Universität Leipzig
Wirtschaftswissenschaftliche Fakultät
Institut für Wirtschaftsinformatik
Professur Softwareentwicklung

Visualisierung von Anforderungsdokumenten

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science – Wirtschaftsinformatik

Betreuender Hochschullehrer:	Prof. Dr. Eisenecker
Bearbeiter:	Benjamin Gahnz
	Am Sonnenbrink 29
	38855 Wernigerode
	Matr.-Nr.: 3740481
	10. Semester
Eingereicht am:	12.11.2024

Abstract

Inhalt
Die Arbeit untersucht die Visualisierung von Anforderungsdokumenten. Das Softwarevisualisierungswerkzeug *Getaviz* wurde prototypisch erweitert, um *ReqIF*-Dateien zu verarbeiten und in einem 3D-Modell darzustellen. Die verwendete *City-Metapher* wurde um Aspekte der *nested Treemap* erweitert, um Struktur und Kontextbeziehungen eines Anforderungsdokuments zu modellieren. Die Konzepte der Lokalität und Habitabilität wurden auf den Anwendungsfall übertragen und sollen eine intuitive Navigation und Interaktion mit der Visualisierung ermöglichen.

Schlüsselwörter
ReqIF, Anforderungsdokument, Visualisierung, Softwarevisualisierung

Gliederung

Abbildungsverzeichnis

Abkürzungsverzeichnis

Bspw.	Beispielsweise
d.h.	Das heißt
DOM	Document Object Manager
ID	Identifikations Attribut
ISO	International Organization for Standardization
RE	Requirements Engineering
ReqIF	Requirements Interchange Format
REV	Revision
SV	Softwarevisualisierung
XML	Extensible Markup Language
z.B.	Zum Beispiel

1 Einleitung

1.1 Motivation

Software dient in fast allen Industriebereichen zunehmend als Grundlage zur Realisierung innovativer Funktionen und Services. Als Pfeiler des erfolgreichen Software-Engineerings gewinnt auch Requirements Engineering damit zunehmend an Bedeutung (Pohl, 2008). Traditionell werden die gewonnenen Anforderungen in diesem Feld vorrangig natürlichsprachlich festgehalten (Cooper et al., 2009). Eine Möglichkeit, diese zu dokumentieren und auszutauschen, ist das *Requirements Interchange Format* (*ReqIF)*. Hier werden Anforderungen, Attribute, Werte und Beziehungen in einer auf *XML-basierenden* Datei gespeichert. In den gängigen User Interfaces (UI) werden die Daten in tabellarischer Form dargestellt. So können die enthaltenen Informationen detailliert und vollständig präsentiert werden, jedoch leidet die Übersichtlichkeit. Dies führt dazu, dass es nicht trivial ist, einen Überblick über die Struktur des Dokuments und die Beziehungen der Anforderungen zu erlangen. Als erklärtes Ziel des Requirements Engineerings, Anforderungen in Zusammenarbeit mit Product Ownern, Usern, Developern und Testern zu erheben, zu dokumentieren und zu validieren, ist es unerlässlich, das gemeinsame Verständnis und die Zugänglichkeit für alle Stakeholder größtmöglich herzustellen. Die Visualisierung von Anforderungen kann hier einen wichtigen Beitrag zur Verbesserung der Informationswahrnehmung und des Informationsflusses liefern (Aurum & Wohlin, 2003). Zur Visualisierung soll das Toolset *Getaviz* verwendet werden. Bisher dient es der Erzeugung von grafischen Artefakten aus Quellcodes und unterstützt 2D- sowie 3D-Visualisierungen in Form der *Recursive Disk*- bzw. *City-Metapher* (Baum et al., 2017). Ursprünglich zur Visualisierung von Java-Quellcode entwickelt, werden stetig weitere Extraktoren hinzugefügt. Auch die Erweiterbarkeit um das *ReqIF-Format* wird angestrebt, da es sich durch seine Verwendung als Austauschformat und die daraus resultierende vielfältige Tool-Integration besonders eignet, um als Grundlage einer Visualisierung von Anforderungsdokumenten zu dienen (Ebert & Jastram, 2012). Es wird erhofft, so einen Beitrag zur Beantwortung der Frage zu liefern, ob und inwieweit metapherbasierte Modellierungen auf Anforderungsdokumente anwendbar sind und ob Werkzeuge der Softwarevisualisierung dafür genutzt werden können.

1.2 Zielstellung

Die Arbeit untersucht, wie *Getaviz* genutzt werden kann, um dem ReqIF-Standard entsprechende Anforderungsdokumente zu visualisieren. Es wird erhofft, so einen Beitrag zur Beantwortung der Frage zu liefern, ob und inwieweit metapherbasierte Modellierungen auf Anforderungsdokumente anwendbar sind und ob Werkzeuge der Softwarevisualisierung dafür genutzt werden können.

Zur Bearbeitung wird die Forschungsmethode des Prototypings (vgl.[Wilde & Hess, 2007]) verwendet. Zunächst werden die dazu notwendigen Grundlagen erarbeitet. Dazu zählt die Identifikation der zu visualisierenden Aspekte der Anforderungsdokumente sowie ihre Repräsentation im *ReqIF-Standard*. Anschließend werden verschiedene Metaphern des Softwarevisualisierungsgenerators *Getaviz* auf ihre Übertragbarkeit geprüft und dem Anwendungsfall angepasst. Der zum Erzeugen der Visualisierung notwendige Verarbeitungsprozess wird konzipiert. Die anschließende Umsetzung in Form der prototypischen Implementierung soll die Machbarkeit belegen. Eine abschließende Evaluation wird skizziert, kann aber im Rahmen dieser Arbeit nicht durchgeführt werden.

1.3 Aufbau

Nach der Einführung des Themas im ersten Kapitel werden im zweiten Kapitel die grundlegenden Konzepte der Arbeit definiert.

Danach erfolgt der Entwurf des Prototyps, der in die Softwarevisualisierung eingeordnet und dessen Anwendungsfall abgegrenzt wird. Darauf aufbauend werden die darzustellenden Aspekte der Anforderungsdokumente sowie deren Repräsentation im ReqIF-Standard vorgestellt. Anschließend werden verschiedene Metaphern zur Modellierung analysiert und ihre Eignung diskutiert. Schließlich werden die Extraktion der zu visualisierenden Elemente, das Datenbankmodell, die Verarbeitungsschritte zum 3D-Modell und die Darstellung im User Interface entworfen. Das vierte Kapitel behandelt die Umsetzung des Entwurfs als Prototyp. Abschließend werden die Resultate der Arbeit zusammengefasst, diskutiert und ein Ausblick auf zukünftige Schritte gegeben.

1.4 Literaturrecherche

Als Grundlage für die Einordnung in das Requirements Engineering dient Pohls Buch *Requirements Engineering: Grundlagen, Prinzipien, Techniken* (Pohl, 2008). Hier wird ein umfangreicher Überblick zum Thema geschaffen. Im Kontext der Softwarevisualisierung wird auf den Arbeiten Diehls aufgebaut, insbesondere das Buch *Software Visualization:*

Visualizing the Structure, Behaviour, and Evolution of Software (Diehl, 2005). Dieses Werk legt die Grundlagen für die Softwarevisualisierung fest, einschließlich Definitionen, Klassifizierung der drei Hauptaspekte und verschiedener Visualisierungstechniken. Die Arbeit mit dem Toolset *Getaviz* stützt sich vor allem auf die Veröffentlichungen der Forschungsgruppe *Visual Software Analytics*. Diese bieten eine Übersicht über die Bestandteile des Toolsets, die verwendeten Visualisierungsmetaphern sowie die Erweiterung um *Neo4J*.

