

## Masterseminar

# Softwarevisualisierung für Virtual Reality

Marcel Pütz

Fakultät für Informatik

In Zusammenarbeit mit der QAware GmbH

SS 2017

Visualisierung spielt in vielen Bereichen der Wissenschaft eine wichtige Rolle. In der Informatik ist die Softwarevisualisierung bei der Größe und Komplexität zeitgemäßer Softwaresysteme eine hilfreiche Unterstützung für die Entwicklung, Exploration und Analyse von Software.

Mit dem zunehmend Einzug haltenden Medium der Virtuellen Realität, bekommt die Softwarevisualisierung neue Möglichkeiten und besseren Zugang zu dreidimensionalen Modellen der Visualisierung.

In dieser Arbeit sollen Modelle der dreidimensionalen Softwarevisualisierung untersucht werden. Dazu wird mithilfe den Ergebnissen einer Umfrage erarbeitet, was eine Softwarevisualisierung überhaupt erreichen soll. Anhand der daraus abgeleiteten Anforderungen, können die verschiedenen Modelle miteinander verglichen werden. Die mögliche Realisierung der Modelle für die Microsoft HoloLens spielt in dieser Arbeit ebenfalls eine Rolle.

Es wird besonders auf die Metapher der Software-Stadt eingegangen, aber auch kreativ nach neuen, alternativen Metaphern gesucht. Am Schluss der Arbeit wird mithilfe des Gegenüberstellung und der Bewertung der behandelten Modelle eine Empfehlung für die Realisierung einer Softwarevisualisierung für die HoloLens gegeben.

# Inhaltsverzeichnis

<b>1 Motivation für 3D-Softwarevisualisierung in VR</b>	<b>3</b>
<b>2 Evaluierung von Kriterien für eine gute 3D-Softwarevisualisierung</b>	<b>4</b>
2.1 Mögliche Metriken zur Visualisierung . . . . .	4
2.2 Auswahl der wichtigsten Metriken und Randbedingungen . . . . .	7
<b>3 Vorhandene 3D-Modelle von Software-Städten</b>	<b>9</b>
3.1 CodeCity von Wettel und Lanza . . . . .	9
3.2 SoftVis3D – ein Plugin für SonarQube . . . . .	12
<b>4 Alternative Ansätze</b>	<b>15</b>
4.1 CodeForest . . . . .	15
4.2 CodeUniverse . . . . .	18
<b>5 Technische Machbarkeit für die Microsoft HoloLens</b>	<b>20</b>
<b>6 Fazit</b>	<b>21</b>
6.1 Vergleich der behandelten Ansätze . . . . .	21
6.2 Ausblick . . . . .	23
<b>Literatur</b>	<b>24</b>

# 1 Motivation für 3D-Softwarevisualisierung in VR

Niemand konnte bislang unser Sonnensystem von außen betrachten. Dennoch haben wir alle eine ziemlich gute Vorstellung, wie dieses aufgebaut ist. Durch die *Visualisierung* der Planeten und der Sonne entsteht in uns ein geistiges Abbild der Realität. Das Konzept komplexe Realitäten zu abstrahieren und zu visualisieren, um dadurch die Realität besser verstehen zu können, ist in vielen Disziplinen der Wissenschaft vertreten.

Neben Wissenschaften wie Physik, Chemie oder Biologie, nimmt Visualisierung auch besonders in der Informatik eine wichtige Rolle ein. In vielen Bereichen müssen Informationen in eine visuelle Form gebracht, die für das menschliche Auge besser zu lesen sind. Im Allgemeinen hat Gershon Visualisierung wie folgt definiert:

„*Visualization is the process of transforming information into a visual form, enabling users to observe the information. The resulting visual display enables the scientist or engineer to perceive visually features which are hidden in the data but nevertheless are needed for data exploration and analysis.*“ [6]

In [4] wird Visualisierung in zwei Arten unterteilt. Die *wissenschaftliche Visualisierung* – wie es beispielsweise die Visualisierung unseres Sonnensystems ist – beschäftigt sich mit der Aufbereitung von physikalischen Daten, wohingegen sich die *Informations-Visualisierung* mit abstrakten Daten befasst. Demzufolge handelt es sich bei der *Softwarevisualisierung* um eine Informations-Visualisierung und die abstrakten Daten sind alle Artefakte einer Software. Softwarevisualisierung ist für Diehl die „*visualization of artifacts related to software and its development process*“ [4].

Dies ist eine sehr weit gefasste Definition. In dieser Arbeit soll der Fokus mehr auf den Nutzen für den Betrachter gelegt werden und deshalb wird für diese Arbeit folgende Zielvorstellung für eine Softwarevisualisierung verwendet:

*Softwarevisualisierung ist die bildliche oder auch metaphorische Darstellung einer Software, um dem Betrachter durch Vereinfachung und Abstraktion das bessere Verständnis oder die einfachere Analyse von Software zu ermöglichen.*

Soll zum Beispiel die zu Grunde liegende Struktur einer Software Außenstehenden erklärt werden, gelingt das anfänglich kaum mit der Betrachtung des Source-Codes. Es wird sich mit UML-Diagrammen oder anderen schematischen Darstellungsformaten behelfen, sodass sich der Außenstehende ein Überblick über die Software verschaffen kann.

So wie UML-Diagramme, war die Darstellungsform der Softwarevisualisierung bislang meist zweidimensional. Mit dem Einzug der Virtuellen Realität und Datenbrillen wird dieser Disziplin der Visualisierung jedoch wortwörtlich ein neuer Raum an Möglichkeiten eröffnet. In dieser Arbeit soll deshalb herausgefunden werden, wie man das

neue Medium der Virtuellen Realität am besten für die Softwarevisualisierung nutzen kann.

Dabei wird zunächst untersucht welche Metriken einer Software für die Visualisierung von Interesse sind, um dann zu evaluieren, welche Metapher oder welches 3D-Modell am besten dafür geeignet ist. Die bereits am meisten verbreitete Metapher der Software-Stadt und unterschiedliche Implementierungen werden untersucht, aber auch alternative Ansätze werden kreativ entwickelt.

## **2 Evaluierung von Kriterien für eine gute 3D-Softwarevisualisierung**

### **2.1 Mögliche Metriken zur Visualisierung**

Bevor auf einzelne Modelle der Softwarevisualisierungen eingegangen wird, werden in diesem Kapitel Metriken definiert, die es in einer Softwarevisualisierung darzustellen gilt. Anhand derer und deren möglichen Darstellung, können die verschiedenen Modelle miteinander verglichen und bewertet werden.

Grundlegend können die zu visualisierende Informationen laut [4] in drei Kategorien aufgeteilt werden.

**Statik** sind die Informationen, die ohne die Ausführung der Software generiert werden können. Darunter fallen Struktur der Pakete, aber auch Abhängigkeiten von Klassen und Funktionen oder Code-Violations.

**Dynamik** beschreibt die Informationen, die zur Laufzeit einer Software generiert werden können. Das kann die Abfolge des ausgeführten Codes, Laufzeit von Code-Segmenten oder Aufruf-Häufigkeit von Funktionen sein.

**Evolution** beschreibt den zeitlichen Verlauf statischer oder dynamischer Metriken.

Um zu evaluieren, welche Informationen der genannten Kategorien für Nutzer von Softwarevisualisierungen besonders wichtig sind, wurde im Rahmen dieser Arbeit Mitarbeiter der *QAware GmbH* (im Folgenden mit QAware bezeichnet) befragt. Die QAware ist ein Projekthaus mit den Kerngeschäften Diagnose, Sanierung, Exploration und Realisierung von Software [11]. Durch die Erfahrung in Projekten für namhafte Kunden, zeichnen sich die Mitarbeiter durch fundiertes Wissen und Expertise aus. Es wurden insgesamt 22 Mitarbeiter mit unterschiedlichen Rollen in der Softwareentwicklung befragt.

Alle die von den Befragten genannten Metriken können in die drei Kategorien eingeordnet werden. In Abbildung 1 wird die Verteilung des Interesses an den Kategorien dargestellt, wobei die Befragten insgesamt 37 Wünsche äußerten. Es ist gut zu erkennen, dass die Statik das größte Interesse findet. Rund ein Drittel der genannten Metriken sind in der Dynamik anzusiedeln. Die Visualisierung der Evolution einer

Software nimmt den kleinsten Anteil ein, ist jedoch mit rund einem Zehntel nicht zu vernachlässigen.

