden. Dazu kommen nun die Instanzen der gezählten Klassenbestandteile selbst, die über typisierte Listen verwaltet werden, was in der *Entity*-Klasse in Form der Attribute *data* und *methods* ersichtlich ist.

Beinahe alle benötigten Attribute wie Position und Abmessungen erhält *BuildingSegment* automatisch durch die Ableitung aus *Entity*. Da jedoch definierte Eigenschaften der Klassenbestandteile visualisiert werden sollen, wird die Definition um weitere Elemente ergänzt. Diese Erweiterungen sind für die Visualisierung nicht unbedingt erforderlich, ermöglichen jedoch eine Verwaltung der spezifischen Informationen direkt innerhalb der jeweiligen Instanzen, was eine Nachbearbeitung der Elemente, wie Sortierung anhand mehrerer Kriterien und Färbung der Elemente, erleichtert. Unter diesen Erweiterungen findet sich die Speicherung der Modifikatoren wie *public* oder *static*, die Anzahl an Anweisungen innerhalb der Methode (NOS) und in besonderem Maße die Definition der Struktur *DeclaredType*. Mit deren Hilfe wird der (Rückgabe-)Typ der Klassenbestandteile gespeichert. Dieser Typ wird als Zeichenkette (engl. *string*), abgelegt, jedoch werden auch zusätzliche Informationen daraus extrahiert und gespeichert. Beispielsweise definiert die boolesche Variable *isPrimitive*, ob dieser Typ ein primitiver Datentyp wie *int* oder *double* ist, oder ein zusammengesetzter Datentyp.

Speziell für die Backstein-Variante wird eine zusätzliche Variable innerhalb der *Building*-Klasse mit der Bezeichnung *sideCapacity* definiert. Diese Variable dient später der Berechnung von Länge und Breite der Grundfläche eines Gebäudes und der darauf platzierten Backsteine. Die Beschreibung des Algorithmus findet sich im anschließenden Abschnitt 3.4.1.

Ähnlich wie in der Backstein-Variante müssen auch in der Paneel-Variante die einzelnen Segmente differenzierbar sein. Um dies zu gewährleisten, sind die Segmente durch entsprechend gewählte Abstände getrennt. Eine zweite Möglichkeit, die Segmente optisch voneinander trennbar zu machen, besteht im Einfügen definierter Separator-Elemente. Dies hat den Vorteil, die Segmente auch aus ungünstigen Blickwinkeln besser unterscheiden zu können. In der Klasse *PanelSeparator* werden diese Trennelemente definiert und zwei verschiedene Ausführungen, *PanelsSeparatorBox* und *PanelSeparatorCylinder,* abgeleitet, da für die Paneele ebenso die zwei Formen der Box und des Zylinder vorgesehen sind. Die Separatorelemente werden nicht den *entities* in der *Document*-Klasse zugeordnet, da sie zwar Entitäten der Visualisierung sind, jedoch keine Informationen beinhalten und nur der besseren Übersicht dienen. Stattdessen wird *Document* eine getrennte Listenvariable hinzugefügt, in der die Elemente vorgehalten werden. Außerdem verhindert die Trennung spätere Probleme in den Transformationsschritten und sorgt dafür, dass Fallunterscheidungen nicht mehr notwendig sind.

3.4 Modell-Transformationen und Modifikationen

Nachdem das Metamodell um neue Definitionen erweitert wurde, müssen diese neuen Attribute bei der Verarbeitung der *FAMIX*-Dateien berücksichtigt werden. Die Klasse *Famix2City* wird deshalb modifiziert. Die Zuweisung der eingelesenen Methoden und Attribute aus den *FAMIX*-Dateien zu ihren jeweiligen Listen der Building-Instanzen und die Instanziierungen der *declaredType*-Attribute stellen allerdings die einzigen strukturellen Änderungen von *Famix2City* dar. Die folgenden Abschnitte beschreiben die Ergänzungen der Modell-Transformationen sowie Modifikationen wie Sortierungsvarianten und Zuweisungen von Farbschemata.

3.4.1 Modell-zu-Modell-Transformation der Backsteine

Die Generierung der Stadtmetapher mit ihren visuellen Elementen wird algorithmisch übernommen, was bedeutet, dass der *CityLayout*-Algorithmus von [Zilch 2015, 30ff] unverändert durchlaufen wird, bevor die Generierung der Backsteine startet. Im Gegensatz zur bestehenden Implementierung wird jedoch die vor Ausführung des Algorithmus gesetzte Gebäudegrundfläche anders berechnet. Da der bereits in Abschnitt 2.5.2 erwähnte Algorithmus zur Berechnung der Backsteinpositionen die Segmente in drei Dimensionen anordnet, muss die Gebäudegrundfläche auf Basis der von diesem Algorithmus erzeugten Grundfläche erfolgen. Dafür wird der Algorithmus in zwei Teile gegliedert und getrennt voneinander zu verschiedenen Zeitpunkten ausgeführt.

Algorithm 3.2 Progressive Bricks Layout for a collection of *elements*

```
1.  sc ← 0
2.  repeat {find the side capacity sc such that it satisfies: bc_min ≤ sc ≤ bc_max}
3.      sc ← sc + 1
4.      lc(sc + 1) ← sc * 4
5.      nol_min(sc + 1) ← sc * 2
6.      bc_min(sc + 1) ← lc(sc + 1) * nol_min(sc + 1)
7.      bc_max(sc) ← bc_min(sc + 1) − 1
8.  until bc_max ≥ elements.size
9.  for i = 0 to elements.size − 1 do
10.     if sc = 1 then
11.         lc ← 1
12.         biws ← 0
13.         si ← 0
14.     else
15.         lc ← (sc − 1) * 4
16.         biwl ← i mod lc
17.         biws ← biwl mod (sc − 1)
18.         si ← biwl ÷ (sc − 1)
19.     end if
20.     if si = 0 then {northern side}
21.         pi_x ← biws
22.         pi_y ← 0
23.     else if si = 1 then {eastern side}
24.         pi_x ← sc − 1
25.         pi_y ← biws
26.     else if si = 2 then {southern side}
27.         pi_x ← sc − biws − 1
28.         pi_y ← sc − 1
29.     else {western side}
30.         pi_x ← 0
31.         pi_y ← sc − biws − 1
32.     end if
33.     pi_z ← i ÷ lc
34.     move element_i to the position corresponding to pi
35. end for
```

Abbildung 23: *CodeCity*, Algorithmus Backsteine [Wettel 2010, 39]

Der erste Teil[3] dieses Algorithmus ist in Abbildung 23 schematisch dargestellt und grün hervorgehoben. Er berechnet für jedes Gebäude die Grundfläche und speichert das Ergebnis in der *sideCapacity*-Variablen der Gebäude-Instanz. Die *sideCapacity*-Variable gibt die Anzahl an Backsteinen wieder, die pro Seite auf dem Gebäude platziert werden können. Zusammen mit den Abständen zwischen den Backsteinen bildet dieser Wert die Grundfläche des Gebäudes und wird entsprechend zugewiesen. Die Berechnung des *sideCapacity*-Wertes ist nötig, da das Gebäude in Abhängigkeit der darzustellenden Backsteine sowohl in der Höhe als auch der Länge und Breite wächst. Eine Veranschaulichung dieses Prinzips findet sich in Abbildung 24.

Im oberen Teil des Bildes sieht man die Vogelperspektive auf die Backsteine, darunter die Seitenansicht. Es lässt sich erkennen, dass die Segmente bei steigender Anzahl auf eine größere Fläche verteilt werden. Links im Bild stapeln sich die Segmente übereinander, falls deren Anzahl zwischen 0 und 7 beträgt. Im rechten Teil des Bildes sind bei einer Segmentanzahl zwischen 72 und 127 die Segmente auf einer Grundfläche von 4x4 verteilt

[3] org.svis.generator\src\org\svis\generator\city\m2m\City2City_Bricks.xtend::calculateSideCapacity()

und zusätzlich nach oben gestapelt. Eine Besonderheit ist die Positionierung der Elemente als Außenwände der Gebäude, wobei innen liegende Positionen freigelassen werden und eine Art Innenhof bilden. Daraus ergibt sich der Vorteil, dass bei einer Drehung des Gebäudes alle Elemente von außen sichtbar sind und eine Überdeckung vermieden wird.

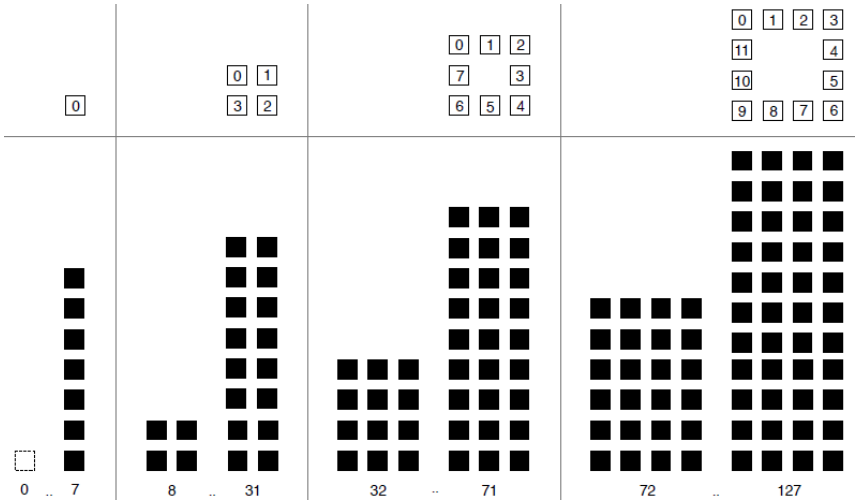

Abbildung 24: *CodeCity*, Layout Backsteine [Wettel 2010, 40]

Nach der Berechnung der Gebäudegrundfläche und der Generierung der Stadt durch den *CityLayout*-Algorithmus werden die Backsteine mithilfe des zweite Teils[4] des Backstein-Algorithmus erstellt. Dieser positioniert alle Elemente auf der Basis des hinterlegten *sideCapacity*-Wertes in Abhängigkeit von der Position[5] des Gebäudes. Die Arbeitsweise findet sich als blau gefärbter Pseudocode in Abbildung 23. Der Algorithmus füllt zuerst die unterste Ebene im Uhrzeigersinn mit Backsteinen auf und wechselt bei voller Belegung auf die nächst höhere Ebene. Dieses Verfahren wiederholt sich bis alle Elemente platziert wurden.