Die Nutzung von Visualisierungen im Anforderungsmanagement erfährt in der Forschung zunehmende Aufmerksamkeit. Die Studien *Requirements Engineering Visualization: A Survey on the State-of-the-Art* (Cooper et al., 2009) und *Requirements Engineering Visualization: A Systematic Literature Review* (Abad et al., 2016) ordnen die Ergebnisse des *I. – V. International Workshop on Requirements Engineering Visualization* (REV) ein und bieten einen fundierten Einstieg in das Thema. Sie klassifizieren Nutzungsmuster und Ziele der Visualisierung, ordnen verwendete Visualisierungsmethoden den Aktivitäten und Phasen des Requirements Engineering zu und identifizieren aktuelle Forschungsfelder.

Im Zuge der Literaturrecherche konnten keine weiteren metapherbasierten 3D-Modelle zur Visualisierung von Anforderungsdokumenten ausgemacht werden. Ebenfalls nicht gefunden werden konnten wissenschaftliche Artikel zur Anwendung von Werkzeugen der Softwarevisualisierung im Kontext der REV.

2 Grundlagen

2.1 Anforderungsmanagement

Nach Pohl ist das Requirements Engineering „ein inkrementeller, iterativer und kooperativer Prozess, dessen Ziel es ist, zu gewährleisten, dass alle relevanten Anforderungen bekannt und in dem erforderlichen Detaillierungsgrad verstanden sind, die involvierten Stakeholder eine ausreichende Übereinstimmung über die bekannten Anforderungen erzielen und alle Anforderungen konform zu den Dokumentationsvorschriften dokumentiert […] sind.“ (Pohl, 2008, S. 48ff) Um dies zu erreichen, werden die Kernaktivitäten der Dokumentation, Übereinstimmung und Gewinnung kontinuierlich durch die Querschnittsaufgaben der Validierung und des Managements von Aktivitäten, Anforderungsartefakten und des Systemkontextes unterstützt. Pohl kategorisiert Anforderungen nach funktionalen Anforderungen, die

bereitzustellende Funktionen eines Systems definieren, Qualitätsanforderungen, die qualitative Eigenschaften festhalten und Rahmenbedingungen, die die organisatorische oder technologische Art und Weise einschränken, wie ein Produkt entwickelt wird. (Pohl, 2008, S. 15 ff). Um diese zu dokumentieren, werden unterschiedliche Anforderungsartefakte erstellt. Das umfasst lösungsorientierte Anforderungen, Szenarien und Ziele. Beispiele für Anforderungsartefaktarten sind u.a. Use Cases/Stories, ER-, Sequenz- und Klassendiagramme oder Traceability-Tabellen. (Pohl, 2008, S. 15 ff) Als Endprodukt werden im deutschsprachigen Raum Lasten- und Pflichtenhefte gefordert, welche eine natürlichsprachlichen Dokumentation vorsehen. Dies ermöglicht ein detailliertes und allgemein verständliches Festhalten der Anforderungen, kann jedoch unübersichtlich und zu Teilen mehrdeutig sein. Durch das zusätzliche Erstellen von grafischen Modellen können komplexe Anforderungsdokumente zusammengefasst, detailliert, sowie Kontextbeziehungen zwischen Anforderungen, den Anforderungs- und Entwicklungsartefakten (z.B. Komponenten und Testfällen) intuitiv und strukturiert dargestellt werden (Pohl, 2008, S.299).

2.2 Requirements Interchange Format

Das *Requirements Interchange Format* ist ein *XML-basiertes* Austauschformat. Es wurde entwickelt, um Anforderungen und deren zugehörige Metadaten zwischen verschiedenen Software-Werkzeugen auszutauschen. Es wurde von der deutschen Automobilindustrie pioniert und wird mittlerweile von vielen weiteren führenden Unternehmen zum Austausch von Anforderungsdokumenten über Unternehmensgrenzen hinweg verwendet. Darüber hinaus ist es in viele der gängigen RE-Tools integriert, darunter *PTC Integrity*, *Requisis Rex*, *IBM Rational DOORS* und *IBM DOORS Next* (Ebert & Jastram, 2012). Als Grundlage dient eine nested-XML-Struktur, die sich in drei Kernelemente gliedert. Der <ReqIF-Header > enthält die Metadaten der ReqIF-Datei, <ReqIF-ToolExtension > enthält Inhalte für externe Werkzeuge und <ReqIF-Content> die Spezifikationen, Anforderungen, Attribute, Werte und Beziehungen (Object Management Group, 2016).

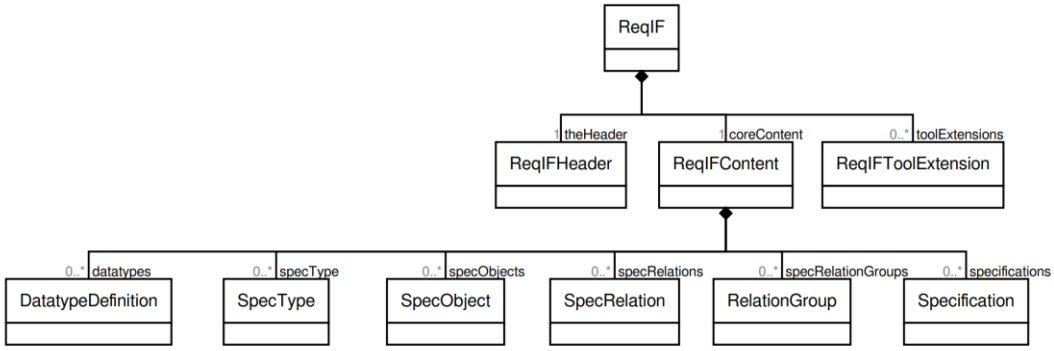

Abbildung 1 – ReqIF-Metamodell (vgl. (Michael Jastram))

2.3 Getaviz

Getaviz entstand an der Universität Leipzig aus der Arbeit der Forschungsgruppe *Visual Software Analytics*. Es dient der Erzeugung von Visualisierungen in 2D und 3D aus dem Quellcode objektorientierter oder prozeduraler Programmiersprachen. Als Toolset besteht es aus mehreren Werkzeugen. Zunächst werden sprachspezifische Extraktoren genutzt, um Softwareprojekte einzulesen und in *Neo4J* zu speichern. Anschließend entnimmt der Generator alle relevanten Informationen, um eine Visualisierung zu erzeugen. Das dritte Werkzeug ist ein *User Interface*, welches über den Browser Interaktionsmöglichkeiten mit den erzeugten Visualisierungen bereitstellt. Außerdem existiert ein Evaluierungsserver, der zur Evaluierung der Visualisierungen genutzt werden kann. Zum Zeitpunkt der Arbeit werden in der aktuellen Version von *Getaviz* die *Recursive-Disk* und verschiedene Abwandlungen der *City-Metaphern* unterstützt (Baum et al., 2017; Vogelsberg, 2018).

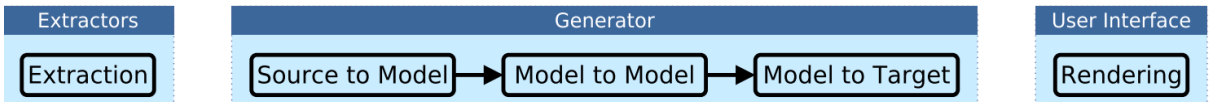

Abbildung 2 - Die Komponenten von *Getaviz* (vgl. (Baum et al., 2017))

3 Konzeption

Dieses Kapitel widmet sich dem Entwurf der Visualisierung und des dafür notwendigen Verarbeitungsprozesses. Der Entwurf beginnt mit der Beschreibung des spezifischen Anwendungsfalles, aus dem die Auswahl der zu visualisierenden ReqIF-Elemente abgeleitet wird. Im nächsten Schritt wird untersucht, welche Metaphern genutzt werden können, um diese *ReqIF-Elemente* anschaulich darzustellen. Dazu gehört auch das

Mapping der *ReqIF-Elemente* auf entsprechende Metapherelemente, um eine intuitive und nachvollziehbare Visualisierung zu gewährleisten.
Darauf folgt die Konzeption eines Datenmodells in einer Graphdatenbank, begleitet von einem Vergleich verschiedener Möglichkeiten zur Extraktion der relevanten Daten. Darauffolgend werden die notwendigen Anpassungen im Generierungsprozess des Prototyps beschrieben. Abschließend wird das *User Interface* sowie die Interaktionsmöglichkeiten mit dem 3D-Modell entworfen, um eine effiziente und benutzerfreundliche Nutzung zu ermöglichen.