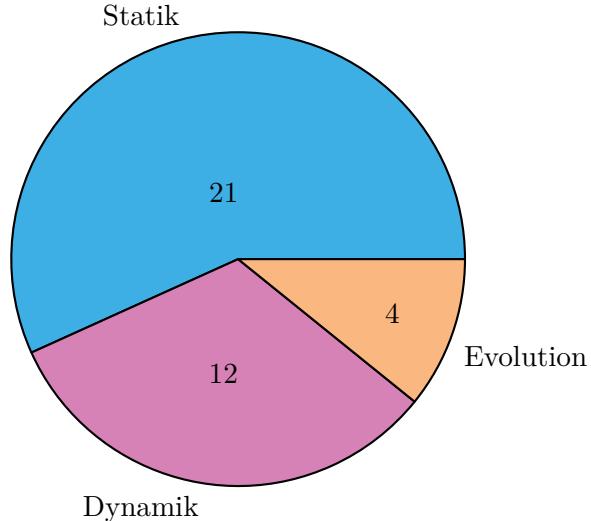


Abbildung 1: Verteilung der Interesse nach Kategorien

In den Tabellen 1 - 3 sind die Ergebnisse, die aus der Umfrage hervorgegangen, in 22 einzelnen Metriken konsolidiert und nach Kategorie gruppiert. Die ID dient zur Nachverfolgbarkeit der Metriken. Die zweite Spalte beschreibt die Metrik, also die Informationen, die den Befragten gerne visualisiert hätten und der Nutzen ist die Antwort auf die Frage, was dadurch erreicht werden soll. In der vierten Spalte sind alle Rollen gelistet, die ein Interesse daran haben diese Metrik zu visualisieren. Die letzte Spalte gibt Aufschluss darüber, von wie vielen Befragten die Metrik genannt wurde und ist damit ein guter Indikator, wie wichtig die Unterstützung dieser Metrik in einem 3D-Modell ist.

Tabelle 1: **Statik** – Gewünschte Metriken

ID	Metrik	Nutzen	Zielgruppe	Anzahl
S1	Lines of Code	Kennenlernen der Software	SW-Entwickler	1
S2	Komplexität	Komplexitätsnester identifizieren	SW-Entwickler	3
S3	Testabdeckung	Fehlende Abdeckung sehen	SW-Architekt, SW-Entwickler	1

*Fortsetzung auf nächster Seite...*

Tabelle 1 – Fortsetzung von vorheriger Seite

S4	Struktur	Übersicht über den Aufbau der Software erhalten, Schachtelungstiefe sehen, Software kennen lernen	SW-Entwickler, SW-Architekt, SW-Sanierer	5	
S5	Abhängigkeiten	Architektur Verletzungen, Inkonsistenzen erkennen, Datenmodell analysieren, Auswirkungen von Refactoring beurteilen	SW-Architekt, SW-Entwickler, Technischer Chef Designer	7	
S6	Code-Violations	Problemgebiete identifizieren	SW-Architekt, SW-Entwickler, Technischer Chef Designer	4	

Interessanterweise sind Anzahl von Methoden oder Attributen in einer Klasse, die als Metriken in einer der ersten Software-Städte von Wettel und Lanza visualisiert wurden [16], überhaupt nicht genannt worden. Dagegen sticht heraus, dass die **Struktur** (S4) der Software und die dazugehörigen **Abhängigkeiten** (S5) für die Mitarbeiter der QAware besonders interessant sind.

Danach sind **Code-Violations** (S6), zum Beispiel Verletzungen von vereinbarten *Code-Conventions*, die am häufigsten genannte Metrik. Diesen drei Metriken waren zudem auch Mitarbeitern mit unterschiedlichen Rollen in Software-Projekten wichtig. Das heißt mit deren Visualisierung können unterschiedliche Zielgruppen gleichermaßen Nutzen daraus ziehen.

Tabelle 2: **Dynamik** – Gewünschte Metriken

ID	Metrik	Nutzen	Zielgruppe	Anzahl
D1	Anzahl Aufrufe	Hotspots, tote Stellen, Sackgassen erkennen	SW-Architekt, SW-Sanierer	5
D2	Laufzeiten	Zeitintensive Abschnitte erkennen	SW-Architekt, SW-Sanierer, Test-Betreuer	5
D3	Laufzeitfehler	Sehen wo die Software fehlschlägt	SW-Sanierer	1
D4	Ressourcen-Auslastung	Ausreißer und Trends erkennen	SW-Sanierer	1

Bei der Kategorie Dynamik ist das Ergebnis sehr eindeutig. Die Befragten waren sich einig, dass **Anzahl der Aufrufe** (D1) von einzelner Funktionen und deren **Laufzeiten** (D2) von besonderem Interesse ist.

Die Metrik von auftretenden **Laufzeitfehlern** (D3) wurde zwar nur von einer Person genannt, kann aber besonders bei der Analyse oder Sanierung von fehlerhafter Software sehr hilfreich sein.

**Ressourcen-Auslastung** (D4) wird in dieser Arbeit als ein wenig ausschlaggebendes Kriterium erachtet, da diese wenig Einfluss auf die Modellierung der Visualisierung hat. Vielmehr wäre das eine Metrik, die man parallel zu der eigentlichen Softwarevisualisierung darstellen könnte.

Tabelle 3: **Evolution** – Gewünschte Metriken

ID	Metrik	Nutzen	Zielgruppe	Anzahl
E1	Zeitlicher Entwicklung statischer Metriken	Langzeitentwicklung beobachten, Abweichung vom Zielbild erkennen, Trends verfolgen	SW-Architekt, Technischer Chef Designer	3 
E2	Zeitlicher Verlauf eines Themas	Bestimmtes Thema nachverfolgen und Qualität überprüfen	Projektleiter	1 

Durch die im Vergleich zu den anderen beiden Kategorien eher geringere Beachtung der Evolution der Software, lässt sich nicht zwangsläufig schlussfolgern, dass diese nicht hilfreich sei. Grund dafür kann vielmehr die höhere Anzahl von Software-Architekten und Software-Entwicklern unter den Befragten sein. Aus Management-Sicht wie die eines Projektleiters oder auch eines Technischen Chef Designers kann eine langfristige Beobachtung (E1) zur Qualitätssicherung einen wertvollen Beitrag leisten. Als statische Metriken, die über einen gewissen Zeitraum interessant zu beobachtet wären, wurden vornehmlich Komplexität, Struktur und Abhängigkeiten genannt.

## 2.2 Auswahl der wichtigsten Metriken und Randbedingungen

Durch die Umfrage sind einige Metriken hervorgegangen, auf die bei der Suche nach einem geeigneten 3D-Modell für Softwarevisualisierung besonders Wert zu legen ist. Die mögliche Visualisierung dieser Metriken finden sich in den folgenden Kriterien für ein gutes 3D-Modell wieder. Teilweise sind manche Metriken zusammengefasst und andere, wenig frequentierte, sind in den Kriterien vernachlässigt.

**Statische Metriken (S1 - S3)** Die 3D-Softwarevisualisierung soll verschiedene Metriken einer Klasse konfigurierbar darstellen können, um dem Betrachter die Arte-

fakte, wie zum Beispiel Klassen, einer Software bezüglich der ausgewählten Metrik vergleichbar zu machen.

**Struktur (S4)** Die übersichtliche Darstellung der Struktur soll bei der Visualisierung der Statik ein vorrangiges Ziel darstellen. Der Nutzer soll auf einen Blick eine gut Übersicht über die Struktur der Software erhalten.

**Abhängigkeiten (S5)** Die Abhängigkeiten sollen dabei auf die Struktur der Software abgebildet werden können, ohne die Übersichtlichkeit negativ zu beeinflussen.

**Dynamik (D1 + D2)** Neben der Statik soll die Dynamik (primär die Anzahl der Aufrufe und deren Laufzeiten) visualisiert werden können.

**Evolution (E1)** Die 3D-Softwarevisualisierung soll eine zeitliche Entwicklung der Software darstellen können, um verschiedene Metriken über einen bestimmten Zeitraum zu beobachten.

Neben der Möglichkeit dies Metriken zu visualisieren, sind einige Randbedingungen zu beachten.

In der Softwarevisualisierung wurde in der Literatur oftmals der Begriff *Habitability* von Gabriel in [5] aufgegriffen. Dieser bedeutet, dass sich ein Entwickler in einer Software zuhause fühlt. Dafür muss der Entwickler die Software nicht zwangsläufig selbst entwickelt haben. Die meisten Menschen fühlen sich auch in ihren Häusern zuhause, die sie nicht selbst gebaut haben, führt Gabriel als Metapher an.

**Habitability** Die 3D-Softwarevisualisierung sollte die Habitability fördern, oder diese bestenfalls sogar erreichen. Dies kann jedoch nur erreicht werden, wenn das Modell der Softwarevisualisierung sich nicht durch geringe Änderungen der Software grundlegend verändert. Nur so kann gewährleistet werden, dass ein vertrauter Nutzer der Software sich nicht nach jedem Release die Habitability erneut aneignen muss.

**Drilldown** Die Interaktion mit der visualisierten Struktur der Software soll so gestaltet werden, dass sich der Nutzer auf UnterPakete der Software konzentrieren kann und der Rest der Visualisierung in den Hintergrund rückt.