3.4.2 Modell-zu-Modell-Transformation der Paneele

Die Generierung der Stadtmetapher in der Paneel-Variante gestaltet sich ähnlich der Backstein-Variante. Analog wird zuerst die Gebäudegrundfläche berechnet und anschließend das *CityLayout* gesetzt, worauf die Platzierung der Gebäudesegmente folgt[6]. Die Berechnung der Gebäudegrundfläche erfolgt in dieser Variante jedoch nach einer einfacheren Methodik. Um möglichst viele Aspekte bei der Darstellung an Informationen abzudecken, spiegelt der Wert im Fall der Anzeige von Methoden die NOA wieder, was in Abbildung 25 demonstriert wird. Dies stellt eine identische Darstellung zur bestehenden Stadtmetapher dar. Im umgekehrten Fall der Repräsentierung von Attributen durch die

[4] org.svis.generator\src\org\svis\generator\city\m2m\BrickLayout.java::seperateBuilding()

[5] Die Position ist im position-Attribut der Building-Instanz abgelegt.

[6] org.svis.generator\src\org\svis\generator\city\m2m\City2City_Panels.xtend

Paneele, wird der Wert mit der NOM assoziiert. Die Erweiterung bietet allerdings auch die Möglichkeit der gleichzeitigen Darstellung von Methoden und Attributen. In dieser Konfiguration wird die Grundfläche ebenfalls die NOA visualisieren, um das Verhältnis der beiden Klassenbestandteile aus der Höhe und der Grundfläche des Gebäudes annähernd wiederzugeben. Die Gebäudegrundfläche entspricht gleichzeitig auch der Länge und Breite der darauf gestapelten Paneele, die als flache Quader dargestellt werden. Die Paneelhöhe bekommt ebenfalls eine Bedeutung zugewiesen. Sie ist Ausdruck der NOS innerhalb der Methode. Dabei wird die Höhe in Intervallen an diese Metrik gekoppelt. Vorgesehen sind zehn Abstufungen, die sich in Intervallen wie 1-3 NOS oder 240-∞ NOS staffeln. In Abbildung 25 kann man die hellblauen Paneele in unterschiedlicher Höhe über den dunkelblau skizzierten Gebäudesockeln erkennen. Paneele mit höheren NOS-Metrikwerten sind stets unter den Paneelen mit geringeren Werten angeordnet und zeigen somit eine Sortierung der Elemente nach dieser Metrik auf.

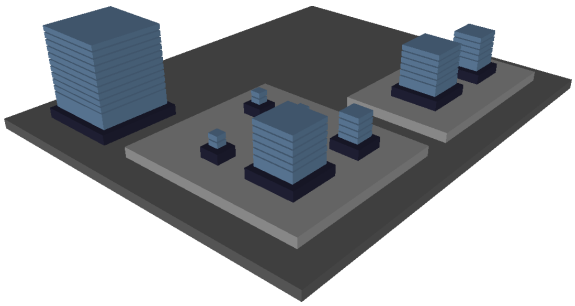

Abbildung 25: *FAMIX*-Datei *Bank*, Paneele

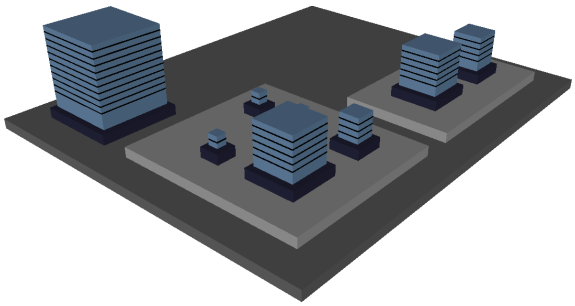

Abbildung 26: *FAMIX*-Datei *Bank*, Paneele und Separatorelemente

In Abbildung 25 ist zu erkennen, dass eine Differenzierung der einzelnen Paneele trotz Abständen voneinander, besonders bei gleicher Färbung, schwierig erscheint. Um diese Problematik abzuschwächen, wurde ein optisches Hilfselement definiert, das die Leerräume zwischen den Paneelen ausfüllt und durch die intensive Schwarzfärbung den Kontrast erhöht. Abbildung 26 zeigt den Einsatz dieser Separatoren und wie sie die Unterscheidbarkeit der Paneele im Vergleich zu Abbildung 25 verbessern.

3.4.3 Modellmodifikationen

Die folgenden Erläuterungen gelten sowohl für die Backstein- als auch für die Paneel-Variante und stellen die Nutzung von unterschiedlichen Farben sowie Sortiermöglichkeiten für die Gebäudesegmente vor.

Um der Stadtmetapher ein bestimmtes Maß an Variabilität in ihrer Konfiguration zu bieten, werden einige Aspekte bei der Generierung der Visualisierung mithilfe von

Variablen gesteuert, die in einer speziellen Datei[7] einstellbar sind. Über diese Konfigurationsdatei *CitySettings* können, neben Größeneinstellungen aller Elemente und Abstände zueinander, die anzuzeigenden Klassenbestandteile bestimmt werden. Die Wahlmöglichkeiten sind dabei auf *Methoden*, *Attribute* und *Methoden_und_Attribute* beschränkt. Des Weiteren können die blauen Gebäudesockel, wie in Abbildung 25 unterhalb der Paneele zu sehen, wahlweise ausgeblendet werden, was keine Informationsreduzierung bedeutet, sondern lediglich dem Nutzer eine Wahl der Darstellungsform, je nach persönlicher Präferenz, gestattet.

Auch eine Änderung des in Abschnitt 3.4.1 beschrieben Algorithmus zur Berechnung der *sideCapacity* ist vorgesehen. Eine Optionsmöglichkeit dabei ist, die Backsteine konsequent aufeinander zu stapeln, ohne die Gebäude auch in die Länge und Breite wachsen zu lassen. Das hat zur Folge, dass alle Gebäude eine 1x1-Grundfläche besitzen und sich nur in ihrer Höhe unterscheiden, die sich wiederum aus der Anzahl an übereinander gestapelten Backsteinen ergibt. Daneben gibt es die Optionen *Progressiv* und *Balanciert*. Das *Progressiv*-Layout entspricht den *progressive layout* von [Wettel 2010, 38ff] und berechnet sich nach der Anzahl der darzustellenden Klassenbestandteile. Das bedeutet im Fall der Visualisierung von Methoden, dass die Methodenanzahl als Basis der Berechnung dient, was analog auch für die Attribute gilt. Das *Balanciert*-Layout wandelt dieses Konzept ab, und nutzt zur Berechnung der Fläche die jeweils andere Entitätenanzahl. Für die Anzeige von Methoden wird somit die Attributanzahl herangezogen und umgekehrt. Die Intention dahinter ist die Darstellung des Verhältnisses der Klassenbestandteile zueinander, das durch die Gebäudehöhe näherungsweise vergleichbar wird. Dieses Prinzip ähnelt dem Ansatz der Stadtmetapher, bei der lediglich die Klassen als Gebäude dargestellt werden und die Grundfläche die NOA repräsentiert, während die Höhe die NOM darstellt. Das *Progressiv*-Layout und das *Balanciert*-Layout sind identisch, falls beide Klassenbestandteile visualisiert werden, da sich die Fläche dabei aus der Summe der Methoden- und der Attributanzahl ergibt.

Für die Paneel-Variante existiert ebenfalls die Wahlmöglichkeit in Bezug auf die Anzeige der Abstände zwischen den Paneelen. Wie bereits in den Abbildungen 25 und 26 gezeigt, können die Paneele wahlweise durch spezielle Separatoren oder durch Leerräume voneinander getrennt werden. Ist eine Trennung nicht gewünscht, kann auch dies eingestellt werden, wodurch die Paneele nahtlos aufeinander gestapelt werden. Die Paneele selbst sind nicht auf die Quaderform beschränkt. Wahlweise können die Attribute zur besseren

[7] org.svis.generator\src\org\svis\generator\city\CitySettings.java

Unterscheidung auch als flache Zylinder gestapelt werden, was in Abbildung 27 veranschaulicht ist.