3.1 Anwendungsfall

Ausgangspunkt ist zunächst die Einordnung des Prototyps in die Softwarevisualisierung. Die enge Definition nach Diehl, welche Softwarevisualisierung nur als „die Visualisierung von Algorithmen und Programmen“ (Diehl, 2005; S.3) beschreibt, würde dies nicht umfassen. Jedoch erlaubt die erweiterte Definition Diehls „die Visualisierung von Artefakten, die sich auf Software und deren Entwicklungsprozess beziehen“ (Diehl, 2005; S.3) die Einordnung in die Softwarevisualisierung. Die Hauptziele der Softwarevisualisierung sind gemäß Diehl die Erleichterung des Verständnisses von Software und die daraus resultierende Erhöhung der Produktivität im Softwareentwicklungsprozess (Diehl, 2005; S. 4). Es werden drei Arten der Softwarevisualisierung unterschieden: Strukturvisualisierung, Verhaltensvisualisierung und Entwicklungsvisualisierung. Strukturvisualisierung konzentriert sich auf die statischen Aspekte eines Softwaresystems, wie Architektur und Klassenhierarchie, um ein besseres Verständnis der Softwarestruktur zu ermöglichen. Verhaltensvisualisierung befasst sich mit den dynamischen Aspekten wie Interaktionen und Zustandsänderungen zur Laufzeit, um das Laufzeitverhalten zu analysieren. Entwicklungsvisualisierung fokussiert sich auf den Entwicklungsprozess und die Evolution des Systems, um den Fortschritt zu überwachen und die Zusammenarbeit im Team zu verbessern. Die Arbeit beschäftigt sich mit der Strukturvisualisierung, aber auch die Entwicklungsvisualisierung könnte in Zukunft von Interesse sein (vgl. (Diehl, 2005; S.4 ff.)).
Im zweiten Schritt wird der spezifische Anwendungsfall definiert, um so die Grundlage für die Entwicklung eines geeigneten Softwareprototyps zu schaffen (Pohl, 2008) Auch die Usability wie in *ISO 9241* definiert, wird maßgeblich durch die Eignung für den Benutzer und die Aufgabe bestimmt.

Zur Beschreibung eines Anwendungsfalls definiert Hundhausen vier Aspekte: (vgl. (Hundhausen, 1997; S.4)

Ziel der Aufgabe:

Ausgangspunkt der Arbeit ist die Komplexität von Anforderungsdokumenten und ihrer Beziehungen. Die tabellarische Darstellung, die das Ziel verfolgt, Anforderungen und ihre Attribute bearbeitbar zu machen, soll um eine Übersicht des gesamten Dokuments ergänzt werden. Daraus resultiert das Ziel, insbesondere die Struktur des Anforderungsdokuments und die Beziehungen zwischen den Anforderungen zu verstehen.

Der Ausgangspunkt der Arbeit liegt in der Komplexität von Anforderungsdokumenten durch die hohe Anzahl von Anforderungen und ihren Beziehungen. Die traditionelle tabellarische Darstellung verfolgt das Ziel, Anforderungen detailliert darzustellen und ihre Attribute bearbeitbar zu machen. Dies soll um eine visuelle Übersicht des gesamten Dokuments ergänzt werden. Daraus resultiert das Ziel, insbesondere die Struktur des Anforderungsdokuments und die Beziehungen zwischen den Anforderungen zu verstehen.

Gruppe von Menschen, die dieses Ziel verfolgt:

Die primären Adressaten sind Requirements Engineers, deren Aufgaben unter anderem das Erstellen, Verstehen und Bearbeiten von Anforderungsdokumenten umfassen. Weiterhin fungieren sie als Schnittstelle zwischen verschiedenen Stakeholdern, wie Auftraggebern und Entwicklerteams, um ein gemeinsames Verständnis der Anforderungen sicherzustellen(Pohl, 2008). Neben den Requirements Engineers gehören auch alle weiteren Stakeholder, die an der Entwicklung und dem Vertragsabschluss beteiligt sind, zu den potenziellen Nutzern der Visualisierung.

Softwarevisualisierungsartefakt das genutzt wird:

Getaviz soll zur Erzeugung einer Strukturvisualisation verwendet werden. Die Modellierung soll metapherbasiert erfolgen und im Rahmen eines User Interfaces nutzerfreundlich dargestellt werden.

Artefakt das visualisiert wird:

ReqIF wurde als Grundlage für den Visualisierungsprozess ausgewählt, da es als Austauschformat eine breite Tool-Integration besitzt. Zudem basiert ReqIF auf dem *XML-Format*, was die maschinelle Verarbeitung und Integration in bestehende Systeme erleichtert.

Zusammengefasst basiert der Anwendungsfall auf der Herausforderung, die Komplexität von Anforderungsdokumenten und somit die Ordnung und Beziehungen der

Anforderungen innerhalb dieser Dokumente verständlich darzustellen. Da die Nutzergruppe vielfältig ist und kein einheitliches Vorwissen vorausgesetzt werden kann, ist eine intuitive Visualisierung unerlässlich. Zum Erzeugen einer Strukturvisualisierung soll *Getaviz* um die Verarbeitung von *ReqIF-Dateien* erweitert werden.

3.2 Entwurf der Visualisierung

Aufbauend auf dem zuvor zusammengefassten Anwendungsfall soll nun die Visualisierung entworfen werden. Dazu werden zunächst die darzustellenden Elemente identifiziert. Anschließend werden verschiedene Metaphern vorgestellt und hinsichtlich ihrer Anwendbarkeit analysiert.

3.2.1 Identifikation der zu visualisierenden Elemente

Zur Strukturierung von natürlichsprachlichen Anforderungsdokumenten werden unterschiedliche Referenzmodelle verwendet, daher sollen die Struktur und Bestandteile hier auf den kleinsten gemeinsamen Nenner reduziert werden. Durch das Nutzen eines generischen Grundmodells für die Visualisierung können anschließend verschiedene Referenzmodelle visualisiert werden.

Betrachtet werden textuelle, attributisierte Anforderungsdokumente; weitere Formen der Spezifikation existieren (siehe Kapitel 2.1), werden in dieser Arbeit jedoch nicht berücksichtigt. Im Wesentlichen bestehen Anforderungsdokumente aus einer oder mehreren Spezifikationen. Diese umfassen, meist in Kapitel gegliederte, strukturierte Sammlungen von Anforderungsartefakten und Kontextbeziehungen. Zusätzlich beinhalten sie weiterführende Informationen, unter anderem Glossar, Änderungshistorie und Abkürzungsverzeichnis. Diese sollen nicht Bestandteil der Visualisierung sein.

Zum Verständnis des *ReqIF-Meta-Modells* werden zunächst die strukturgebenden Elemente identifiziert, anschließend wird die Referenzierung erklärt, da diese zur Datenverarbeitung aufgelöst werden muss. Im ReqIF-Standard werden einzelne Anforderungen durch <Spec-Objects> repräsentiert, die eine eindeutige ID und Referenzen zu Attributen sowie die Werte der Attribute umfassen. Der Tag <Spec-Type> definiert die verschiedenen SpecObject-Klassen und gibt vor, welche Attribute sie besitzen und von welchem Datentyp die Werte der Attribute sind. Die Datentypen werden im Tag <Datatypes> definiert und können einen Wertebereich besitzen. Als Datentypen können u. a. *String*, *Boolean*, *Integer*, *Enumeration*, *Real*, *Simple*, *Date* und *XHTML* verwendet werden. <Specification> repräsentieren Spezifikationen und können durch Attribute und

Werte näher beschrieben werden. Ebenfalls umfassen sie das Element <Spec-Hierarchy> dies wird verwendet, um <Spec-Objects> über Eltern-Kind-Beziehungen in Form einer geordneten Hierarchie aufzulisten.