**Technische Machbarkeit** Die 3D-Softwarevisualisierung soll auf eine Microsoft *HoloLens*<sup>1</sup> gebracht werden können. Bei diesem Kriterium soll bewertet werden, ob eine bestehende Implementierung einer 3D-Softwarevisualisierung technisch auf die HoloLens portiert werden kann. Modelle, die so noch nicht implementiert sind (siehe

---

<sup>1</sup><https://www.microsoft.com/microsoft-hololens>

Kapitel 4), werden positiv bewertet, da auf Standardtechnologien für die Entwicklung für Windows Holographic gesetzt werden kann (siehe Kapitel 5).

Im nächsten Kapitel werden vorhandene 3D-Modelle vorgestellt und anhand der erarbeiteten Kriterien bewertet. Im Fazit dieser Arbeit werden alle behandelten Ansätze gegenübergestellt und übersichtlich verglichen.

### 3 Vorhandene 3D-Modelle von Software-Städten

Eine der ersten 3D-Visualisierung von hierarchischen Informationen wurde 1997 von Andrews et al. in [1] vorgestellt.

Mit den darin beschriebenen Pyramiden können Ordner-Strukturen eines Unix-Systems dargestellt werden. Plateaus entsprechen Ordnern, Quader einzelnen Dateien. In Abbildung 2 ist die Datei-Struktur vom TeX Home-Ordner von Andrews in einer Übersicht (2a) und im Detail (2b) zu sehen.

Auch eine Interaktion in Form von Fokussierung eines Unterordners wurde schon unterstützt. Der ausgewählte Ordner wird dann zur neuen Basis der Pyramide.

Das Konzept von Andrews et al. weißt einige Parallelen mit der 10 Jahre später veröffentlichten Arbeit von Wettel und Lanza über eine sogenannte *CodeCity* [15] auf, welche im Folgenden genauer beleuchtet werden soll.

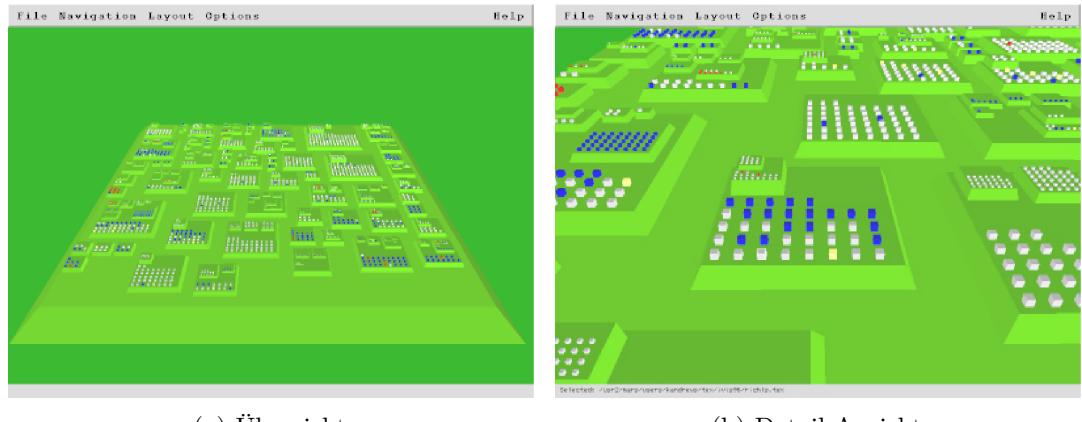


Abbildung 2: Information Pyramids [1]

#### 3.1 CodeCity von Wettel und Lanza

2007 stellten Wettel und Lanza ein Konzept vor, mit dem sich eine objektorientierte Software mithilfe der *Stadt-Metapher* visualisieren ließ. Im Vordergrund dieser Herangehensweise der Visualisierung stand die Habitability, welche durch die Metapher und die damit einhergehende Möglichkeit der Orientierung erreicht werden sollte [15].

So wie bei Andrews et al. werden bei Wettel und Lanza Ordner bzw. Pakete durch Plateaus dargestellt, die jedoch weniger in die Höhe gehen sondern sich als Stadtviertel (engl. *districts*) in die Stadt-Metapher einfügen.

In diesen Vierteln, die bei geschachtelten Paketen auch aufeinander geschichtet sein können, ragen dann Klassen als Gebäude in die Höhe.

Die Software kann dann laut Wettel und Lanza wie eine echte Stadt erkundet werden. Die Stadt im Ganzen vermittelt eine gute Übersicht über die Software und deren Komplexität. Einzelne Pakete können dann wie bei einem Stadtrundgang besichtigt und Klassen wie Häuser besucht werden.

Die Inhalte der Klassen werden nicht weiter visualisiert, da sie für das größer gefasstes Verständnis einer Software nicht notwendig sei.

Für die Breite und Tiefe der Gebäude wurde für die CodeCity die Anzahl der Attribute (engl. *number of attributes (NOA)*) und für die Höhe die Anzahl der Methoden (engl. *number of methods (NOM)*) der visualisierten Klasse gewählt. In späteren Weiterentwicklungen der CodeCity wurden auch andere Metriken wie beispielsweise Komplexität unterstützt.

Für das Anordnung der Stadtviertel und der sich darin befindlichen Klassen verwendeten Wettel und Lanza einen *rectangle-packing* [15] Algorithmus, durch den die Elemente rekursiv der Größe nach in rechteckigen Grundflächen angeordnet werden.

In Abbildung 3 ist die Software ArgoUML als CodeCity abgebildet.

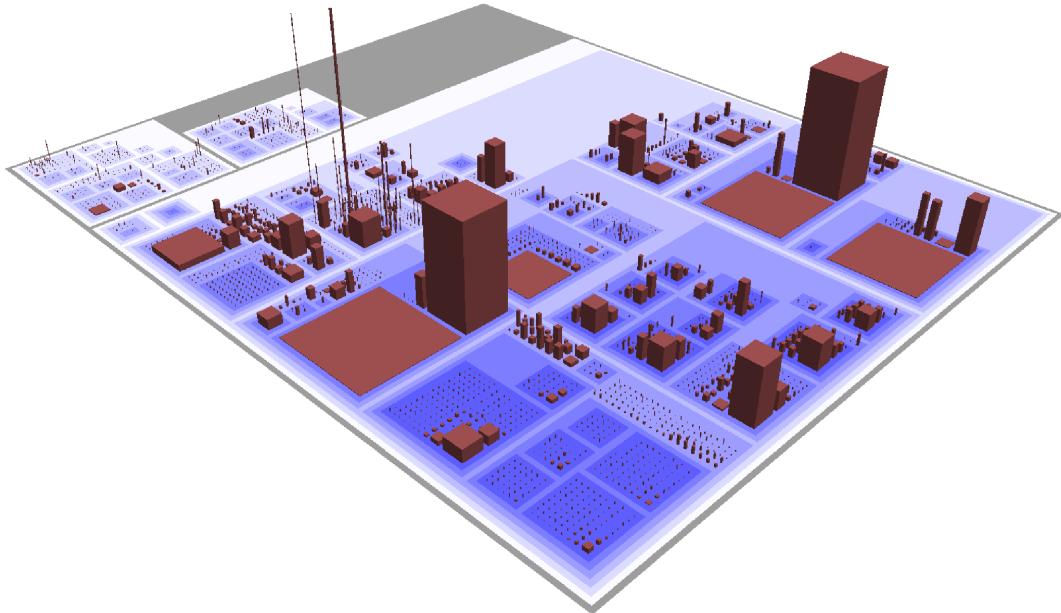


Abbildung 3: CodeCity von ArgoUML [16]

Alternativ zu der linearen Darstellung der Größe der Gebäude, wurden auch Mapping Algorithmen untersucht, bei denen das Stadtbild harmonischer aussieht und die

Habitability noch stärker angesprochen werden soll. Dabei werden die Klassen in Kategorien eingeteilt, die dann in sechs verschiedene Arten von Gebäuden – von Haus (engl. *house*) bis Wolkenkratzer (engl. *skyscraper*) – aufgeteilt werden.

## Vorteile

Die **Habitability** nimmt in der CodeCity einen großen Stellenwert ein, da das Modell primär dafür entworfen wurde.

Neben NOA und NOM ist in einer CodeCity auch S1 darstellbar. Die **Struktur** wird durch eine CodeCity sehr gut visualisiert und sogar **Abhängigkeiten** sind verfügbar. Die Abhängigkeiten von einer einzelnen Klasse ist in Abbildung 4a zu sehen.

Auch die **Evolution** einer Software kann durch ein Farbverlauf (in Abbildung 4b zu sehen) der Gebäude oder durch eine „Zitreise“ analysiert werden.

Der *Drilldown* wird insofern unterstützt, dass einzelne UnterPakete in einem neuen Fenster geöffnet werden.