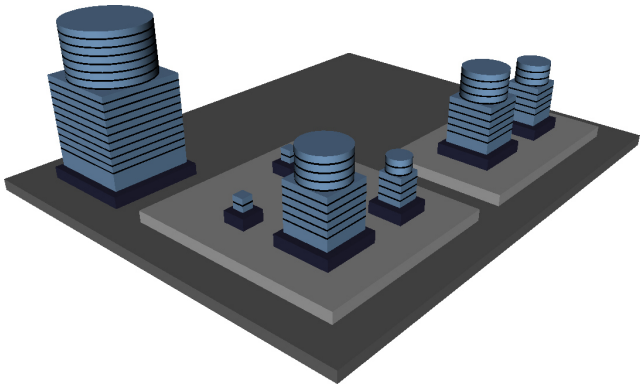

Abbildung 27: *FAMIX*-Datei *Bank*, Paneele, Attribute als Zylinder

Wie in Abschnitt 3.3 bereits angedeutet, werden für die Klassenbestandteile Informationen wie Name, Modifikatoren und Datentyp aus den *FAMIX*-Dateien extrahiert und zur weiteren Verwendung im Speicher gehalten. Die beiden implementierten Visualisierungsvarianten sollen neben dem schlichten Anzeigen der Entitäten auch bestimmte Eigenschaften der Klassenbestandteile wiederspiegeln und zur besseren Übersicht nach definierten Kriterien in der Visualisierung sortiert werden. Da die Stadtmetapher bisher noch wenig Gebrauch von dem zur Verfügung stehenden Farbraum gemacht hat, wird dieser nun für die Gebäudesegmente genutzt, um Modifikatoren und Datentypen beziehungsweise die Art der Methoden zu veranschaulichen, was in Abschnitt 3.4.3.1 beschrieben wird. Des Weiteren sollen die Gebäudesegmente nach den benannten Eigenschaften sortiert werden können. Die Erläuterung der Umsetzung folgt in Abschnitt 3.4.3.2.

3.4.3.1 Farbschemata

Die Gebäudefarben sollen nun für die Darstellung von Modifikatoren und Datentypen genutzt werden. Da allerdings eine gleichzeitige Darstellung verschiedener Modifikatoren und Datentypen zu viele nicht mehr differenzierbare Farben nötig machen würde, müssen Einschränkungen hinsichtlich des Detaillierungsgrades der genannten Eigenschaften getroffen werden. Zunächst werden die Eigenschaften in zwei Kategorien getrennt und nicht mehr gleichzeitig visualisiert. Dafür wird in der Konfigurationsdatei eine weitere Option definiert, die es ermöglicht, zwischen der Darstellung von Modifikatoren oder den Datentypen und damit der Farbschemata zu wählen.

Bei der Darstellung der Modifikatoren werden die vier verschiedenen Zugriffsmodifikatoren *private*, *package*, *protected* und *public* farblich hervorgehoben. Eine Differenzierung zwischen Methoden und Attributen findet hierbei nicht statt. In der vorliegenden Konfiguration der Stadtmetapher werden als *private* definierte Entitäten rot

markiert, *package* als blau, *protected* gelb und *public* grün gefärbt, was in Abbildung 28 anhand der Backstein-Variante dargestellt ist.

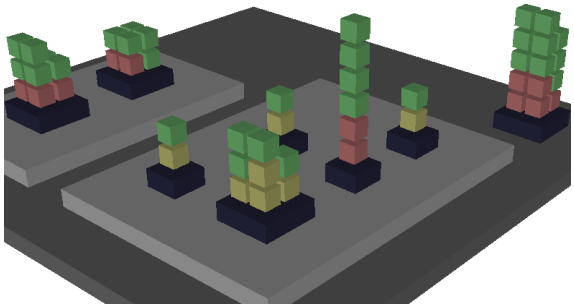

Abbildung 28: *FAMIX*-Datei *Bank*, Backsteine, Zugriffsmodifikatoren[8]

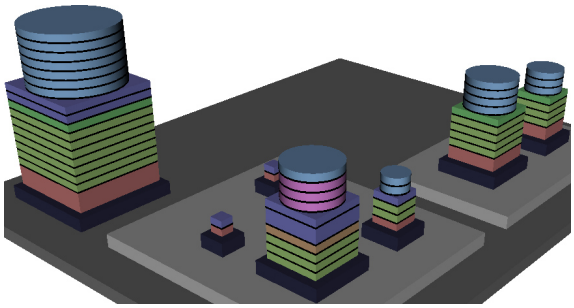

Abbildung 29: *FAMIX*-Datei *Bank*, Paneele, Datentypen[9]

Für die Visualisierung der Datentypen beziehungsweise Methodentypen werden den Methoden und Attributen unterschiedliche Farben zugewiesen, da sie unterschiedliche Eigenschaften widerspiegeln sollen. Die Datentypen der Attribute werden in die zwei Kategorien des einfachen und des zusammengesetzten Datentyps unterschieden. Dafür wird der Wert der zuvor in Abschnitt 3.3 erwähnten booleschen Variable *isPrimitive* der *BuildingSegment*-Instanz genutzt, der bei der Analyse der *FAMIX*-Dateien generiert wurde. Im Fall des einfachen Datentyps wird das Segment rosa gefärbt und hellblau, wenn es ein zusammengesetzter Datentyp ist. Die farbliche Gestaltung der Methoden entscheidet sich auf Basis anderer Kriterien, da den Rückgabewerten von Methoden bei einer Analyse von Softwaresystemen weniger Beachtung geschenkt wird, was den beschriebenen Metaphern in Kapitel 2 entnommen werden kann. Stattdessen soll die Funktionsweise der Methoden besser in den Fokus gerückt werden. Die Methoden werden in sechs Kategorien unterteilt, wodurch ihre Arbeitsweise besser veranschaulichet werden soll. Differenziert werden Konstruktoren, Getter- und Setter-Methoden, statische und abstrakte Methoden, und schließlich Methoden, die keiner dieser Kategorien zugeordnet werden können. In Abbildung 29 kann man am Beispiel der ganz links zu findenden Klasse erkennen, dass neben einem rot markierten Konstruktor am Sockel des Gebäudes sich mehrere olivgrün gefärbte Getter-Methoden anschließen. Darauf befindet sich eine hellgrüne Setter-Methode, gefolgt von zwei unkategorisierten Methoden in Dunkelblau. Gelbgefärbte statische Methoden kommen ebenso wenig vor wie orangenfarbene abstrakte Methoden. Eine abstrakte Methode ist allerdings bei der Klasse zu finden, welche in der Mitte, nahe des unteren Bildrandes in dieser Abbildung, platziert ist. Aufgrund der Charakteristika der

[8] X3D-Beispiele\Beispiel_1.x3d auf der CD

[9] X3D-Beispiele\Beispiel_2.x3d auf der CD

genannten Methoden, lässt sich die Klasse wahrscheinlich als Datenklasse ohne besondere Funktionalitäten einordnen.

3.4.3.2 Sortierung

Zur Verbesserung der Identifizierbarkeit definierter Eigenschaften der Entitäten werden diese in der Stadtmetapher nach einem in der Konfigurationsdatei einstellbaren Muster angeordnet. Die Sortierung erfolgt nach bis zu drei Kriterien mit abnehmender Priorität. Zur Veranschaulichung kann man beispielsweise die Sortierung von Aktenschränken betrachten, wobei auf erster Sortierebene nach Thematik der Aktenordner, und auf zweiter Ebene nach Datum der einzelnen Unterlagen und schließlich auf dritter Ebene nach Seitennummer innerhalb der Akten unterschieden wird.

Die Sortierfunktion wurde als Wrapper[10] für die *BuildingSegment*-Klasse implementiert, die diese Funktion kapselt. Eine direkte Implementierung des *Comparator*-Interfaces innerhalb der *BuildingSegment*-Klasse ist aufgrund der in *Xtext* verfassten Definition und der automatisierten Generierung des korrespondierenden *Java*-Quelltextes der Klasse nur schwer möglich. Dies macht es nötig, den Generator um diesen speziellen Wrapper zu ergänzen. Bei einer Sortierung werden alle Segmente jeweils einer Wrapper-Instanz zugewiesen, diese Wrapperliste wird dann sortiert, und anschließend werden die Segmente in der sortierten Reihenfolge in die originale Liste zurückgeschrieben. Der Speicherverbrauch während der Verarbeitung erhöht sich bei dieser Methodik nur unwesentlich, da pro Wrapper-Instanz lediglich die Referenz auf das Segment und jeweils drei *Integer*-Variablen benötigt werden. Ein signifikanter Anstieg an Rechenzeit ist ebenfalls nicht zu verzeichnen, da nur der Arbeitsspeicher in vergleichsweise geringe Höhe allokiert werden muss und die Zuweisung der Objektreferenzen im Verhältnis zur kompletten Transformationsfolge von der *FAMIX*-Datei in eine *X3D*-Datei vernachlässigt werden kann.