Kontextbeziehungen zwischen den Anforderungen werden durch <SpecRelations> definiert und können in <SpecRelationGroups> gruppiert werden. Der *ReqIF-Standard* verwendet die Referenzierung über IDs zum Verknüpfen der Elemente über mehrere Ebenen hinweg, um eine konsistente Datenstruktur zu schaffen und so einen möglichst verlustfreien Datenaustausch zu ermöglichen. Diese müssen vom Parser aufgelöst werden und sind in der Abbildung 3 vereinfacht dargestellt (vgl. (Object Management Group, 2016)).

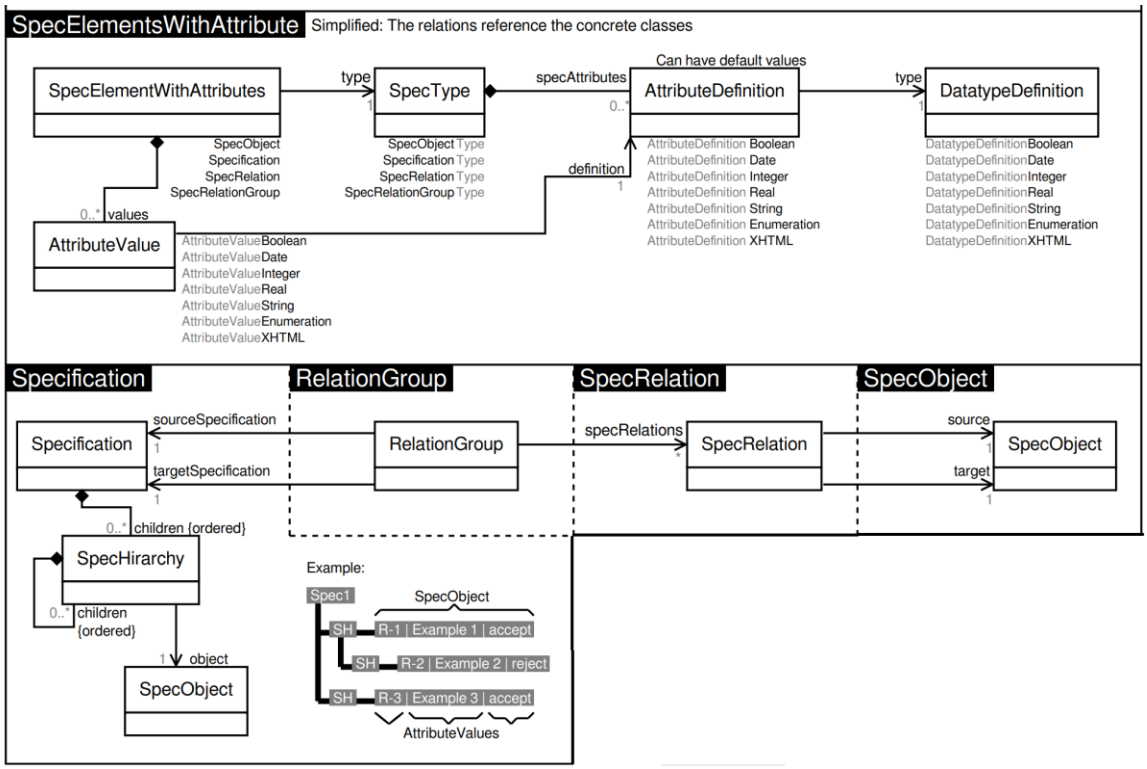

Abbildung 3 - Referenzierungen im ReqIF-Standard (vgl. (Jastram))

Als problematisch bei der Analyse des *ReqIF-Standards* erwies sich die Identifikation von Kapiteln und Anforderungen. Im Rahmen der Recherche wurden mindestens drei verschiedene Unterscheidungsmöglichkeiten identifiziert, weitere unternehmensspezifische Umsetzungen sind anzunehmen. Eine Möglichkeit besteht darin, separate <Spec-Types> zu erstellen: einen oder mehrere für Anforderungen und einen für Abschnitte.

Eine weitere Methode besteht darin, zu überprüfen, ob das Attribut *ReqIF.ChapterName* im <Spec-Objekt> vorhanden ist. Wenn es vorhanden ist, handelt es sich um einen Abschnitt; wenn nicht, um eine Anforderung.

Eine alternative Herangehensweise ist die Verwendung eines einzigen <Spec-Type>, der ein Type-Attribut bereitstellt, um zwischen Anforderung und Abschnitt zu unterscheiden. Auch die notwendige Existenz von Attributen ist im ReqIF-Standard nicht näher spezifiziert (vgl. (Object Management Group, 2016)).
Als Grundlage der Arbeit werden daher die Empfehlungen des *prostep ivip Recommendations* vorausgesetzt (prostep ivip, 2024). Dieser Leitfaden bietet praktische Empfehlungen zur Gestaltung von ReqIF-kompatiblen Werkzeugen und zielt darauf ab, die Interoperabilität zwischen verschiedenen ReqIF-Implementierungen zu verbessern. Die Empfehlungen werden in Zusammenarbeit mit den größten Tool-Anbietern sowie Vertretern von Unternehmen, welche den ReqIF-Standard verwenden, erstellt. In der Arbeit wird die Existenz der Attribute *ReqIF-WF.Type, ReqIF.ForeignID* und *ReqIF.ChapterName* vorausgesetzt. *ReqIF-WF.Type* kann die Werte Requirements, Chapter und Information besitzen. So können Anforderungen, Kapitel und zusätzliche Informationen unterschieden werden. Eine für den Menschen lesbare ID bietet das Attribut ReqIF.ForeignID. Das Attribut *ReqIF.ChapterName* wird zur Namensgebung verwendet, falls es sich bei dem SpecObject um ein Kapitel handelt (prostep ivip, 2024).

3.2.2 Analyse der Metaphern

Im Folgenden sollen die zu evaluierenden Metaphern vorgestellt und entsprechende Mappings der Anforderungsdokumentbestandteile auf die Metapherelemente vorgeschlagen werden.

Zur Strukturvisualisierung bietet *Getaviz* zwei Grundmetaphern an: die *Recursive-Disk-Metapher* sowie die *City*-Metapher. Es existieren weitere Abwandlungen der *City*-Metapher, jedoch konnte aufgrund des Umfangs der Arbeit eine Evaluation aller Metaphern nicht umgesetzt werden.
Die *Recursive-Disk*-Metapher ist eine 2D-Visualisierungsmethode, die verwendet wird, um strukturelle Aspekte und Beziehungen von Software darzustellen. Diese Methode nutzt verschachtelte kreisförmige Glyphen, d.h. graphische Darstellungen, um die Elemente eines Softwaresystems zu visualisieren. Dabei werden Klassen durch Ringe nach außen hin abgegrenzt. Innerhalb dieser Klassen-Ringe werden die dazugehörigen Klassenbestandteile wie innere Klassen ebenfalls als Teil-Ringe und Attribute und Methoden als Ringsegmente dargestellt.

Pakete wiederum können mehrere dieser Klassen oder Pakete beinhalten und umschließen diese durch einen weiteren Ring. Farben und Größen der Kreise können verwendet werden, um zusätzliche Informationen wie die Anzahl der Methoden oder die Komplexität einer Klasse darzustellen. Die Metapherelemente werden so angeordnet, dass die Glyphen mit der größten Fläche in der Mitte platziert werden; weitere Glyphen werden nach absteigender Fläche um die zuerst platzierte Glyphe positioniert. Beziehungen zwischen Klassen, wie Abhängigkeiten oder Vererbungen, werden durch das Ausblenden von nicht beteiligten Elementen veranschaulicht.