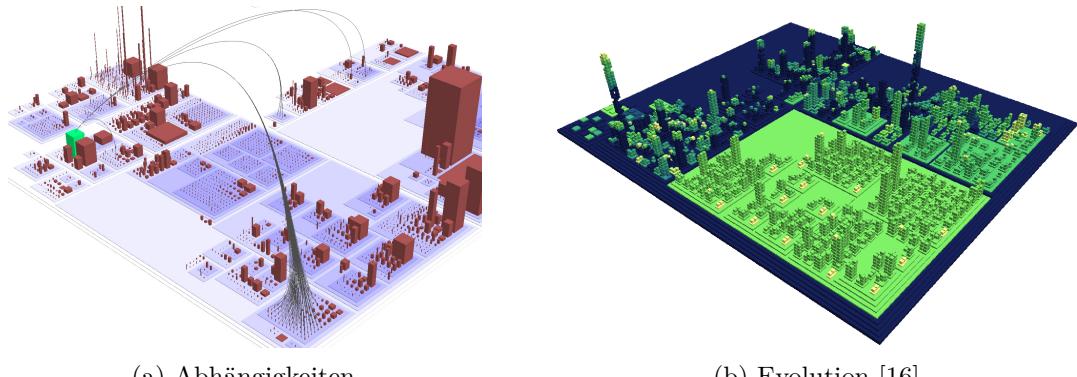


Abbildung 4: Vorteile einer CodeCity

## Nachteile

Bei Betrachtung von Abhängigkeiten in ganzen Paketen oder des gesamten Systems werden die Abhängigkeiten schnell unübersichtlich. Die Abhängigkeiten sind alle einfarbig und werden können auch nur von dem aktuell ausgewählten Element angezeigt werden.

Der größte Nachteil der Implementierung der CodeCity ist die veraltete **Technologie**. Es wurde auf *Moose*<sup>2</sup> gesetzt, eine Platform für die Analyse von Software [8]. Moose wurde seit 1996 mit *VisualWorks*<sup>3</sup> entwickelt, eine kommerzielle cross-platform Implementierung der Programmiersprache *Smalltalk*. Smalltalk wurde in den frühen

<sup>2</sup><http://moosetechnology.org/>

<sup>3</sup><http://www.cincomsmalltalk.com/main/products/visualworks/>

1970ern im *Xerox Palo Alto Research Center* entwickelt und ist eine der ersten objektorientierten Programmiersprache mit Konzepten, die zum Beispiel die JVM stark beeinflusst haben [7].

Seit 2008 wurde Moose von VisualWorks auf *Pharao*<sup>4</sup>, einer open-source Implementierung von Smalltalk, migriert. Seit 2009 wurde CodeCity nicht weiterentwickelt und auch der Workflow des Imports des Source-Code der zu visualisierende Software in ein von CodeCity lesbaren Format, funktioniert heute nicht mehr so, wie es Wettel und Lanza dokumentiert hatten. Durch die Verwendung von Moose ist die 3D-Visualisierung der CodeCity auch für die Microsoft HoloLens nicht effektiv portierbar.

CodeCity weiß viele durchdachte Konzepte auf, ist jedoch 2017 nicht mehr für neue Software produktiv einsetzbar.

### 3.2 SoftVis3D – ein Plugin für SonarQube

Eine ähnliche Implementierung der Stadt- Metapher wird seit 2012 von Rinderle entwickelt. Sein Projekt *SoftVis3D*, das er im Rahmen seiner Master-Thesis ins Leben rief, ist ein Plugin für *SonarQube*<sup>5</sup>. SonarQube ist eine open-source Plattform für statische Code-Qualität. Die in Java entwickelte Software kann verschiedene Qualitätskriterien einer Momentaufnahme des betrachteten Source-Codes im Browser darstellen und ermöglicht darüber hinaus Trends zu erkennen [3]. Mithilfe von Plugins kann die Funktionalität bei der Aggregation der dargestellten Daten erweitert werden. Von neuen Metriken bis hin zu ganzen Programmiersprachen kann Funktionalität ergänzt werden.

Nicht nur für die Quelle der dargestellten Daten können Plugins entwickelt werden. Auch für Darstellung der aggregierten Daten ist es möglich eigene Plugins zu realisieren. SoftVis3D ist ein solches Plugin, das jegliches Projekt, das in SonarQube eingebunden ist, als Software-Stadt darstellen kann.

Das Grundprinzip von SoftViz3D ist das der CodeCity: Klassen werden als Häuser visualisiert. Die Konfigurierbarkeit ist jedoch bei SoftVis3D weitaus höher. Der Grundfläche und der Höhe der Häuser kann jegliche verfügbare Metrik von SonarQube zugeordnet werden. Für Java-Projekt ohne weitere Plugins sind dies 80 verschiedene Metriken. Für das Farbschema stehen sechs verschiedene Metriken, wie Complexity und Coverage zur Verfügung. Hervorzuheben ist die Metrik *Package Name*, die Pakete und Klassen mit ähnlichem Namen in ähnlicher Farbe darstellt und so bei sprechender Benennung der Artefakte thematische Zusammengehörigkeit visualisiert.

Ein Java-Projekt der QAware mit 3.101 Klassen und 154,449 LOC als SoftVis3D Visualisierung ist in Abbildung 5 zu sehen. Als Konfiguration wurde Komplexität als Grundfläche, LOC als Höhe und Coverage als Farbe gewählt.

Neben dem District-Layout, wie es auch in CodeCity Verwendung findet, wurde im Zuge der Master-Thesis von Niedrich eine zweite Layout Strategie für SoftVis3D reali-

---

<sup>4</sup><http://pharo.org/>

<sup>5</sup><https://www.sonarqube.org/>

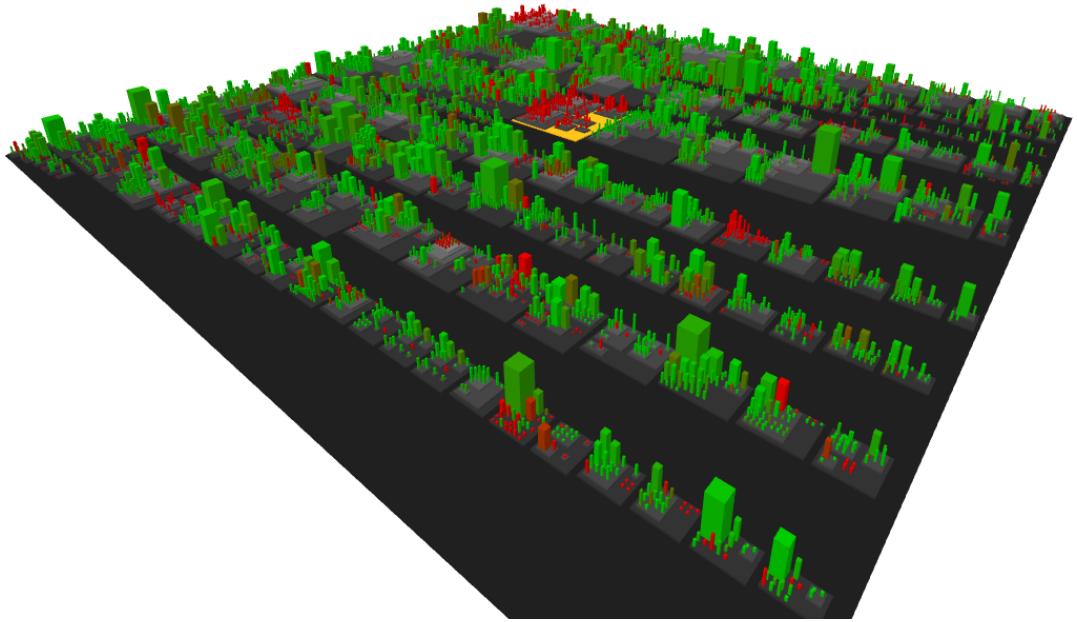


Abbildung 5: SoftVis3D

siert [10]. Der sogenannte *Evostreet*-Algorithmus wurde ursprünglich von Steinbrück für die bessere Darstellung der Evolution einer Software entwickelt [14]. Mit diesem Algorithmus soll sich die Struktur der Stadt nicht verändern und so eine Evolution der Software auch über längere Zeiträume ermöglichen. Die Pakete der Software werden nicht wie bei dem District-Layout als Plateaus dargestellt, sondern als Straßen. Das bedeutet das *root*-Verzeichnis wird als Hauptstraße abgebildet und alle Unterpakete zweigen von dieser Straße ab. Alle weiteren UnterPakete verzweigen sich rekursiv weiter und einzelne Klassen werden am Ende einer Straße positioniert. Neue Klassen werden dadurch weitgehend am Rand der Stadt platziert, wodurch sich die Stadt mit der fortschreitender Zeit der Entwicklung nach außen hin ausbreitet.

Die Interaktion in SoftVis3D ist intuitiv gestaltet. Mit der Maus kann stufenlos in die Stadt hinein gezoomt und die Perspektive verändert werden. In Abbildung 6 kann eine Detailansicht vom District-Layout und vom Evostreet-Layout betrachtet werden.