Die höchste Priorität bei der Sortierung der Elemente hat die Unterscheidung zwischen Methoden und Attributen. Eine Auswahl, in welcher Reihenfolge die Klassenbestandteile aufgebaut werden sollen, kann durch die Wahl eines der Modi *Methoden_zuerst*, *Attribute_zuerst* oder *Unsortiert* erfolgen. In letzterem Fall wird der Sortiercharakter von der nächst feineren Sortierstufe dominiert. Hierbei kann wiederum zwischen den Sortiermodi *Alphabetisch*, *Schema*, *NOS* oder *Unsortiert* gewählt werden. Bei dem *Alphabetisch*-Sortiermodus sind die Namen der Attribute und Methoden als Sortierkriterium

[10] org.svis.generator\src\org\svis\generator\city\m2m\BuildingSegmentComparator.java

definiert. Der Modus *NOS* ordnet die Elemente anhand der in ihnen definierten Anzahl an Anweisungen, was keinerlei Einfluss auf die Sortierung der Attribute hat.

```
public static enum SortPriorities_Types {;
    public static final int CONSTRUCTOR = 1;
    public static final int GETTER      = 2;
    public static final int SETTER      = 3;
    public static final int STATIC      = 4;
    public static final int ABSTRACT    = 5;
    public static final int LEFTOVER    = 6;
}
```

Listing 1: Prioritätenliste der Methodentypen

```
public static enum SortPriorities_Visibility {;
    public static final int PRIVATE = 1;
    public static final int PROTECTED = 2;
    public static final int PACKAGE = 3;
    public static final int PUBLIC = 4;
}
```

Listing 2: Prioritätenliste der Zugriffsmodifikatoren

Schema sortiert die Elemente anhand des gewählten Farbschemas aus Abschnitt 3.4.3.1. Da hierbei allerdings bis zu sechs Differenzierungen existieren, wurden Prioritätslisten[11] für diese Sortiermodi definiert, in denen die Elemente anhand von *Integer*-Werten die Sortierreihenfolge vorgeben. Die Prioritätenliste der Methodentypen ist in Listing 1 veranschaulicht und ist Bestandteil der Visualisierung in Abbildung 29. Für das Farbschema beziehungsweise Visualisierungsschema der Zugriffsmodifikatoren gilt eine Prioritätenliste analog zu Listing 2. Je kleiner der Wert der jeweiligen Konstanten gewählt wird, desto höher ist die Priorität des Typs, was bedeutet, dass die assoziierten Entitäten in ihrer Abarbeitungsreihenfolge zuerst platziert werden. Durch den Aufbau der Segmente von unten nach oben, ergibt sich daraus, dass sich die Segmente näher am Sockel des Gebäudes befinden. Mit der Regelung der Sortierreihenfolge über diese Enumeratoren ist es möglich, mit wenig Zeitaufwand die Sortierung anzupassen oder gar mehrere verschieden kategorisierte Elemente durch Wahl gleicher Werte zusammenzufassen.

Es existiert noch eine dritte Sortierebene, welche jedoch automatisch ausgeführt wird und nicht über die Konfigurationsdatei beeinflussbar ist. Die Sortierung in dieser letzten und feinsten Ebene wird nur genutzt, falls in der zweiten Ebene entweder die Modi *Schema* oder *NOS* ausgewählt wurden. Der darauf folgende Sortieralgorithmus arbeitet jeweils nach dem anderen Sortiermodus der zweiten Ebene. Konkret wird in der dritten Sortierebene immer nach *NOS* sortiert, falls in der zweite Ebene nach *Schema* sortiert wurde, und umgekehrt.

3.4.4 Modell-zu Text-Transformation

Die Überführung[12] der generierten Entitäten mit all ihren Eigenschaften in eine *X3D*-Datei ist der letzte Transformationsschritt zur Erstellung der Visualisierung der Stadtmetapher. Die Transformation erfolgt durch die Abarbeitung der *entities*-Liste innerhalb der *Document*-Klasse des in Abschnitt 3.3 vorgestellten Metamodells, in welcher alle Entitäten

[11] Enumeratoren *SortPriorities_Visibility* und *SortPriorities_Types* in der Datei CitySettings.java.

[12] org.svis.generator\src\org\svis\generator\city\m2t\City2X3D.xtend

der Stadtmetapher vorgehalten werden. Abhängig vom Entitätstyp werden dabei für jeden Typ eigens gekapselte Methoden *toDistrict* und *toBuilding* ausgeführt, die eine mittels Parameter übergebene Entität in eine *X3D*-Defintion transformieren. Mit der Definition der neuen Entität *BuildingSegment* wird dieser Abarbeitung eine zusätzliche Methode *toBuildingSegment* hinzugefügt, welche nun auch die generierten Gebäudesegmente der *X3D*-Datei hinzufügt. Die Definition der Methode ist in Listing 3 veranschaulicht.

```
112  def String toBuildingSegment(BuildingSegment entity) '''
113      «var x = entity.position.x»
114      «var y = entity.position.y»
115      «var z = entity.position.z»
116      «var width = entity.width»
117      «var height = entity.height»
118      «var length = entity.length»
119      <Group DEF='«entity.fqn»'>
120          <Transform translation='«x +" "+ y +" "+ z»'>
121              <Shape>
122              «IF CitySettings.X3D_Mode == "Panels"
123                      && entity.type.equals("FAMIX.Attribute")
124                      && CitySettings.SET_SHOW_ATTRIBUTES_AS_CYLINDERS»
125                  <Cylinder radius='«width/2»' height='«height»'></Cylinder>
126              «ELSE»
127                  <Box size='«width +" "+ height +" "+ length»'></Box>
128              «ENDIF»
129                  <Appearance>
130                      <Material diffuseColor='«entity.color»'></Material>
131                  </Appearance>
132              </Shape>
133          </Transform>
134      </Group>
135  '''
```

Listing 3: *X3D*-Konvertierung von Gebäudesegmenten

Die Funktionsweise der Methode orientiert sich stark an den bereits vorhandenen Methoden *toDistrict* und *toBuilding*. Eine Reihe von Variablendeklarationen in den Zeilen 113-118 dient lediglich der besseren Lesbarkeit der eigentlichen Transformation ab Zeile 119. Der wesentliche Unterschied zu den übrigen Methoden ist zwischen den Zeilen 122 und 127 erkennbar, wobei eine Fallunterscheidung getroffen wird, in welcher Form die Paneele dargestellt werden. Ein erster Test auf die Visualisierungsvariante ist in Zeile 122 definiert, welche im Falle der Backstein-Variante (Zeile 126) die Segmente als Box darstellt und die Methode *toBuildingSegment* damit identisch zu den übrigen Methoden *toDistrict* und *toBuilding* ausführt. Soll jedoch die Paneel-Variante visualisiert werden (Zeile 122), werden weitere Bedingungen getestet. Falls das Segment ein Attribut der Klasse repräsentiert (Zeile 123) und die Konfiguration eine Darstellung der Attribute als Zylinder vorsieht (Zeile 124), dann wird das Attribut als Zylinder visualisiert. Gilt eine dieser drei Bedingungen nicht, erhält das Segment die in Zeile 127 definierte Boxform.

Die Paneelseparatoren werden in einer zusätzlichen Methode *toPanelSeparator* nach dem selben Schema wie *toBuildingSegment* behandelt, wobei jedoch der Typ der definierten *PanelSeparator*-Instanz geprüft wird, der bereits zur Modell-zu-Modell-Transformation

erzeugt wurde. Anhand dieses Typs wird die entsprechende *X3D*-Form *Box* oder *Cylinder* zur Darstellung der Paneelseparatoren gewählt.

3.5 Evaluation der Varianten

Nach der vorangehenden Beschreibung der Modell-Änderungen der Stadtmetapher des Generators sollen die beiden erstellten Varianten zur Darstellung anhand der *FAMIX*-Datei *Freemind* evaluiert werden. Abbildung 30 zeigt die Paneel-Variante mit den Methoden als Boxen und den Attributen als Zylinder. Aus gezeigter Perspektive werden die Komplexitäten der Klassen, bezogen auf die Zahl ihrer Attribute und Methoden, teilweise sichtbar, und auch ihr Zahlenverhältnis zueinander wird erkennbar. Breite, hohe Gebäude besitzen in der Summe viele Methoden und Attribute, während schmale, hohe Gebäude lediglich viele Methoden haben. Im unteren rechten Bereich des Bildes ist eine Klasse mit vielen Attributen und verhältnismäßig wenigen Attributen erkennbar. Ebenfalls scheint ein großer Teil der Methoden nicht in die vordefinierten Methodentypen kategorisierbar zu sein, was der große Anteil an dunkelblauen Paneelen wiederspiegelt.