Das Besondere an der *RD-Metapher* ist, dass so alle Bestandteile von objektorientierten Programmiersprachen dargestellt werden können. Ziel ist es, Designfehler in der Software durch Muster in der Visualisierung erkennbar zu machen (Müller & Zeckzer, 2015).

Ein mögliches Mapping der Elemente eines Anforderungsdokuments auf die Metapherelemente ist es, Spezifikationen und Kapitel als Pakete darzustellen. Innerhalb dieser werden die dazugehörigen Anforderungsartefakte als Klassen mit den entsprechenden Attributen als Methoden dargestellt.

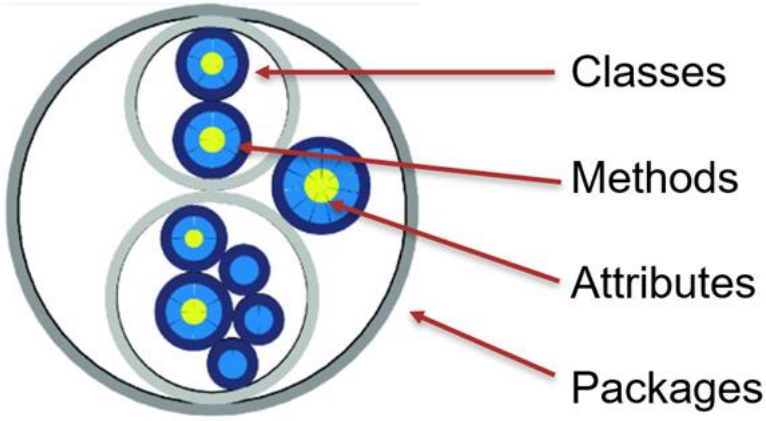

Abbildung 4 - RD-Metapher (in Anlehnung an (Baum et al., 2017))

Im Gegensatz zur abstrakten *RD-Metapher* referenziert die dreidimensionale *City*-Metapher die Echtwelt und ist in Struktur und Aussehen an eine Stadt angelehnt. Zentrales Element sind Gebäude, welche Klassen repräsentieren. Pakete werden durch Districts in Form von Bodenplatten dargestellt. Auf den Districts können rekursiv weitere Gebäude oder Districts positioniert werden. Attribute und Methoden bestimmen die Fläche bzw. die Höhe der Gebäude.

Die Positionierung der Gebäude und Districts, sowie die Fläche der Districts, wird in Form eines *Bin-Packaging-Problems* gelöst. Dabei werden die Elemente der Größe nach sortiert und in absteigender Ordnung möglichst platzsparend angeordnet. Beziehungen zwischen Klassen werden als verbindende Kanten zwischen Gebäuden dargestellt.
Ziel der *City-Metapher* ist es, das Verständnis von Softwaresystemen zu stärken, um die Einarbeitung in neue Software, den Vergleich verschiedener Programme, die Untersuchung von Auswirkungen von Änderungen und Re-Engineering-Aufgaben zu erleichtern.
Dafür sollen die Konzepte der Lokalität und Habitabilität angewendet werden. Lokalität bezieht sich dabei auf die Idee, Software als physischen Raum wahrzunehmen, in dem sich Anwender orientieren können. Sie wird umgesetzt, indem nur relevante Elemente visualisiert und klare Orientierungspunkte verwendet werden. Habitabilität ist eng verwandt mit Familiarität und beschreibt die Möglichkeit, sich intuitiv orientieren zu können. Sie soll durch eine vertraute Metapher sowie intuitive Navigations- und Interaktionsmöglichkeiten erreicht werden (Wettel et al., 2010).

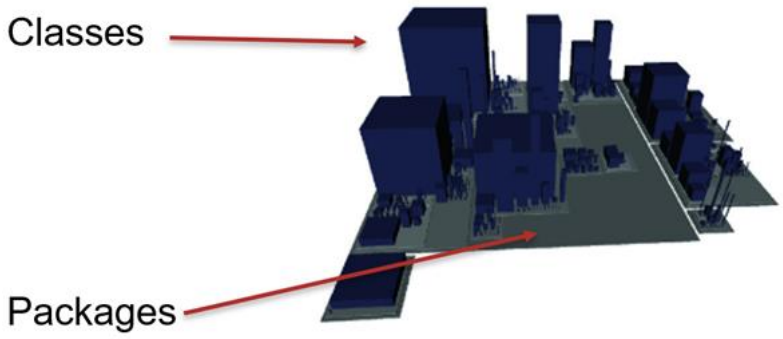

Abbildung 5 - City-Metapher (in Anlehnung an (Baum et al., 2017))

Für diese Arbeit wird vorgeschlagen, Spezifikationen und Kapitel als Bodenplatten darzustellen. Anforderungen sollen durch Gebäude repräsentiert werden, wobei die Höhe der Anzahl der Beziehungen und die Fläche der Anzahl von Attributen entspricht.

3.2.3 Diskussion der Metaphern

Zur Evaluation der vorgeschlagenen Mappings wurden Mockups erstellt. Dafür wurde ein Java-Programm geschrieben, welches in seiner Struktur und Bezeichnungen einer *Software Requirement Specification* entspricht (Engineering Standards Committee of the IEEE Computer Society, 2011). Somit konnte *Getaviz* zur Visualisierung genutzt werden.

Spezifikationen und Kapitel entsprachen Packages, Anforderungsartefakte wurden zu Klassen und Attribute zu Methoden.

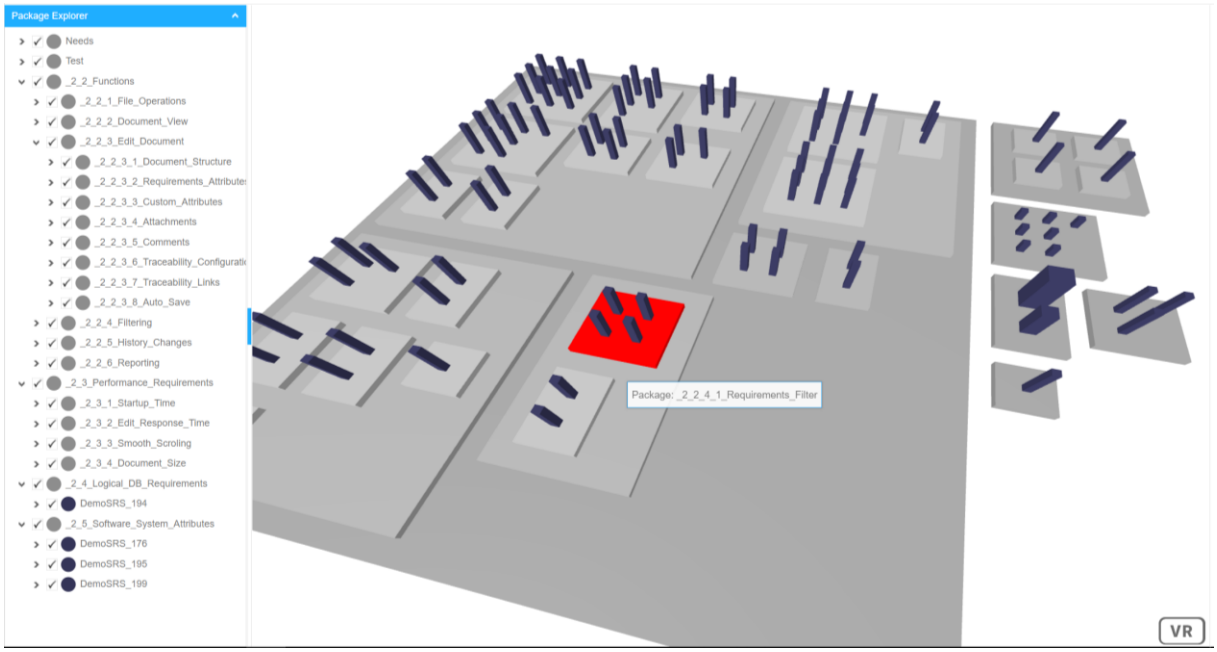

Abbildung 6 - Mockup der City-Metapher

Dabei wurde die *RD-Metapher* negativ bewertet. Die Ziele sind nicht auf den gegebenen Anwendungsfall übertragbar, da nicht die Anforderungsartefakte im Speziellen, sondern ihre Strukturierung im Anforderungsdokument als Fokus gesetzt werden. Das Layout lässt sich ebenfalls nur mit Einschränkungen anwenden, da es sich zur Darstellung von Containment-Beziehungen eignet, nicht aber für geordnete Hierarchien, da klare Bezugspunkte zur Orientierung fehlen. Auch Kontextbeziehungen zwischen Anforderungsartefakten können durch das Ausblenden unbeteiligter nicht unterschieden werden.