Alle Artefakte der Stadt können ausgewählt werden (in Abbildung 5 und 6 gelb markiert), woraufhin in einem Menu alle Kinder gelistet werden können. Zusätzlich kann aus dem Plugin heraus direkt in den Source-Code einer ausgewählten Klasse gesprungen werden.

Ein weiterer Unterschied zur CodeCity ist die Darstellung der Abhängigkeiten in SoftVis3D. Die aktuellen Version des SonarQube Plugins unterstützt die Darstellung der Abhängigkeiten zwar nicht, jedoch hat Rinderle 2016 in [12] dazu ein Konzept vorgestellt indem die Abhängigkeiten einer Software in einer Software-Stadt mit mehreren Ebenen visualisiert werden. Dazu werden die Abhängigkeiten aggregiert.

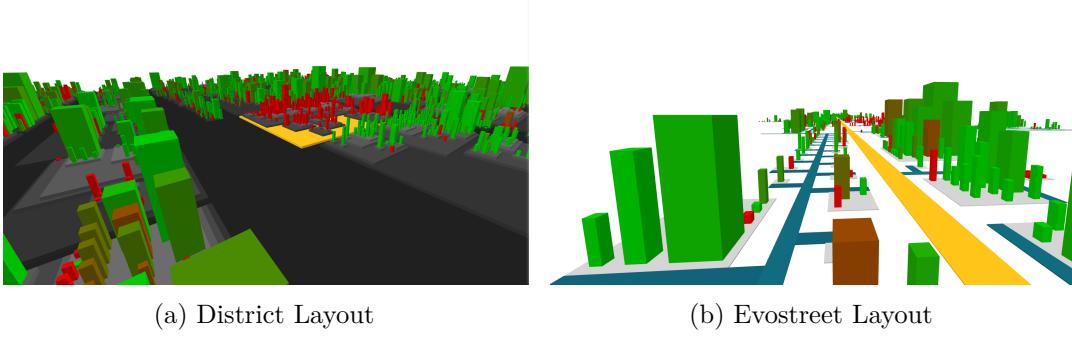


Abbildung 6: Unterstützte Layouts von SoftVis3D

Aggregation von Abhängigkeiten lässt sich allgemein folgendermaßen definieren:

**Definition 3.1** Bei der Aggregation von Abhängigkeiten zwischen Klassen werden die Abhängigkeiten nicht als direkte Verbindungen dargestellt, sondern über die Eltern-Pakete geleitet. Eine Klasse  $A \in x$  (im Paket  $x$ ) besitzt eine Abhängigkeit zu Klasse  $B \in y$ . Die Abhängigkeit von  $A$  geht zu einem zusätzlich Konstrukt  $\Psi_x$ , das die Außensicht des Pakets  $x$  repräsentiert. Angenommen  $x$  und  $y$  befinden sich im Paket  $z$ , dann geht die aggregierte Abhängigkeit entweder direkt von  $\Psi_x$  zu  $\Psi_y$ , oder geht über  $\Psi_z$  zu  $\Psi_y$  und die Hierarchie weiter nach unten zu  $B_y$ .

In SoftVis3D werden Abhängigkeiten zwischen Klassen innerhalb eines Pakets werden nicht aggregiert, sondern direkt als Pfeile angezeigt. Abhängigkeiten über Paketgrenzen hinweg werden aggregiert angezeigt und  $\Psi$  wird als zusätzliches Gebäude realisiert, wie es in Abbildung 7a zu sehen ist. Für jede Hierarchie-Ebene wird eine neue Ebene verwendet. Die zusätzlichen Gebäude enden in der jeweils darüber liegenden Ebene in einem Plateau, das die Größe des Pakets darunter widerspiegelt. Die aggregierten Abhängigkeiten werden dann zwischen den Plateaus dargestellt.

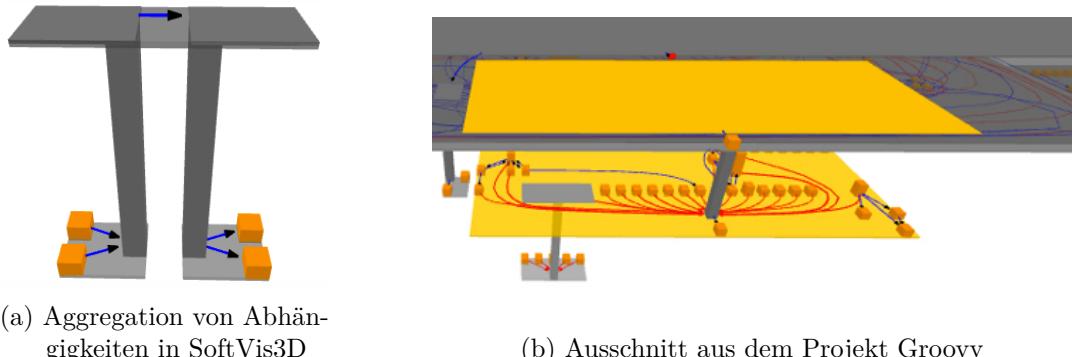


Abbildung 7: Abhängigkeiten in SoftVis3D [12]

## Vorteile

Bei der Visualisierung verschiedener **statischen Metriken von Klassen** kann SoftVis3D punkten. Die ganze Vielfalt an Metriken, die SonarQube unterstützt, kann auch in SoftVis3D Verwendung finden. SoftVis3D unterstützt für die **Struktur** der Software sogar verschiedene Layouts, wodurch das am besten für den jeweiligen Use-Case passende Layout gewählt werden kann. Wie auch bei der CodeCity ist die **Habitability** durch die Stadt-Metapher sehr gut und der **Drilldown** ist durch die intuitive Interaktion bei SoftVis3D noch besser.

## Nachteile

Die Darstellung der Abhängigkeiten beeinflusst die Darstellung der Struktur der Software, da zusätzliche Hilfsgebäude und mehrere Ebenen gebraucht werden. Die Übersichtlichkeit des 3D-Modells wird dadurch stark negativ beeinflusst.

Wie auch bei der CodeCity sind **Dynamische** Metriken nicht Teil des Modells.

Auch wenn mit dem Evostreet-Algorithmus ein Layout für eine Visualisierung der **Evolution** gegeben wäre, gibt es in SoftVis3D keine Möglichkeit diese auf einen Blick darzustellen.

Da SoftVis3D für den Browser konzipiert ist, wurde das Plugin **technologisch** auf *WebGL*-Basis mit *three.js*<sup>6</sup> entwickelt. WebGL wird jedoch nicht nativ auf der Microsoft HoloLens unterstützt. Ergo ist eine Verwendung oder Erweiterung von SoftVis3D für die Verwendung auf der Microsoft HoloLens nicht ohne Weiteres möglich. Ob das Rendern des SoftVis3D-Modells überhaupt mit tragbarem Aufwand für die Microsoft HoloLens verwendbar ist, soll teil der technischen Machbarkeit in Kapitel 5 sein.

## 4 Alternative Ansätze

Es wurden bisher zwei Umsetzungen der Stadt-Metapher behandelt. In diesem Abschnitt der Arbeit soll durch die kreative Entwicklung eigener Metaphern zwei Alternativen beleuchtet werden und ebenfalls anhand der in Abschnitt 2.2 aufgestellten Kriterien bewertet werden.

### 4.1 CodeForest

Die Softwarevisualisierung beschäftigt sich im Allgemeinen meist mit der Darstellung von hierarchischen Informationen. “Hierarchies are almost ubiquitous [...]”<sup>7</sup> [13], stellten Robertson et al. schon 1991 bei der Visualisierung von hierarchischen Informationen fest.

Die Struktur einer Software mit geschachtelten Paketen entspricht immer der einer Baumstruktur. In der Informatik weit verbreitet, ist diese Struktur auch nichts An-

---

<sup>6</sup> JavaScript 3D Library <https://threejs.org/>

<sup>7</sup> dt.: „Hierarchien sind fast allgegenwärtig [...]“

deres als eine Metapher. Die Ursprung entstammt einer Realität, die sich jeder gut vorstellen kann – eines Baumes. Zwar wird die Baumstruktur in der Informatik schon seit jeher zweidimensional verwendet, jedoch wäre eine dreidimensionale Visualisierung, der eigentlichen Metapher folgend, nur logisch. Wieso also nicht Software als dreidimensionale Bäume visualisieren?

Diesem Ansatz nachgehend, entsteht das Modell des *CodeForest*. Die Bäume werden in „Wuchsrichtung“ auf einer Ebene dargestellt. Die Analogie in diesem Modell ist simpel. Einzelne Bäume stellen Pakete im root-Verzeichnis der Software da, Verzweigungen Unterpakete und die Blätter der Bäume stellen die eigentlichen Dateien der Software dar. Die darzustellenden Metriken können dann als „Lauffärbung“ angewandt werden.

Es können einzelne Pakete genauer betrachtet werden, indem ein spezielles Paket das neue root-Verzeichnis wird und alle Unterpakete als neue Bäume auf der Ebene stehen.

Die Evolution einer Software ist durch das Wachstum des Waldes möglich zu visualisieren.