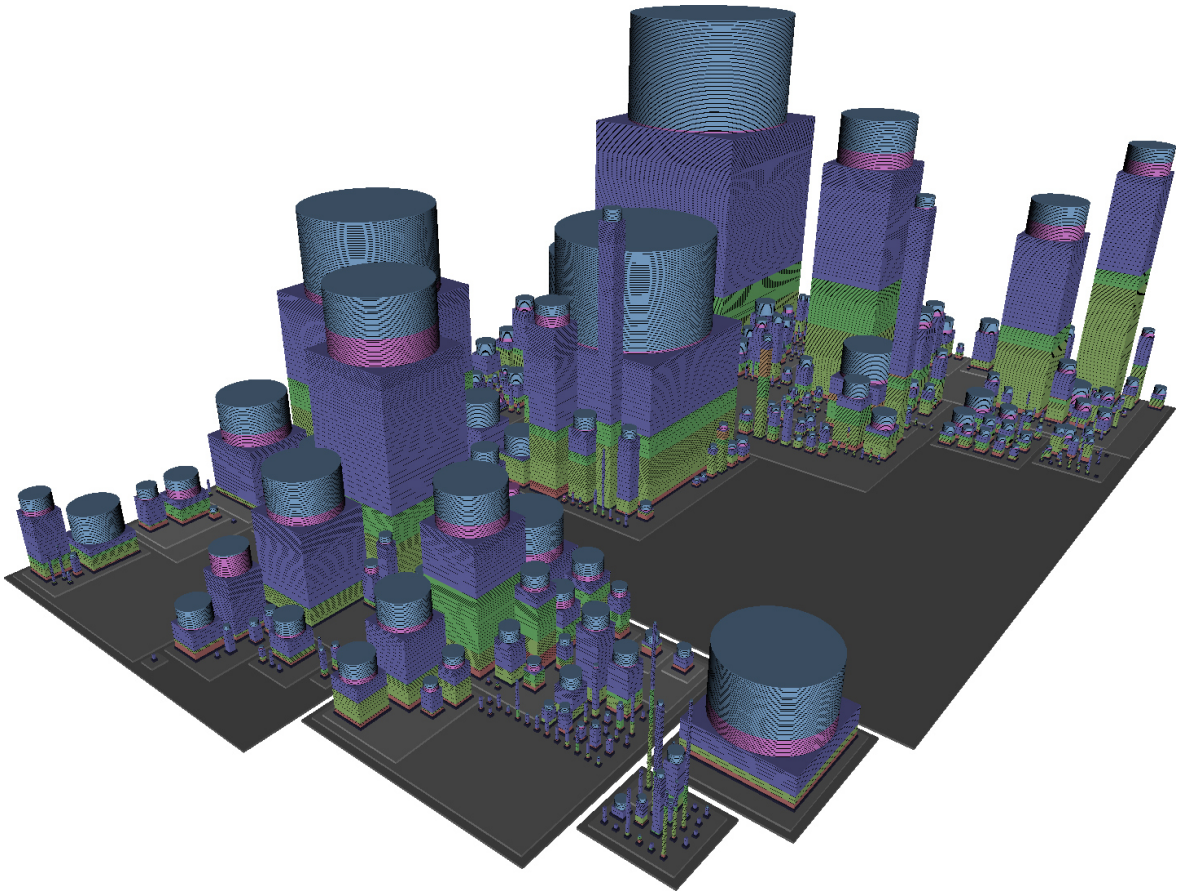

Abbildung 30: *FAMIX*-Datei *Freemind*, Paneele, Typen hervorgehoben[13]

[13] X3D-Beispiele\Beispiel_3.x3d auf der CD

Leider erzeugen die einzelnen Gebäudesegmente durch die Abstände sehr unästhetische Moiré-Effekte, die auch mit einer anderen Farbe der Separatorelemente oder einfaches Weglassen dieser nicht beseitigt werden können. Einzig das Zusammenfügen aller Segmente ohne Zwischenraum lässt diesen optischen Effekt verschwinden, was allerdings ein Differenzieren der Elemente unmöglich macht. Lediglich die Wahl einer wesentlich höheren Auflösung oder anderer Zoomstufen vermag den Effekt zu mildern.

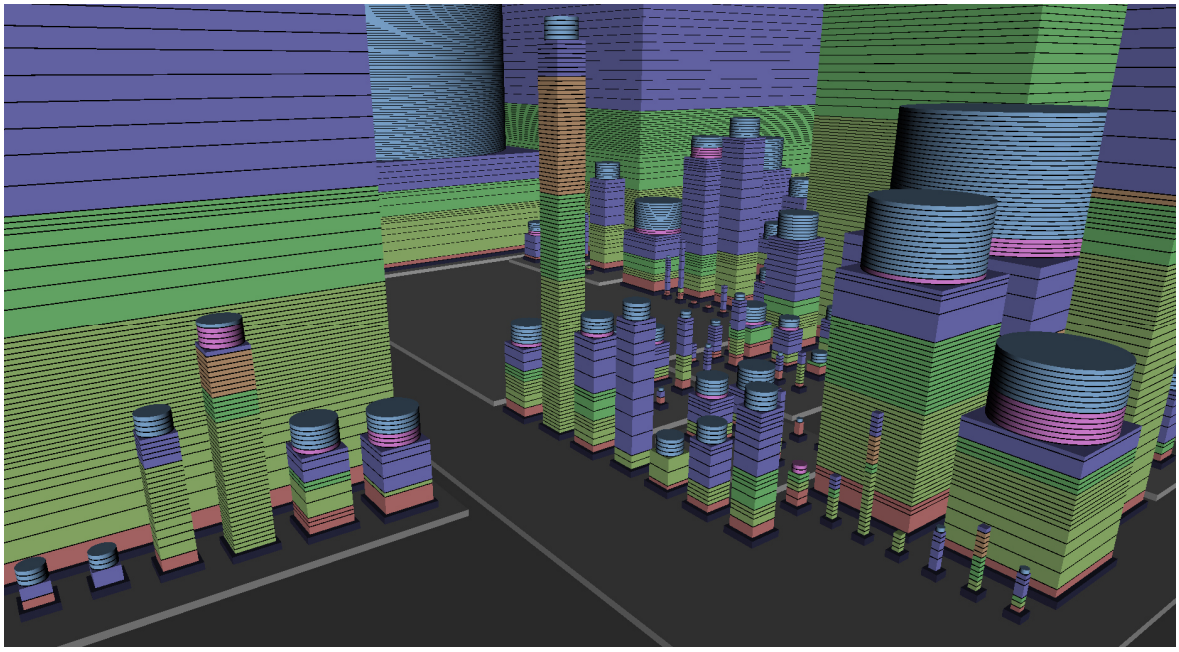

Abbildung 31: *FAMIX*-Datei *Freemind*, Paneele, Typen hervorgehoben, Zoomansicht

Abbildung 31 zeigt den vergrößerten Bildausschnitt von Abbildung 30, der sich leicht rechts vom Bildmittelpunkt befindet. Darin sind die in dieser Stadtregion platzierten Gebäude besser erkennbar und einzelne Methodentypen lassen sich anhand der unterschiedlichen Farben identifizieren. Das schmale hohe Gebäude in der Mitte der Abbildung hat beispielsweise eine nicht unerhebliche Anzahl an abstrakten Methoden, welche orange hervorgehoben sind. Außerdem beinhaltet diese Klasse eine noch größere Anzahl an Getter- und Setter-Methoden. Diese Eigenschaften heben das Gebäude von den anderen in ihrer Umgebung ab. Anhand des großen Gebäudes im linken Bereich des Bildes kann man die konfigurierte feinere Sortierreihenfolge *Types-NOS* erkennen. Im Groben werden hierbei die Methoden unter den Attributen angeordnet.

Abbildung 32 zeigt den gleichen Stadtteil wie in Abbildung 31, jedoch in einer anderen Konfiguration der Metapher. Zunächst werden lediglich die Methoden und keine Attribute mehr angezeigt und der dunkelblaue Gebäudesockel ist ausgeblendet. Ebenfalls werden statt den Typen der Methoden beziehungsweise den Datentypen der Attribute die Zugriffsmodifikatoren farblich hervorgehoben. Die Sortierreihenfolge entspricht hierbei jedoch nicht dem Farbschema, sondern es wird in erster Instanz nach NOS sortiert und erst

innerhalb gleicher NOS-Werte nach dem Farbschema, was besonders an dem großen Gebäude links im Bild deutlich hervortritt.

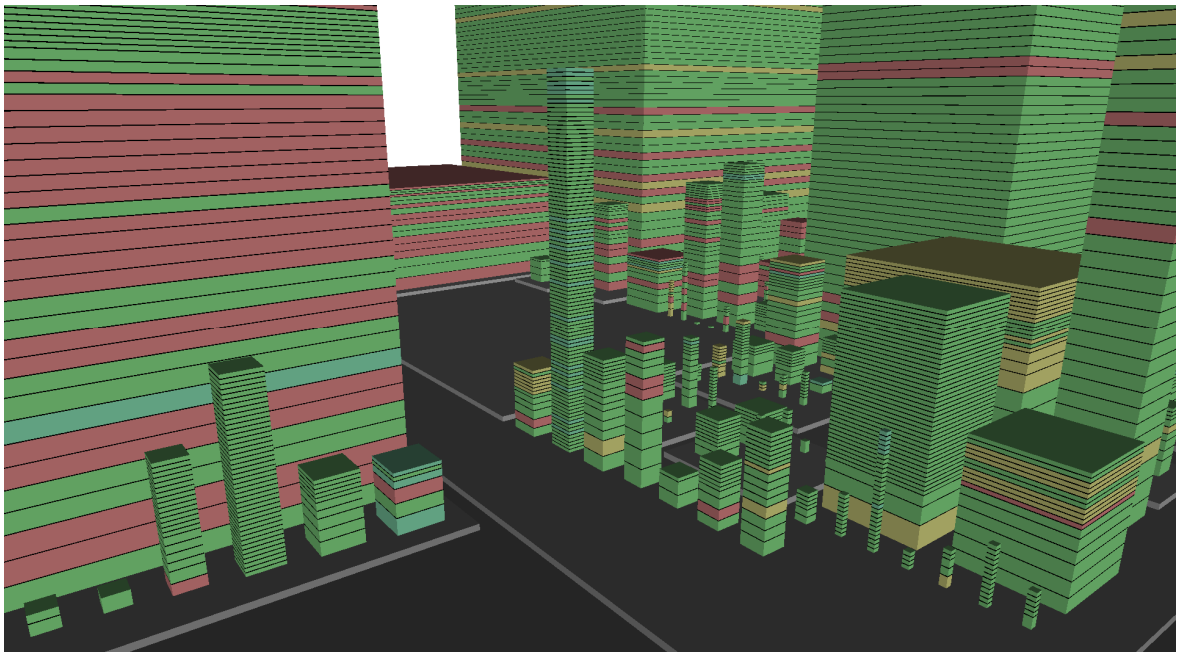

Abbildung 32: *FAMIX*-Datei *Freemind*, Paneele, Modifikatoren hervorgehoben, Zoomansicht[14]

Diese genannten Unterschiede zwischen Abbildung 31 und 32 sollen die Variabilität in der Konfiguration der Stadtmetapher verdeutlichen und die Möglichkeiten der unterschiedlichen Betonung bestimmter Aspekte demonstrieren.