Die *City-Metapher* wurde hingegen positiv bewertet. Klassen als zentrale Elemente lassen sich durch Anforderungsartefakte ersetzen, Bodenplatten dienen der Strukturierung. Die Ordnung der Gebäude lässt sich durch klare Bezugspunkte besser nachvollziehen.

Die Konzepte der Lokalität und Habitabilität unterstützen die Ziele der Arbeit, eine intuitive und übersichtliche Darstellung von Anforderungsdokumenten zu erstellen.

Es wurden dennoch weitere Optimierungsmöglichkeiten für den spezifischen Anwendungsfall erarbeitet. Zur erleichterten Orientierung sollen Gebäude und Bodenplatten ihre Namen als Labels erhalten. Dies ist auch von Vorteil bei der explorativen Erkundung, da diese zur Orientierung genutzt werden können. Spezifikationen sollen zusätzlich farblich differenziert und somit leichter zu unterscheiden

sein. Districts sollen alle Kontextbeziehungen der darauf positionierten Elemente darstellen können. So kann bspw. die Testabdeckung, die Umsetzung der Stakeholderbedürfnisse oder die Konflikte eines gesamten Kapitels eines Anforderungsdokuments dargestellt werden.
Das Layout soll überarbeitet werden, sodass sich die Ordnung der Anforderungsartefakte in der Positionierung der Gebäude wiederfindet.
Das Mapping der Attributanzahl auf die Länge bzw. Höhe wird nicht weiterverfolgt, da Anforderungen einer Spezifikation meist vom gleichen <Spec-Type> sind und die selbe Anzahl an Attributen besitzen.
Wenn Gebäude die gleiche Fläche besitzen, wird die Fläche der Bodenplatten abhängig von der Anzahl der darauf positionierten Elemente und kann somit intuitiv als Maß für den Umfang eines Kapitels oder einer Spezifikation verstanden werden.
In Summe resultiert aus den Änderungen eine Abwendung der Metapher von der Echt-Welt hin zu einer *Nested-Tree-Map.*

3.3 Konzeption des Verarbeitungsprozesses

Nach der Auswahl der Elemente und wie sie dargestellt werden sollen wird im folgenden Abschnitt der Verarbeitungsprozess entworfen.

3.3.1 Entwurf eines Datenmodells

Zum flexiblen und skalierbaren Speichern der extrahierten Informationen wird die Graphdatenbank *Neo4J* verwendet. Sie besteht aus Knoten und Kanten. Knoten repräsentieren Objekte und können durch Kanten, die die Beziehungen zwischen den Knoten verkörpern, verbunden sein. Beide können Labels zur Klassifizierung sowie Attribute und Werte besitzen. Zur Kommunikation mit der Datenbank wird die deklarative Abfragesprache *Cypher* verwendet.
Neben *Neo4J* existiert in *Getaviz* auch die Möglichkeit, *Famix* zur Datenspeicherung zu nutzen. Dabei müssten jedoch alle Daten im Arbeitsspeicher gehalten werden. Um die Performance zu gewährleisten, insbesondere mit Blick auf die teilweise sehr umfangreichen Anforderungsdokumente, wurde sich daher für die Nutzung von *Neo4J* entschieden.
Im Folgenden soll das Graphdatenbankmodell kurz beschrieben und die notwendigen Attribute genannt werden. Nicht notwendige, aber in der *ReqIF-Datei* enthaltene, Attribute

und Werte werden mit in die Datenbank überführt. Dies soll Anpassungen im Entwicklungsprozess oder im Zuge einer potenziellen Weiterentwicklung vereinfachen.

Im verwendeten Datenbankmodell wird für jede Spezifikation ein Knoten mit dem Label *:Specification* und dem Attribut *ID* angelegt. Die dazugehörige Ordnung wird durch gerichtete *:IS_PARENT_OF*-Beziehungen realisiert. Am Ende dieser Beziehungen können sich Kapitel oder Anforderungen befinden, sie werden durch die Labels *:Section* bzw. *:Requirement* unterschieden, besitzen aber ansonsten die gleichen notwendigen Attribute. Kontextbeziehungen werden durch Kanten mit dem Label *:LINKS_TO* gespeichert.

Ziel der Datenbank ist es, die in Kapitel 3.2.1 identifizierten Elemente, ihre hierarchische Gliederung und Kontextbeziehungen zur Verarbeitung durch den Prototypen bereitzustellen.

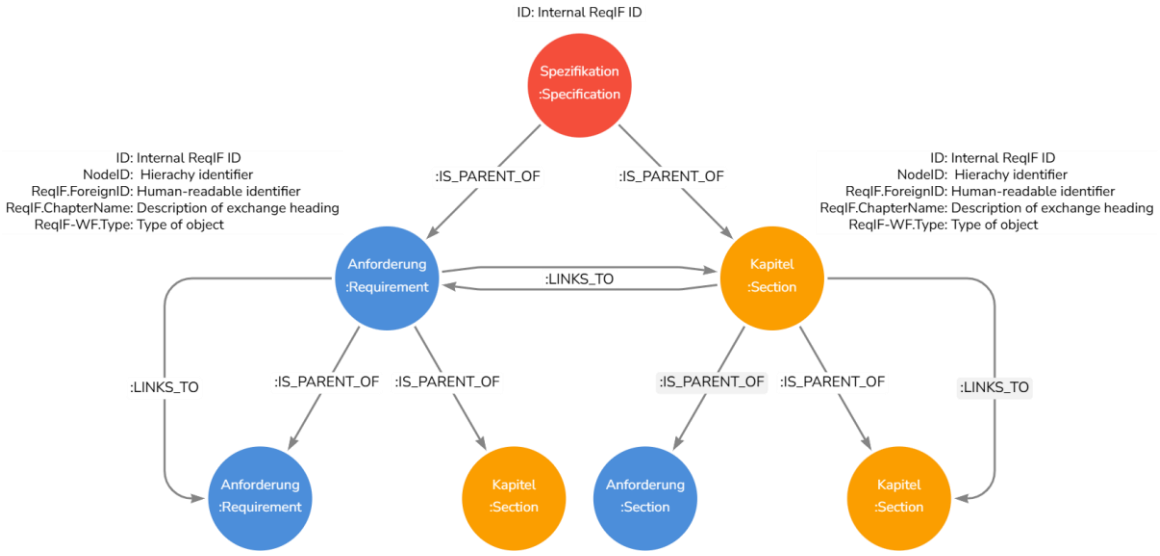

Abbildung 7 - Datenbankmodell des Extraktors

3.3.2 Extraktion der ReqIF-Elemente

Die Verarbeitung der *ReqIF-Dateien* erfolgt, um die Inhalte entsprechend dem Datenmodell in eine Graphdatenbank zu überführen. Um dies durchzuführen, muss ein geeigneter Parser verwendet werden. Folgende Anforderungen an den Extraktionsprozess müssen erfüllt werden:

- Der Extraktor muss *ReqIF-Dateien* verarbeiten können.
- Der Extraktor muss auf einzelne Elemente der *ReqIF-Datei* zugreifen können.
- Der Extraktor muss Verweise über IDs dereferenzieren können.
- Der Extraktor muss die gewonnenen Informationen entsprechend dem Datenmodell in *Neo4J* überführen können.