Besonders interessant ist die Visualisierung der Abhängigkeiten. In zwei Dimensionen sind Verbindungen zwischen Wurzel-Elementen nicht oder bestenfalls schlecht darstellbar. In drei Dimensionen gewinnt die Metapher von Bäumen an einem neuen Element – dem „Wurzelgeflecht“. Was im Zweidimensionalen hinter einer einzelnen Verbindung zwischen den Wurzel-Elementen verschwinden würde, spannt sich nun mit dem Wurzelgeflecht auf dem Waldboden auf.

Ähnlich wie bei dem Ansatz der Abhängigkeiten in SoftVis3D können Abhängigkeiten zwischen Paketen aggregiert werden und dann zwischen den Bäumen als Wurzeln visualisiert werden. Ungewünschte Abhängigkeiten, wie zum Beispiel zyklische, können direkt in den Bäumen farbig in der entsprechenden „Verästelung“ hervorgehoben werden.

Zu dieser reduzierbaren, aber übersichtlichen Darstellung der Abhängigkeiten sind noch zwei weitere Darstellungsformen denkbar. Zum einen können alle Abhängigkeiten von einem ausgewählten Blatt den Stamm hinunter über die Wurzeln, den Astverlauf hinauf bis hin zum Ziel-Blatt dargestellt werden, wodurch eine genauere Untersuchung der Abhängigkeiten möglich ist. Zum andern wäre für eine Gesamtdarstellung auch eine direkte Verbindung der Blätter durch „Spinnweben“ zwischen den Bäumen denkbar.

In Abbildung 8 ist beispielhaft ein solcher CodeForest illustriert. Es ist das Paket `lib` im root-Verzeichnis ausgewählt. Es werden die aggregierten Abhängigkeiten zu anderen Bäumen dargestellt und eine ungewünschte Abhängigkeit innerhalb des ausgewählten Pakets existiert.

## Nachteile

Gegenüber der Stadt-Metapher hat die Wald-Metapher den Nachteil, dass lediglich mit der Farbe der Blätter und damit nur eine **statische Metrik** gleichzeitig darstellbar ist. Bei der Stadt-Metapher sind es mit Grundfläche, Höhe und Farbe der Gebäude 3

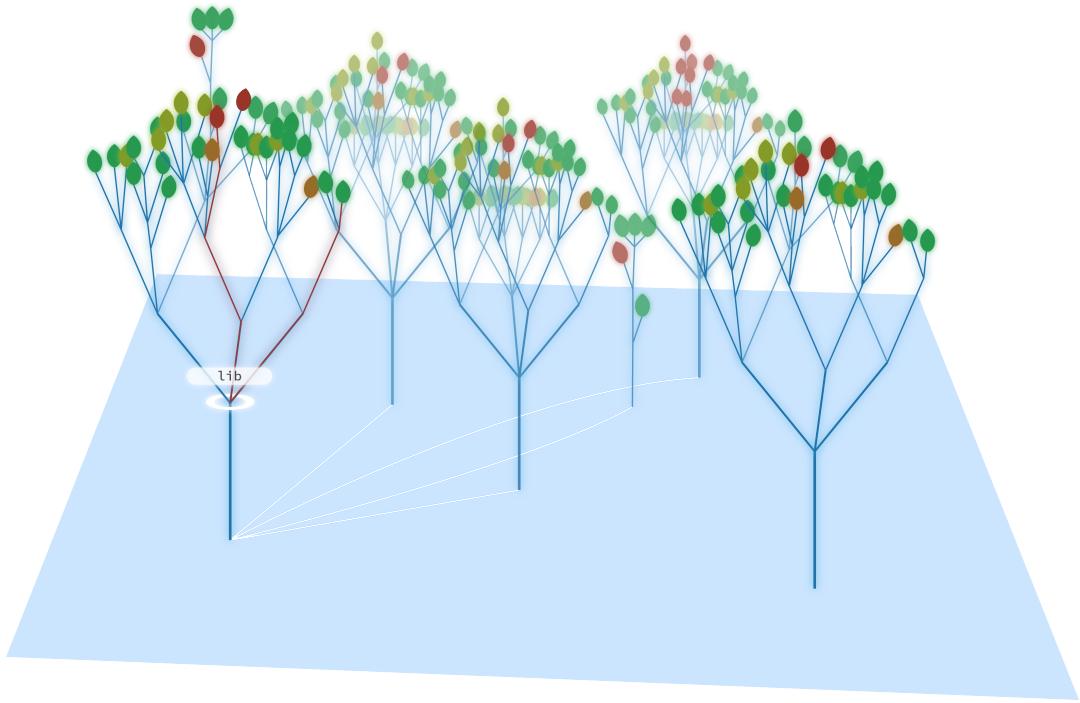


Abbildung 8: Beispielhafter Entwurf eines CodeForest

Metriken gleichzeitig.

### Vorteile

Die **Struktur** der Software wird durch Eins-zu-eins-Umsetzung auf die Bäume sehr gut wiedergespiegelt. **Abhängigkeiten** können durch das Wurzelgeflecht auf verschiedener Schachtelungstiefe, farbige Verästelung oder alternativen Spinnweben sehr interaktiv dargestellt werden.

Die **Evolution** der betrachteten Software integriert sich mit dem Wachstum eines Waldes nahtlos in die Wald-Metapher.

Die Darstellung der **Dynamik** bzw. Aufrufe können mithilfe von Verbindung, wie sie bei den Abhängigkeiten zum Einsatz kommen, dargestellt werden. Zudem könnte die Stamm- bzw. Ast-Dicke ein Indikator für die Anzahl der Aufrufe sein. Die Laufzeiten kann sich in der Farbe des Laubwerks wieder spiegeln.

Durch die Struktur der Bäume ist die Habitability gut umgesetzt. Ein Entwickler, der mit der Code-Base vertraut ist, wird sich im CodeForest schnell zurecht finden können. Jedoch können Bäume weniger markant eingeschätzt werden, als Stadt-Vierteln mit markanten Gebäuden.

Der Drilldown ist durch die „Verschiebung des Waldbodens nach oben“ interaktiv gestaltbar.

## 4.2 CodeUniverse

Bei der Suche nach einer weiteren eventuell geeigneten Metapher für die Darstellung von Software stellte sich die Frage, mit welchem Modell sich potentiell sehr viele Artefakte darstellen lassen können. Die Idee ein Universum als Softwarevisualisierung zu verwenden, lag dabei nicht fern. Im Folgendem wird deshalb das Modell *CodeUniverse* vorgestellt.

Tatsächlich stellte sich nach einer Recherche heraus, dass sich in [2] aus 2004 von Balzer et al. ein ähnlicher Ansatz finden lässt. Hier wird zwar nicht von einem Universum gesprochen, aber von geschachtelten Sphären: “The hierarchy of packages, which can be arbitrarily deep, is represented by nested spheres. The outermost sphere stands for the root of the package hierarchy.”<sup>8</sup> [2] Genau das ist das Grundprinzip. Bei Balzer et al. sind die Sphären von außen undurchsichtig. Erst beim Zoom in eine Sphäre wird die Hülle durchsichtig und offenbart weitere Sphären. Für einen Driltdown wäre das eine gute Alternative, für die Metapher des Universums und der Übersicht über eine Software, wäre es aber besser alle Artefakte der Software als sichtbare Sterne darzustellen. Die Sphären können aber dennoch für die Zugehörigkeit zu den Paketen um die Sterne als „Nebel dargestellt werden“.

Die Zugehörigkeit zu einem Paket wird aber auch über die Distanz der Sterne zueinander deutlich. Die Sterne des Pakets mit der tiefsten Hierarchie liegen am nächsten zusammen. Je näher am root-Verzeichnis, desto weiter dehnt sich das Universum aus, um innerhalb eines Pakets zwischen den Unterpaketen bzw. einzelnen Klassen gleich viel Abstand zu haben.

Die Größe und Farbe der Sterne – von „Weißen Zwergen“ bis hin zu „Roten Riesen“ – lässt sich jeweils einer Metrik zuordnen. Beispielsweise stechen dann besonders große, in entsprechender Farbe leuchtende Sterne gut ins Auge.

Abhängigkeiten können als direkte Verbindungen dargestellt werden, oder wiederum aggregiert (vgl. Definition 3.1) werden. Dabei würden Verbindungen zwischen den Sphären der einzelnen Pakete entstehen.

In Abbildung 9 ist ein beispielhafter Entwurf einer kleinen CodeUniverse abgebildet. Es ist das root-Verzeichnis mit zwei einzelnen Klassen und vier Unterpaketen dargestellt. In diesem Beispiel wurde für Abhängigkeiten direkte Verbindung gewählt und eine unerwünschte Abhängigkeit farbig hervorgehoben.

### Vorteile

Im Vergleich zum CodeForest, kann das CodeUniverse gleichzeitig eine **statische Metrik** mehr anzeigen.