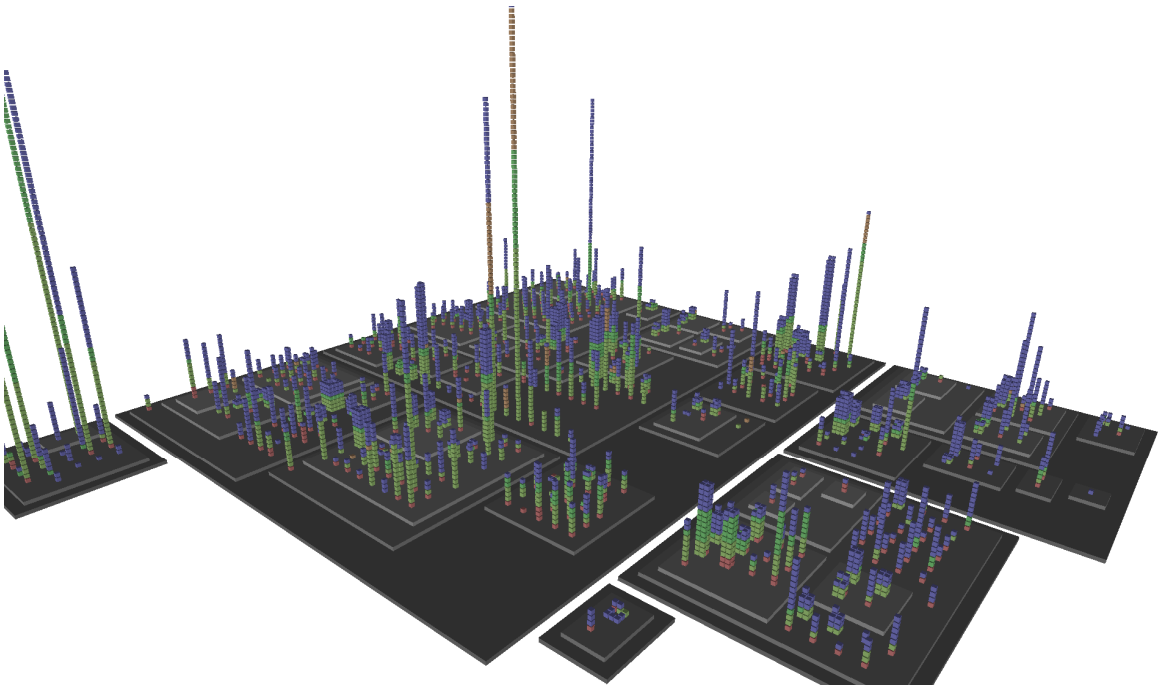

Abbildung 33: *FAMIX*-Datei *Freemind*, Backsteine, Modifikatoren hervorgehoben[15]

[14] X3D-Beispiele\Beispiel_4.x3d auf der CD

[15] X3D-Beispiele\Beispiel_5.x3d auf der CD

Abbildungen 33 und 34 sind Backstein-Varianten mit identischer Konfiguration der Metapher, jedoch in verschiedenen Zoomstufen des Bildausschnitts der Visualisierung. In dieser Backstein-Variante wurden Methoden visualisiert, die farblich ihren jeweiligen Methodentyp darstellen und ebenfalls nach dieser Eigenschaft sortiert sind.

Die automatische drittrangige Sortierung nach der NOS wird in dieser Variable konzeptbedingt nicht deutlich, da dies visuell nicht durch verschiedene Größen dargestellt wird. Als Modus zur Anordnung der Elemente wurde das *Balanciert*-Layout gewählt, was bedeutet, dass zur Grundflächenberechnung nicht die Methodenanzahl, sondern die Attributanzahl der Klasse herangezogen wurde. Dies wird ersichtlich aus der Tatsache, dass hohe Gebäude existieren, die nicht gleichzeitig auch in ihrer Länge und Breite gewachsen sind.

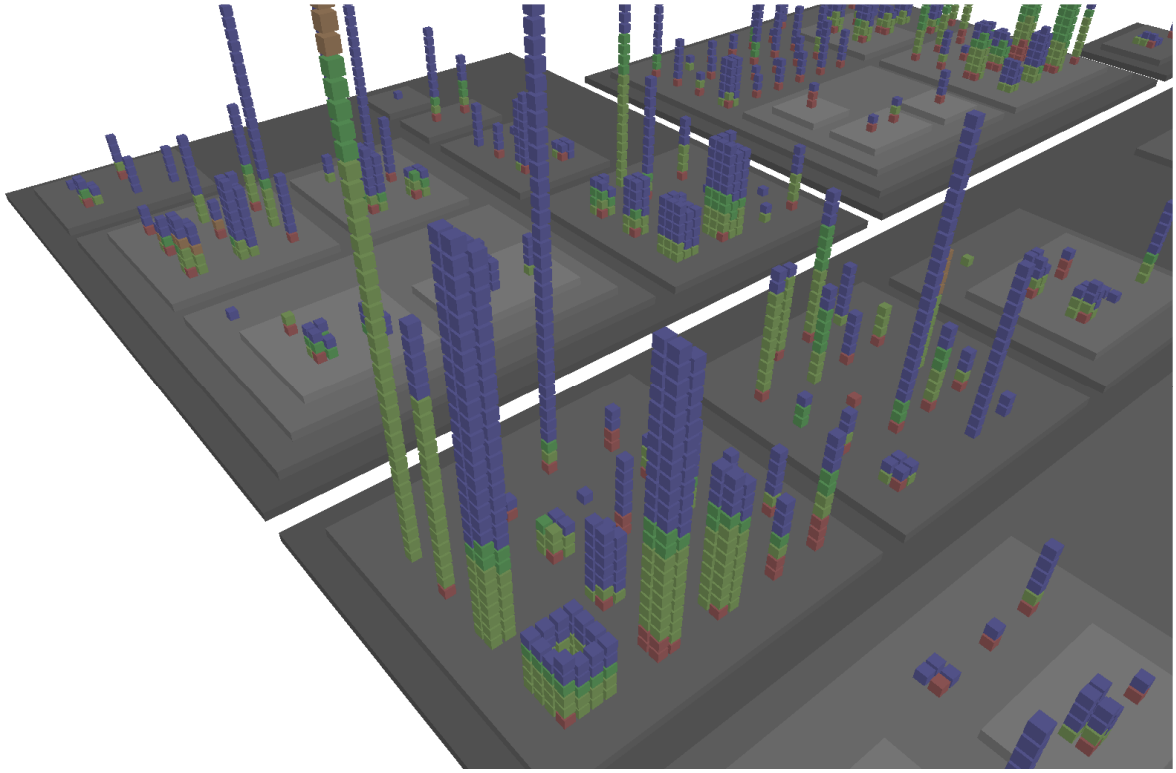

Abbildung 34: *FAMIX*-Datei *Freemind*, Backsteine, Modifikatoren hervorgehoben, Zoomansicht

In Abbildung 34 wird dieser Sachverhalt besonders deutlich. Zu sehen sind fast ausnahmslos Gebäude mit einer 1x1- oder 2x2-Grundfläche. Am unteren Rand der Abbildung ist in der Mitte allerdings ein Gebäude mit einer 4x4-Grundfläche zu sehen, bei welchem maximal sechs Segmente in der Höhe gestapelt sind. Dieses Verhältnis zwischen Grundfläche und Höhe zeigt eine relative Ausgeglichenheit zwischen der Attribut- und Methodenanzahl der Klasse. Außerdem ist aus dieser Perspektive der Gebäudeaufbau zu erkennen, der die Backsteine ausschließlich am Rand des Gebäudes stapelt, aber einen Leerraum in der Mitte des Gebäudes lässt. Das Ausblenden der Gebäudesockel demonstriert in dieser Abbildung den Effekt, dass Klassen ohne Methoden, welche allerdings Attribute besitzen, nicht

angezeigt werden. Damit ergeben sich stellenweise Leerräume zwischen den einzelnen Gebäuden, was im rechten Bereich des Bildes auf mittlerer Höhe zu erkennen ist.

4 Zusammenfassung und Ausblick

Das Ergebnis dieser Arbeit ist die Erweiterung der Stadtmetapher um die Darstellung von Methoden und Attributen der Klassen. Damit wurde der Stadtmetapher eine weitere Darstellungsebene hinzugefügt und die Analysemöglichkeiten von Softwaresystemen auf die Klassenbestandteile wurden erweitert. Die Implementierung von zwei verschiedenen Varianten sollte einerseits den Ansatz der *Bricks* von [Wettel 2010, 38ff] weiterverfolgen, jedoch zusätzlich auch einen neuen Ansatz für die Repräsentierung der Klassenbestandteile bieten. Eine Analyse bestehender Stadtmetaphern bot dabei die Grundlage und Inspiration für den Ansatz der Paneel-Variante. Mithilfe der Farbgestaltung sollte der Detaillierungsgrad der Metapher durch die Hervorhebung bestimmter Eigenschaften erweitert werden. Die Sortierung bezüglich der Anordnung von Elementen anhand definierter Eigenschaften gewährleistet die Übersichtlichkeit in der Metapher und komplettiert die Erweiterung.