Aus den Anforderungen ergeben sich verschiedene Möglichkeiten der Umsetzung.
Zunächst betrachtet werden soll die Verwendung des bereits in *Getaviz* integrierten Plug-In *jQAssistant*. Es umfasst ein auf *Neo4J* basierendes Werkzeug zur statischen Codeanalyse, das die Darstellung und Analyse von *XML-Daten* ermöglicht. Es bietet die Möglichkeit, Regeln und Konzepte zu definieren, um die Struktur und Validität der Daten zu überprüfen. Zur Verarbeitung können .reqif-Dateien durch Verwendung der java.io.File-Klasse in .xml-Dateien umgewandelt werden. Anschließend können Regeln in *jQAssistant* genutzt werden, um die resultierenden *XML-Dateien* gemäß den XSD-Schemata zu validieren und in *Neo4J* zu speichern. *Cypher-Querys* können verwendet werden, um auf Klassen zuzugreifen und Referenzen aufzulösen.
Eine Alternative zur Verwendung von *jQAssistant* ist die direkte Umsetzung der beschriebenen Schritte in einem Java-Skript, was den Vorteil der direkten Einbindung in den Generator bietet, ohne zusätzliche Plug-Ins nutzen zu müssen. Im ersten Schritt wird die *javax.xml.parsers*-Bibliothek verwendet, um die *XML-Datei* einzulesen und zu parsen. Anschließend erfolgt die Validierung der *XML-Datei* gegen das Schema (XSD) mithilfe der *javax.xml.validation*-Bibliothek. Nachdem die *XML-Daten* erfolgreich validiert wurden, werden sie mit *JAXB* (Java Architecture for XML Binding) in entsprechende *Java*-Klassen gebunden. Abschließend werden die gebundenen *Java*-Klassen genutzt, um die ID-Referenzierungen aufzulösen und der *Neo4J* Java Driver verwendet, um die Informationen in einer Datenbank zu speichern.
Eine weitere Alternative bietet die Verwendung eines geeigneten Parsers; zwei der untersuchten sollen hier vorgestellt werden. Die Bibliothek *pyreqif* umfasst einen First-Stage-Parser in Form einer einfachen *Python*-Implementierung des ReqIF-Objektmodells. Er bietet Unterstützung für den Export zu *Excel* und HTML sowie für die Validierung. Nach dem Parsen kann auf die *Python*-Objekte zugegriffen werden, um Referenzen aufzulösen und iterativ eine *Neo4J*-Datenbank anzulegen (ebroeker, 2021).
Weiterhin untersucht wurde das Framework *Treqz.* Es bietet, zusätzlich zum Parser, eine umfangreiche Bibliothek, einschließlich einer ausführlichen Dokumentation, von Methoden zum Erstellen, Bearbeiten und Zugreifen auf Elemente des ReqIF-Standards. Der *Neo4J Python* Driver kann anschließend genutzt werden, um die gewonnenen Informationen in der Datenbank zu speichern (tiluhlig, 2022).
Die Entscheidung für die Verwendung des *Treqz*-Frameworks wurde getroffen, da es eine benutzerfreundliche Schnittstelle bietet und zusätzliche Verarbeitungsschritte, insbesondere zum Auflösen der Referenzen, vermieden werden. Dies ermöglicht eine effizientere und schnellere Implementierung im Rahmen des Prototyps.

3.3.3 Generierung eines 3D-Modells

Nach dem Entwurf des Datenmodells und der Auswahl eines geeigneten Parsers wird in diesem Abschnitt der Generierungsprozess kurz skizziert und anschließend notwendige Anpassungen konzipiert. Das Datenbankmodell wird vom Generator als Grundlage zur inkrementellen Erzeugung eines 3D-Modells verwendet. Als Grundlage dient die *Getaviz*-Version zum Verarbeiten von *Java*-Quellcode.

3.3.3.1 Ablauf des Generators

Zunächst werde die wichtigsten Verarbeitungsschritte des Generators skizziert. Als Ausgangspunkt und zur transaktionalen Verarbeitung wird die *Neo4J*-Datenbank verwendet.

Im ersten Verarbeitungsschritt des Generators wird die Klasse *JavaEnhancement* aufgerufen, um für jeden Knoten die Attribute *Name, Hash* und *Fully Qualified Name* zu setzen. Anschließend wird die Klasse *JQA2City* im Package *s2m* verwendet, um jedem Knoten der Datenbank das zur Visualisierung genutzte Metapherelement zuzuweisen.

Die Klasse *JQA2JSON* erzeugt die Datei *metaData.json*. Diese wird verwendet, um dem *User Interface* zusätzliche Informationen zu den Elementen des 3D-Modells bereitzustellen. Dazu gehören Id, Name, Typ, Elternelement und Beziehungen.

Im Paket *model2model* werden die Eigenschaften der Bestandteile des 3D-Modells berechnet. Jedem Gebäude wird eine Länge, Breite und Höhe zugewiesen. Darauf aufbauend wird der *City-Layout-Algorithmus* genutzt, um rekursiv Gebäude und Bodenplatten zu positionieren sowie die Flächen der Districts zu berechnen. Abschließend werden die bisher in der Datenbank gespeicherten Informationen und das *A-Frame-Framework* zur Erstellung eines 3D-Modells genutzt und als *model.html*-Datei an das User Interface übergeben.

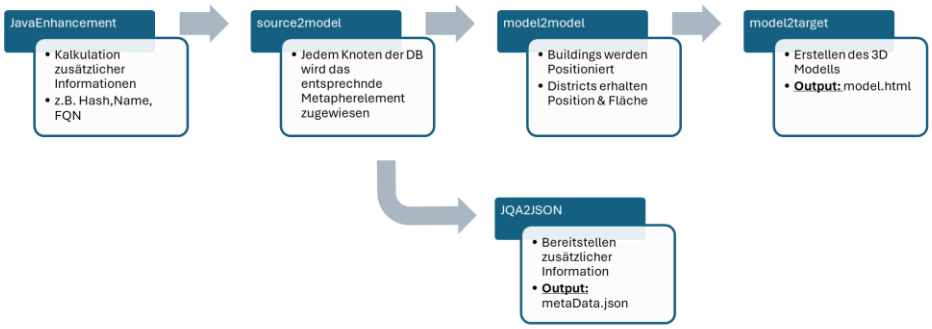

Abbildung 8 - Ablauf der Komponenten des Generators und ihre Aufgabe

3.3.3.2 Anpassungen am Generator

Grundsätzlich sollen die Verarbeitungsschritte des Generators beibehalten werden. Konzeptionelle Änderungen im Generator ergaben sich jedoch durch die Anpassung der Visualisierungsmetapher an den spezifischen Anwendungsfall.

Zunächst sollen die Möglichkeiten zum Hinzufügen von Labels in die Visualisierung diskutiert werden. Zur Erzeugung des 3D-Modells im Generator wird das *A-Frame-Framework* verwendet, das auf *HTML* und *JavaScript* basiert. Es verwendet eine deklarative *HTML*-Syntax, bei der Szenen und Entitäten als *HTML*-Tags definiert werden, und bietet eine modulare Architektur, die durch Komponenten und Systeme erweitert werden kann (A-Frame Documentation).
Um Text in einem A-Frame-Modell darzustellen, gibt es mindestens vier verschiedene Möglichkeiten (vgl. (A-Frame Text Component))

- Verwendung der A-Frame-Text-Komponente: Diese basiert auf der *three-bmfont-text*-Bibliothek und ermöglicht das einfache Hinzufügen von Text innerhalb einer A-Frame-Szene.
- 3D-Text-Geometrie: Diese Methode nutzt *THREE.TextGeometry*, um dreidimensionalen Text zu erzeugen, der in einer 3D-Szene platziert werden kann.
- Manipulation des *Document-Object-Models* (DOM) zur Textdarstellung: Ein Shader rendert *2D-HTML* und *CSS* zu einer Textur, die dann in A-Frame verwendet werden kann.