Die **Evolution** ist in einem CodeUniverse durchaus gut darstellbar. Das Wachstum der Software könnte als „Ausdehnung“ des Universums visualisiert werden.

---

<sup>8</sup>dt.: Die Hierarchie der Pakete, die beliebig tief sein kann, wird durch geschachtelte Sphären repräsentiert. Die äußerste Sphäre entspricht dem root-Verzeichnis der Paket-Hierarchie.

Im Vergleich zur CodeCity wären **Abhängigkeiten** als direkte Verbindungen übersichtlicher, da in einer CodeCity Verbindungen nur überhalb der Stadt sein können. Im CodeUnivers können Verbindungen in alle Richtungen gehen. Weiterhin sind Abhängigkeiten, die hierarchisch nur nach oben bzw. unten gehen, d.h. keine Nachbarpakete beinhalten, aufgrund des ausdehnenden Anordnung der Sterne mit wenig Überschneidungen darstellbar.

Der **Drilldown** ist mit „Öffnen neuer Galaxien“ sehr gut und auch die **Habitability** ist mit der unterschiedlichen Ausdehnung der Galaxien gut.

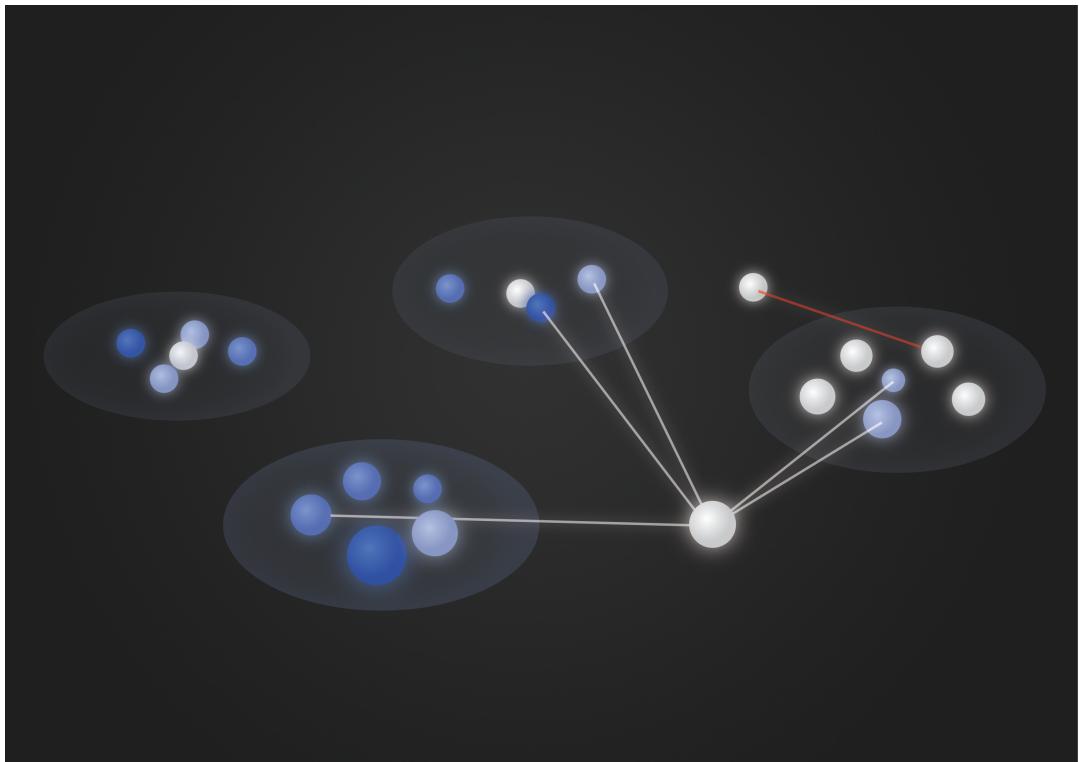


Abbildung 9: Beispielhafter Entwurf eines CodeUniverse

### Nachteile

Bei der Visualisierung der **Struktur** einer Software mit dem Modell des CodeUniverse tritt vor allem folgendes Problem auf:

Bei der Abgrenzung der Pakete voneinander mit durchscheinenden, umschließendem Sphären wird von außen betrachtet bei zunehmender Tiefe der Hierarchie zwangsläufig die Blickdichte immer höher. Bei großen Hierarchien kann dies dazu führen, dass die einzelnen Sterne in besonders tiefer Hierarchie nicht mehr gut zu erkennen sind, was sich negativ auf die Übersicht im Bezug auf Größe und Farbe auswirkt. Eine

Alternative wäre auf die Sphären zu verzichten. Ob dann die Struktur der Software noch gut zu erkennen ist, bleibt fraglich.

Die Visualisierung der **Abhängigkeiten** als direkte Verbindung würde bei großen Software-Systemen, ähnlich wie bei der CodeCity (vgl. Nachteile in 3.1), schnell unübersichtlich werden. Die aggregierten Abhängigkeiten setzen aber umschließende Sphären, oder ein anderes Konstrukt für  $\Psi$  voraus. Würde ein Punkt als  $\Psi$  verwendet werden, entstünde ein Bild, ähnlich wie das, das in Abbildung 10. Dies ist ein Ausschnitt aus der Visualisierungssoftware namens *Gource* und kommt gut mit zwei Dimensionen aus.

Ein Vorteil würde nur entstehen, wenn die Verbindungen in die dritte Dimension gezogen würden. Mit etwas Vorstellungskraft lässt sich feststellen, dass so ein Baum entstehen würde. Damit wären wir aber wieder beim CodeForest angelangt und die Metapher eines Universums wäre auch sehr weit hergeholt. Abhängigkeiten können also nur sinnvoll als direkte Verbindung realisiert werden.

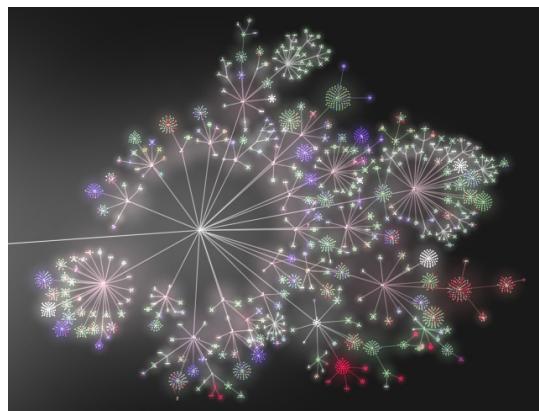


Abbildung 10: Visualisierung von Versionskontrolle mit Gource<sup>9</sup>

## 5 Technische Machbarkeit für die Microsoft HoloLens

Es wurde bereits in Abschnitt 3.1 erörtert, dass sich die CodeCity technologisch nicht eignet auf die HoloLens zu portieren. Im Folgenden soll erprobt werden, ob die bisherige Implementierung von SoftVis3D technisch sinnvoll und mit tragbarem Aufwand auf die HoloLens gebracht und erweitert werden kann.

Die JavaScript Graphik-Bibliothek WebGL wird in *Holographic Windows* von Windows stand 2017 nicht offiziell unterstützt. Es gibt jedoch das open-source Repository *HoloJS* von Microsoft, dass einen experimentellen Support verspricht. Dieses Projekt wandelt mithilfe von C++ jegliche WebGL- in native *DirectX*-Aufrufe<sup>10</sup> um.

<sup>9</sup>Quelle: [http://static.oschina.net/uploads/img/201311/15091800\\_qVuS.jpg](http://static.oschina.net/uploads/img/201311/15091800_qVuS.jpg)

<sup>10</sup>„DirectX ist ein Satz von Komponenten in Windows, mit dessen Hilfe Software, und zwar in erster Linie und insbesondere Spiele, Ihre Video- und Audiohardware direkt verwenden kann. Spiele, die

Nach anfänglichen Schwierigkeiten und Rücksprache mit einem Microsoft-Entwickler, konnte erfolgreich eine erste Beispiel-App mit three.js auf die HoloLens deployed werden.

Leider erkennet die *Live Preview* oder auch Screenshots der HoloLens die three.js-Objekte nicht und deshalb kann die Beispiel-App in dieser Arbeit nicht veranschaulicht werden.

Was bei der Ausführung der Beispiel-App sofort ins Auge sticht, ist, dass der Cursor der HoloLens, ein Punkt im Raum, der der Bewegung des Kopfes folgt oder mit einer verbundenen Zeigergerät bewegt werden kann, doppelt dargestellt wird. Eine sinnvolle Interaktion mit einer three.js-App ist somit nicht ohne Bugfixes möglich.

Nach Aussage von des Microsoft-Entwicklers ist eine aktive Weiterentwicklung des Projekts nicht geplant. Zwar wären Contributions zum HoloJS-Repository möglich, in welchem Umfang dies nötig wäre, um mit three.js-Objekten interagieren zu können, ist jedoch nicht absehbar.