Die Erweiterung der Stadtmetapher erwies sich aufgrund der Nutzung des generativen Paradigmas bei der Entwicklung des Softwarevisualisierungsgenerators sowie der Unterstützung durch Sprachen wie *Xtext* zur Generierung von Codefragmenten als effizient und brachte eine deutliche Zeitersparnis bei dem Implementierungsprozess mit sich. Dennoch hatte die Generierung von Codefragmenten durch *Xtext* auch Nachteile, wie die Erschwerung der nachträglichen Erweiterbarkeit der Klassen um einen Komparator. Für diesen musste eine spezielle Wrapper-Klasse geschrieben werden, die den Sortieralgorithmus kapselt, was im Falle der Nutzung des Comparator-Interfaces direkt in der betreffenden *BuildingSegment*-Klasse vermieden werden könnte.

Die Backstein- wie auch die Paneel-Variante mit ihren farblichen Ausgestaltungen legen noch nicht ausgeschöpfte Visualisierungspotentiale der Stadtmetapher offen, welche sich durch Einbeziehung zusätzlicher Informationen des Softwaresystems erweitern lassen. Denkbar wäre zusätzlich auch die Variation weitere Entitäteneigenschaften, wie beispielweise die Drehung der Segmente, um Metriken auszudrücken. Auch eine spezielle Belichtung oder der Einsatz von Transparenz sind denkbar. Des Weiteren wurden innere Klassen bisher von der Visualisierung ausgeblendet, was durch eine weitere Schachtelung der Elemente oder die Einführung einer neuen Darstellungsebene, ähnlich wie die Innenansicht der Gebäude in *Software World*, umsetzbar wäre. Für lokale Variablen und Parameter von Methoden wäre diese Erweiterung gleichermaßen möglich. Außerdem ist die Stadtmetapher lediglich auf den statischen Aspekt der Software beschränkt und könnte

ebenso für den dynamischen oder historischen Aspekt erweitert wären. [Wettel 2010, 60ff] nutzt beispielweise sogenannte *Age Maps* und *Time Travels* zur Darstellung von erstellten oder geänderten Entitäten und visualisiert dies anhand einer Farbskala.

Im Falle der Visualisierung der Methodentypen zeigte sich ein großer Teil der Methoden als nicht kategorisierbar. Eine Analyse von Methoden und der darauf folgenden neuen Einteilung der definierten Kategorien könnte eine detailliertere Aussage über die Methoden ermöglichen und die Qualität der Visualisierung erhöhen.

Zusammenfassend betrachtet bietet die Stadtmetapher weiterhin Potential zur Erweiterung und Variation der Darstellungsformen, was die Softwareanalyse zukünftig noch besser unterstützen kann. Die Erweiterung der Stadtmetapher und die Möglichkeit, diese im engen Rahmen zu konfigurieren, soll diese Entwicklung unterstützen und genauere Aussagen über Softwaresysteme ermöglichen.

Anhang – Erläuterungen zur Konfiguration und Ausführung

Die folgende Beschreibung soll die Änderungen an den Dateien zur Ausführung des Transformationsprozesses der einzelnen Varianten der Stadtmetapher verdeutlichen. Zusätzlich werden die Konfigurationsmöglichkeiten der *CitySettings.java*-Datei anhand der einzelnen Konstanten aufgezeigt.

Der bisherige Startpunkt zur Ausführung des Transformationsprozesses von *FAMIX*-Dateien zu *X3D*-Dateien war die Datei *Famix2City.launch* im Paket *org.svis.generator.releng*, welche wiederum den Workflow der Datei *Famix2City.mwe2* im Paket *org.svis.generator.run.src* abarbeitet. Diese Launch-Datei sowie der Workflow wurden zur flexibleren Ausführung der unterschiedlichen Varianten der Stadtmetapher für jede Variante getrennt angelegt. Es existieren somit jeweils eine Startdatei *Famix2City_Bricks.launch* und eine *Famix2City_Panels.launch,* die auch auf ihre etwaigen Workflows *Famix2City_Bricks.mwe2* und *Famix2City_Panels.mwe2* zugreifen. Diese gesonderten Workflows waren nötig, da sie wiederum unterschiedliche Dateien bei den Modell-zu-Modell-Transformationen durchlaufen. Es finden sich daher im Paket *org.svis.genrator.city.m2m* auch die Dateien *City2City_Bricks.java* und ebenfalls die *City2City_Panels.java*. Der originale Workflow zur Darstellung der Klassen als monolithische Blöcke blieb erhalten und ist nicht von diesen Erweiterungen betroffen.

```
5     public static Schemes SET_SCHEME = Schemes.VISIBILITY;
6     public static ClassElementsModes SET_CLASS_ELEMENTS_MODE;
7     public static ClassElementsSortModesCoarse SET_CLASS_ELEMENTS_SORT_MODE_COARSE;
8     public static ClassElementsSortModesFine SET_CLASS_ELEMENTS_SORT_MODE_FINE;
9     public static boolean SET_CLASS_ELEMENTS_SORT_MODE_FINE_DIRECTION_REVERSED;
10    public static boolean SET_SHOW_BUILDING_BASE = true;
11    public static Bricks.Layout SET_BRICK_LAYOUT;
12    public static boolean SET_SHOW_ATTRIBUTES_AS_CYLINDERS = true;
13    public static Panels.SeparatorModes SET_PANEL_SEPARATOR_MODE;
14
15    public static enum Schemes { VISIBILITY, TYPES; };
16    public static enum ClassElementsModes {
17        METHODS_ONLY, ATTRIBUTES_ONLY, METHODS_AND_ATTRIBUTES; }
18    public static enum ClassElementsSortModesCoarse {
19        UNSORTED, ATTRIBUTES_FIRST, METHODS_FIRST; }
20    public static enum ClassElementsSortModesFine {
21        UNSORTED, ALPHABETICALLY, SCHEME, NOS; }
22    public static enum Bricks {;
23        public static enum Layout { STRAIGHT, BALANCED, PROGRESSIVE; }
24    }
25    public static enum Panels {;
26        public static enum SeparatorModes { NONE, GAP, SEPARATOR; }
27    }
```

Listing 4: *CitySettings.java* Konfiguration

Eine teilweise Erläuterung der Konfigurationsmöglichkeiten findet sich im Abschnitt 3.4, die anhand von Listing 4 nun detaillierter beschrieben wird. Das Listing ist ein Zusammenschnitt der in der *CitySettings.java* tatsächlich vorhandenen Definition und fokussiert die wesentlichen Aspekte der Konfiguration. Die Definitionen sind zudem nicht

vollständig ausgeführt und dienen nur der Veranschaulichung der Funktionsweise. Die Datei *CitySettings.java* beinhaltet zudem weitere *JavaDoc*-Kommentierungen. In den Zeilen 5-13 des Listings stehen die Variablen zur Konfiguration der Metapher, zu erkennen an den Variablennamen, welche mit *Set_* beginnen. Diese Variablen finden sich in *CitySettings.java* gleich zu Beginn der Definitionen, um eine schnelle Änderung der Konfigurationen zu gewährleisten. Darunter in den Zeilen 15-27 sind die verschiedenen Werte, die von den *Set_*-Variablen angenommen werden können.

Als Beispiel zur Funktionsweise dient der in Zeile 5 zugewiesene Wert *SET_SCHEME = Schemes.VISIBILITY*, welche das Farbschema der Entitäten für die Zugriffsmodifikatoren setzt. Der zweite gültige Wert wäre *TYPES*, zu sehen in Zeile 15. *TYPES* setzt den Modus zur Anzeige der Attributdatentypen und der unterschiedlichen Methodentypen wie Konstruktor, Getter-Methode usw.

*SET_CLASS_ELEMENTS_*MODE in Zeile 6 steuert die Anzeige der Klassenbestandteile. Es können entweder Methoden, Attribute oder beide gleichzeitig angezeigt werden. Die Definitionen hierzu stehen im Enumerator *CLASS_ELEMENTS_MODE* in den Zeilen 16-17.

SET_CLASS_ELEMENTS_SORT_MODE_COARSE stellt entsprechend Zeile 18 die grobe Sortierung der zwei verschiedenen Klassenbestandteile untereinander ein. *METHODS_FIRST* ist dabei die Einstellung, welche in den Abbildungen konsequent genutzt wurde. Für die Backstein-Variante ist im Falle der Visualisierung beider Klassenbestanteile (*SET_CLASS_ELEMENTS_MODE == METHODS_AND_ATTRIBUTES*) die Einstellung *UNSORTED* anzuraten, da sich sonst trotz weiterer feinerer Sortierungen, zwei optisch ähnliche Bereiche bilden, da sich Attribute und Methoden optisch nicht voneinander unterscheiden.

SET_CLASS_ELEMENTS_SORT_MODE_FINE definiert die Sortierung innerhalb desselben Klassenbestanteil-Typs (Methode oder Attribut). *SCHEME* sortiert dabei nach dem ausgewählten *SET_SCHEME*, also entweder Typ der Methode/Attribut oder Zugriffsmodifikator. Die anderen Optionen sind selbsterklärend. Eine automatische dritte Sortierebene schließt sich an, welche innerhalb identischer Vergleichswerte eine noch feinere Sortierung vornimmt. Diese dritte Sortierebene wird automatisch bestimmt. Im Fall von *SCHEME* wird als dritte Sortierebene *NOS* gewählt und umgekehrt.