Die Voraussetzung für die Portierung des Frontends von SoftVis3D auf die HoloLens kann deshalb nicht ohne erheblichen und nicht genau kalkulierbaren Aufwand gewährleistet werden.

Absatz zu dem Standard-Entwicklungs-Workflow mit Unity und VisualStudio

## 6 Fazit

### 6.1 Vergleich der behandelten Ansätze

Als Fazit soll eine Empfehlung für die Entwicklung einer Softwarevisualisierung für die HoloLens gegeben werden.

Die CodeCity von Wettel und Lanza stellt erstmals die Stadt-Metapher vor und ist als Konzept sehr durchdacht. Die Verwendung der Implementierung der CodeCity für die HoloLens ist aufgrund der veralteten Technologien jedoch ausgeschlossen. Darüber hinaus hat das Modell auch Schwächen, wie beispielsweise die wenig interaktiv gestaltete und eher unübersichtliche Darstellung der Abhängigkeiten.

SoftVis3D ist als weitere Implementierung der Stadt-Metapher sehr ausgereift und im Gegensatz zu CodeCity mit modernen Web-Technologien gebaut. Die statischen Metriken von Software-Artefakten können hervorragend analysiert werden. Für den Einsatz auf der HoloLens ist SoftVis3D jedoch, wie in 5 erläutert, auch nicht geeignet. Die Unterstützung von WebGL ist von Microsoft Stand heute nicht ausreichend gegeben. Das Konzept der Darstellung der Abhängigkeiten in SoftVis3D ist sogar noch unübersichtlicher, als in der CodeCity.

Bei der Betrachtung der zwei vorgestellten Modelle der Stadt-Metapher wäre es natürlich möglich eine neue Implementierung mit den Vorteilen aus beiden Modellen speziell für die HoloLens zu entwickeln und um fehlende Metriken erweitern. Das

---

DirectX nutzen, können in Ihre Hardware integrierte Funktionen zur Multimediasbeschleunigung effizienter nutzen, wodurch das Multimediaerlebnis insgesamt verbessert wird.“ [9]

würde jedoch die wenig attraktive Darstellung der Abhängigkeiten nicht lösen. Zudem wäre sehr viel Aufwand nötig, bis eine neue Softwarevisualisierung überhaupt die Fähigkeiten von SoftVis3D unterstützen würde.

Das CodeUnivers ist ein interessanter und intuitiver Ansatz. Die Struktur ist bei genauerer Betrachtung jedoch schwierig zu realisieren und letztendlich gelangt man bei der Einführung von Abhängigkeiten in das Modell zu einem ähnlichen Konstrukt wie das Modell des CodeForest.

Der CodeForest kann zwar nur eine statische Metrik gleichzeitig anzeigen und lässt deswegen den Zusammenhang zweier Metriken nicht auf einen Blick erkennen, ist aber in allen anderen Bereichen besser bzw. zumindest gleichwertig wie die Alternativen zu bewerten.

Tabelle 4: Vergleich der behandelten Ansätze

	<b>CodeCity</b>	<b>SoftVis3D</b>	<b>CodeForest</b>	<b>CodeUniverse</b>
Statische Metriken	drei gleichzeitig, kaum konfigurierbar	drei gleichzeitig, sehr flexibel	nur eine gleichzeitig	zwei gleichzeitig
Struktur	übersichtlich	unterschiedliche Layouts unterstützt	durch Baumstruktur sehr gut	Paket-Zughörigkeit weniger eindeutig
Abhängigkeiten	wenig interaktiv, unübersichtlich	sehr unübersichtlich	sehr gut darstellbar	Aggregation schwierig
Dynamik	nein	nein	ja	ja
Evolution	sehr gut	nein	Wachstum der Bäume	Ausbreitung des Universums, Entstehung Sterne
Habitability	gut	gut	vertraute Struktur	weniger markant
Drilldown	nicht interaktiv	gut	gut	sehr gut

Technologie	nicht mehr produktiv einsetzbar	WebGL für HoloLens	neue Technologien einsetzbar	neue Technologien einsetzbar
-------------	---------------------------------	--------------------	------------------------------	------------------------------

Ein Überblick über die in dieser Arbeit aufgestellten Kategorien und die Bewertung der vorgestellten Modelle der dreidimensionalen Softwarevisualisierung darin, ist in Tabelle 4 zu sehen.

Das Modell des CodeForest ist vielversprechend. Es kann all die Metriken visualisieren, die bei der Umfrage in Abschnitt 2.1 als wichtig erachtet wurden. Ob Softwareentwickler oder Projektleiter – alle in der Umfrage vorkommenden Rollen könnten in einem CodeForest wichtige und wertvolle Informationen gewinnen.

## 6.2 Ausblick

In einer weiterführenden Arbeit ist es geplant einen CodeForest für die HoloLens zu entwickeln. Dabei kann auf den Standart-Workflow in der HoloLens-Entwicklung gesetzt werden.

Es ist ein Datenmodell zu entwerfen, dass statische, dynamische und Informationen zur Evolution einer Software unterstützt. Des Weiteren müssen geeignete Algorithmen für das Layout der dreidimensionalen Bäume gefunden werden.

Ein wichtiger Bestandteil einer weiterführenden Arbeit ist die Ausarbeitung des Interaktionskonzepts. Zum Beispiel die Realisierung eines verschiebbaren Waldbodens, wäre eine spannende Aufgabe. Vor allem aber muss das Interaktionsskonzept auch mehrere gleichzeitige Nutzer unterstützen. Durch die SharedView der HoloLens sollte das ein großer Vorteil gegenüber herkömmlichen Softwarevisualisierungen darstellen.

## Abbildungsverzeichnis

1	Verteilung der Interesse nach nach Kategorien . . . . .	5
2	Information Pyramids [1] . . . . .	9
3	CodeCity von ArgoUML [16] . . . . .	10
4	Vorteile einer CodeCity . . . . .	11
5	SoftVis3D . . . . .	13
6	Unterstützte Layouts von SoftVis3D . . . . .	14
7	Abhängigkeiten in SoftVis3D [12] . . . . .	14
8	Beispielhafter Entwurf eines CodeForest . . . . .	17
9	Beispielhafter Entwurf eines CodeUniverse . . . . .	19
10	Visualisierung von Versionskontrolle mit Gource . . . . .	20

## Literatur

- [1] Keith Andrews, Josef Wolte und Michael Pichler. "Information PyramidsTM: A new approach to visualizing large hierarchies". In: *Proceedings of the IEEE Visualization'97*. 1997, S. 49–52.
- [2] Michael Balzer u. a. "Software landscapes: Visualizing the structure of large software systems". In: *IEEE TCVG*. 2004.
- [3] G Campbell und Patroklos P Papapetrou. *SonarQube in action*. Manning Publications Co., 2013.
- [4] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Berlin Heidelberg, 2007. ISBN: 9783540465041.
- [5] Richard P Gabriel. *Patterns of software*. Bd. 62. Oxford University Press New York, 1996.
- [6] Nahum D. Gershon. "From perception to visualization". In: *Scientific Visualization: Advances and Challenges*. Academic Press, 1994.
- [7] Adele Goldberg und David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.
- [8] Samuel Merrill. *The moose book*. EP Dutton, 1916.
- [9] Microsoft. *Installieren der neuesten Version von DirectX*. 2017. URL: <https://support.microsoft.com/de-de/help/179113/how-to-install-the-latest-version-of-directx> (besucht am 10.05.2017).
- [10] Yvo D. Niedrich. "Integration neuer Paradigmen zur 3D-Softwarevisualisierung". Master-Thesis. Universität Augsburg Fakultät für Angewandte Informatik, 2016.
- [11] QAware GmbH. *IT-Probleme lösen. Digitale Zukunft gestalten*. 2017. URL: <http://www.qaware.de/leistung/#leistung-realisation> (besucht am 28.03.2017).

- [12] Stefan Rinderle. *Kontinuierliche Architekturanalyse: Softwarevisualisierung mit SonarQube in 3-D*. 2016. URL: <https://jaxenter.de/kontinuierliche-architekturanalyse-softwarevisualisierung-mit-sonarqube-in-3-d-32619> (besucht am 10.04.2017).
- [13] George G Robertson, Jock D Mackinlay und Stuart K Card. “Cone trees: animated 3D visualizations of hierarchical information”. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 1991, S. 189–194.
- [14] Frank Steinbrück. “Consistent Software Cities: supporting comprehension of evolving software systems”. Diss. Cottbus, Brandenburgische Technische Universität Cottbus, 2013.
- [15] R. Wettel und M. Lanza. “Program Comprehension through Software Habitability”. In: *15th IEEE International Conference on Program Comprehension (ICPC '07)*. 2007, S. 231–240. DOI: 10.1109/ICPC.2007.30.
- [16] R. Wettel und M. Lanza. “Visual Exploration of Large-Scale System Evolution”. In: *2008 15th Working Conference on Reverse Engineering*. 2008, S. 219–228. DOI: 10.1109/WCRE.2008.55.