SET_CLASS_ELEMENTS_SORT_MODE_FINE_DIRECTION_REVERSED kann durch Setzen auf *TRUE* die feinere Sortierung insgesamt umkehren.

SET_SHOW_BUILDING_BASE steuert die Anzeige des dunklen Gebäudesockels unterhalb der Gebäudesegmente. Der zugewiesene Wert *FALSE* deaktiviert die Anzeige dieses Elementes.

SET_BRICK_LAYOUT steuert das Layout der Backsteine, indem die Berechnung der Grundfläche anhand unterschiedlicher Basiswerte erfolgt. *STRAIGHT* bildet immer eine Grundfläche von 1x1 Backsteinen. Somit werden die Backsteine ausschließlich in die Höhe gestapelt. *PROGRESSIVE* nimmt als Basiswert die Anzahl an zu visualisierenden Backsteinen des Gebäudes. *BALANCED* nutzt im Fall der Darstellung von Methoden die Anzahl der Attribute und umgekehrt. Falls Methoden und Attribute gleichzeitig angezeigt werden sollen, wird analog zu *PROGRESSIVE* die Summe beider Arten von Elementen genutzt und ist in diesem Szenario identisch zum *PROGRESSIVE*-Layout.

SET_SHOW_ATTRIBUTES_AS_CYLINDERS gibt die Möglichkeit zur Anzeige der Attribute als Zylinder anstatt Boxen. Diese Einstellung hat nur in der Paneel-Variante eine Bedeutung.

SET_PANEL_SEPARATOR_MODE steuert in der Paneel-Variante die Anzeige der Leerräume zwischen den Segmenten. *NONE* Fügt die Elemente nahtlos zusammen, was allerdings eine Differenzierung einzelner Elemente unmöglich macht. *GAP* belässt einen kleinen Abstand zwischen den Elementen, ähnlich den Segmenten in der Backstein-Variante. *SEPARATOR* fügt spezielle schwarze Separatoren zwischen den Segmenten ein, um die Differenzierung zu erleichtern, da somit der Kontrast maximiert wird.

Den erstellten *X3D*-Dateien werden die hier beschriebenen Variablen in einem gesonderten Tag *SettingsInfo* hinzugefügt, um die Konfiguration innerhalb der erstellten Visualisierung zu dokumentieren. Zusätzlich wurden vier Kameraperspektiven, in *X3D Viewports* genannt, deklariert, um eine bequemere Navigation innerhalb der visualisierten Stadt zu ermöglichen.

Literatur- und Quellenverzeichnis

Alam, S. & Dugerdil, P., 2007a. EvoSpaces: 3D Visualization of Software Architecture. In *Int. Conf. Softw. Eng. Knowl. Eng.* S. 500–506.

Alam, S. & Dugerdil, P., 2007b. EvoSpaces Visualization Tool: Exploring Software Architecture in 3D. In *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, S. 269–270.

Balogh, G. u. a., 2016. Using the City Metaphor for Visualizing Test-Related Metrics. *IEEE International Conference on Software Analysis, Evolution and Reengineering*.

Balogh, G. & Beszédes, A., 2013. CodeMetropolis-code visualisation in MineCraft. ... *Code Analysis and Manipulation*.

Balogh, G., Szabolics, A. & Beszedes, A., 2015. CodeMetropolis: Eclipse over the city of source code. *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings*, S.271–276.

Bohnet, J. & Döllner, J., 2011. Monitoring code quality and development activity by software maps. *Proceeding of the 2nd working on Managing technical debt - MTD '11*, S.9.

Borgo, R. u. a., 2013. Glyph-based Visualization: Foundations, Design Guidelines, Techniques and Applications. *Eurographics State of the Art Reports*, S.39–63..

Caserta, P. & Zendra, O., 2011. Visualization of the static aspects of software: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7), S.913–933.

Chapin, N. & Lau, T.S., 1996. Effective Size: An Example of Use from Legacy Systems. *Journal of Software Maintenance*, 8(2), S.101–116.

Charters, S.M. u. a., 2002. Visualisation for informed decision making; from code to components. *Proceedings of the 14th international conference on Software engineering and knowledge engineering - SEKE '02*, S.8.

CodeCity, 2016. CodeCity. URL: http://wettel.github.io/codecity.html [Letzter Zugriff am 16.12.2016].

CodeMetropolis, 2016. CodeMetropolis. URL: http://web.sed.hu/codemetropolis [Letzter Zugriff am 21.12.2016].

Deterding, S., 2012. Gamification. *Interactions*, 19(4), S.14.

Diehl, S., 2007. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*,

Dubois, D.J. & Tamburrelli, G., 2013. Understanding gamification mechanisms for software

development. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, S.659.

Ducasse, S. u. a., 2011. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. , (November).

Eclipse, 2016. Eclipse. URL: https://eclipse.org/ [Letzter Zugriff am 06.11.2016].

Eisenbarth, T. u. a., 2003. Locating Features in Source Code. *IEEE Transactions on Software Engineering*, 29(3), S.210–224.

EvoSpaces, 2016. EvoSpaces. URL: http://www.inf.usi.ch/projects/evospaces/ [Letzter Zugriff am 05.11.2016].

Holten, D., 2006. Hierarchical Edge Bundles: Visualizaiton of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5), S.741–748.

Johnson, B. & Shneiderman, B., 1991. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. *Proceeding Visualization '91*, S.284–291.

Knight, C., 2000. *Virtual software in reality*. Durham University.

Knight, C. & Munro, M., 2001. Mindless Visualisations. , (October).

Kobayashi, K. u. a., 2012. Feature-gathering dependency-based software clustering using Dedication and Modularity. *IEEE International Conference on Software Maintenance, ICSM*, S.462–471.

Kobayashi, K. u. a., 2013. SArF map: Visualizing software architecture from feature and layer viewpoints. *IEEE International Conference on Program Comprehension*, S.43–52.

Langelier, G., Sahraoui, H. & Poulin, P., 2005. Visualization-based Analysis of Quality for Large-scale Software Systems. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. New York, NY, USA: ACM, S. 214–223.

Lanza, M. u. a., 2013. Manhattan: Supporting real-time visual team activity awareness. *IEEE International Conference on Program Comprehension*, S.207–210.

Lanza, M., Gall, H. & Dugerdil, P., 2009. EvoSpaces: Multi-dimensional Navigation Spaces for Software Evolution. *2009 13th European Conference on Software Maintenance and Reengineering*, S.293–296.

Lanza, M., Marinescu, R. & Ducasse, S., 2005. *Object-Oriented Metrics in Practice*, Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Maletic, J.I. u. a., 2001. Visualizing object-oriented software in virtual reality. In

Proceedings - IEEE Workshop on Program Comprehension. S. 26–35.

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

Malhan, P. & Singh, S., 2014. Using Simulation and modeling to visualize Object- Oriented Software. , 1(6), S.1211–1221.

Minecraft, 2016. Minecraft. URL: https://minecraft.net/de/ [Letzter Zugriff am 21.12.2016].

Müller, R. u. a., 2011. Generative Software Visualizaion: Automatic Generation of User-Specific Visualisations. *Proceedings of the International Workshop on Digital Engineering*, S.45–49.

Nierstrasz, O., Ducasse, S. & Girba, T., 2005. The Story of Moose: An Agile Reengineering Environment. *SIGSOFT Softw. Eng. Notes*, 30(5), S.1–10.

Panas, T., 2005. A Framework for Reverse Engineering. In *A Framework for Reverse Engineering*. S. 161.

Panas, T. u. a., 2007. Communicating software architecture using a unified single-view visualization. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS*. S. 217–226.

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

Panas, T., Berrigan, R. & Grundy, J., 2003. A 3D Metaphor for Software Production Visualization.

Scanniello, G. u. a., 2010. *Architectural layer recovery for software system understanding and evolution*,

SkyscrapAR, 2016. SkyscrapAR. URL: https://github.com/rodrigorgs/SkyscrapAR [Letzter Zugriff am 19.12.2016].

Souza, R. u. a., 2012. SkyscrapAR: an augmented reality visualization for software evolution. *Proc. of 2nd Brazilian ...*, S.17–24.

Vizz3D, 2016. Vizz3D. URL: http://vizz3d.sourceforge.net/ [Letzter Zugriff am 22.12.2016].

Wettel, R., 2010. *Software Systems as Cities*.

Wettel, R. & Lanza, M., 2008. Visual exploration of large-scale system evolution. *Proceedings - Working Conference on Reverse Engineering, WCRE*, S.219–228.

Wilde/Hess, 2007. Forschungsmethoden der Wirtschaftsinformatik: Eine empirische Untersuchung. , 49, S.280–287.

X3D, 2016. X3D. URL: http://www.web3d.org/sites/default/files/page/About Web3D Consortium/What_Is_X3D_2016.pdf [Letzter Zugriff am 22.12.2016].

Xtend, 2016. Xtend. URL: http://www.eclipse.org/xtend/ [Letzter Zugriff am 20.12.2016].

Xtext, 2016. Xtext. URL: https://eclipse.org/Xtext/ [Letzter Zugriff am 20.12.2016].

Zhang, E. & Stasko, J., 2000. Focus+ Context Display and Navigation Techniques for Enhancing Radial, Space-Filling Hierarchy Visualizations. , S.57–68.

Zilch, D., 2015. Generative und modellgetriebene Softwarevisualisierung am Beispiel der Stadtmetapher. , S.32.

Selbstständigkeitserklärung

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.

_____
Christian Schulze
Leipzig, 03. Januar 2017