- Integration von Bilddateien in das 3D-Modell: Hierbei werden Bilddateien, die den darzustellenden Text enthalten, in die A-Frame-Szene eingebunden.

Die Integration von Bildern oder 3D-Glyphen in das *A-Frame-Modell* wurde ausgeschlossen. Dies wird damit begründet, dass Anforderungsdokumente sehr umfangreich sein können und das zusätzliche Erzeugen von Bilddateien bzw. das Rendern einer hohen Anzahl von 3D-Glyphen zu Performanceverlusten führen kann. Auch die Manipulation des DOM erwies sich im Test als sehr rechenintensiv. Darüber hinaus muss die Logik zur Erzeugung des Textes in das User Interface integriert werden. Dies widerspricht dem Prinzip der *Separation of Concerns*, welches im modular programmierten *Getaviz* zur Anwendung kommt.

Die Verwendung der A-Frame-Text-Komponente ist die intuitivste Lösung und bietet umfangreiche Attribute zum Darstellen und Manipulieren von Text im 3D-Modell. Als problematisch erwies sich jedoch das Fehlen eines Attributs zur Bestimmung der Höhe des Textes (*A-Frame Text Component*, n.d.) Es ist möglich, die Höhe des Textes zur Render-Laufzeit mittels *JavaScript* zu manipulieren, aber auch hier müssten für den Generator vorgesehene Aufgeben an das *User Interface* ausgelagert werden.

Die Größe von *A-Frame-Text-Komponenten* wird durch die Attribute *width, wrapCount, wrapPixels, scale* und *position* indirekt beeinflusst. Diese eignen sich jedoch in dieser Form nicht, um sicherzustellen, dass die Labels ihre vorgesehene Höhe nicht überschreiten. Auch die Möglichkeit des Auto-Scalings des Textes, wenn dieser gemeinsam mit einer 2D-freundlichen Geometrie-Komponente verwendet wird, wurde getestet. Dies zeigte, dass in Grenzfällen die Höhe des Textes die Höhe der Geometrie-Komponenten übersteigen kann.

Um die *A-Frame-interne* Kalkulation der Texthöhe nachzuvollziehen, wurde der Quellcode der A-Frame-Text-Komponente und der verwendeten *layout-bmffont-text*-Bibliothek inspiziert (*Layout-Bmfont-Text*). Die Formel zur Berechnung der Höhe ergibt sich aus Abbildung 9. Der *widthFactor* berechnet sich aus der durchschnittlichen Pixelanzahl in Richtung der x-Achse. Durch die Verwendung einer *Monospaced-Font*, das heißt einer Schriftart mit immer gleicher Buchstabenlänge, wird dies umgangen. Die Variable *lineHeight* beträgt, wenn nicht anders definiert, 41 Pixel. Die gekürzte Formel wird genutzt, um die Texthöhe mit der Höhe der Margin zu vergleichen, um ein Überlappen des Textes zu vermeiden.

$$height = textScale \times (layout.height + layout.descender)$$
$$textScale = width / textRenderWidth$$
$$textRenderWidth = 0.5 + wrapCount * widthFactor$$
$$widthFactor = computeWithFactor(Font)$$
$$textScale(SourceCode\,\mathrm{Pr}\,o) = width/484.5$$
$$layout.height = lineHeight - layout.decender$$
$$lineHeight = 41$$
$$height = (width/484.5) \cdot lineHeight$$

Abbildung 9 - Formel zur Berechnung der Höhe der A-Frame Komponente <a-text>

Des Weiteren muss das Layout angepasst werden, um Raum für die Labels zu schaffen und die Ordnung der <Spec-Hierarchy> in der Visualisierung zu repräsentieren. Das Anordnen von *Buildings* oder auch *Districts* in ihrem Eltern-*District* kann als zweidimensionales Rechteck-Problem verstanden werden und wird rekursiv für jede Bodenplatte umgesetzt. In *Getaviz* wird zur Positionierung ein angepasster *kd-Tree-Algorithmus* verwendet (Wettel, 2010). Dies ist ein balancierter Suchbaum zur Speicherung von Punkten und ermöglicht die Durchführung orthogonaler Bereichsanfragen (Bentley, 1975). Der Wurzelknoten des binären Baumes repräsentiert die Gesamtfläche des Eltern-Districts. Da es sich bei binären Packaging-Problemen um ein NP-vollständiges Problem handelt, zunächst als Approximationslösung (Wettel et al., 2010). Die NP-Vollständigkeit des Problems bedeutet, dass bisher kein Algorithmus bekannt ist, der NP-vollständige Probleme nicht-deterministisch in polynomieller Zeit lösen kann (Hans Jurgen Ohlbach; S. 3). Kindknoten des Baumes teilen die verfügbare Fläche in Rechtecke auf und besitzen Informationen über ihre Dimensionen und ob sie bereits belegt sind. *Districts* und *Buildings* werden zweidimensional durch Rechtecke repräsentiert, absteigend nach ihrer Fläche sortiert und anschließend positioniert.

Für jedes Rechteck wird zur Positionsbestimmung der am besten passende, noch freie Knoten gewählt, und Länge und Breite des Knotens werden, wenn nötig, rekursiv durch zusätzliche horizontale und vertikale Schnitte optimiert. Mit jedem Schnitt werden zwei zusätzliche Knoten angelegt. Jede Ebene des daraus resultierenden Baumes repräsentiert dabei entweder einen horizontalen oder einen vertikalen Schnitt.

Sollte die Fläche des Wurzelknotens nicht ausreichen, werden die Dimensionen so erhöht, dass der Eltern-District möglichst quadratisch bleibt. Die Grafik zeigt den Ablauf zur Lösung des binären Packaging-Problems unter Verwendung des *kd-Tree-Algorithmus*.

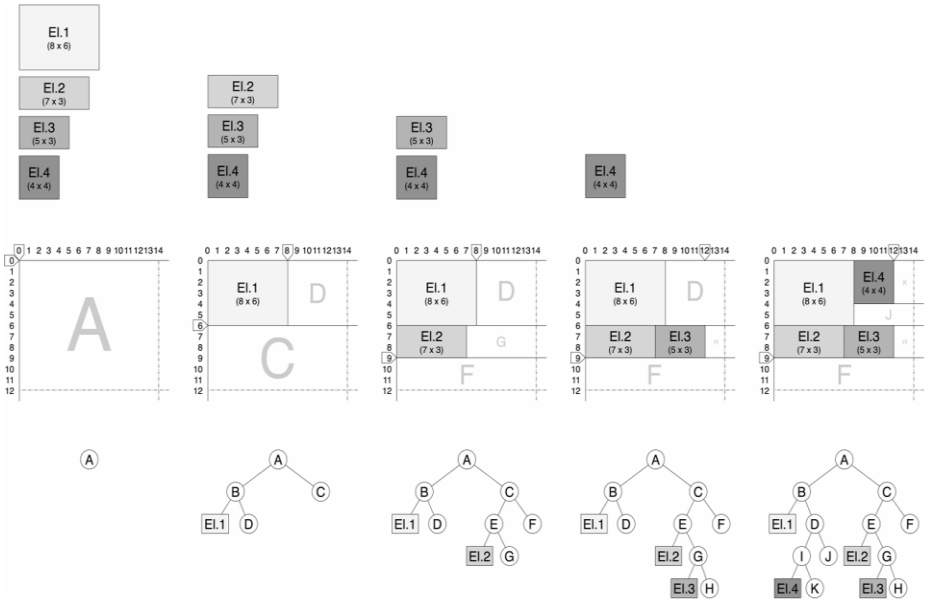

Abbildung 10 - Ablauf zur Positionierung von Rechtecken (vlg. (Wettel, 2010))

Dieser Ansatz wurde ursprünglich gewählt, um die Flächen der Districts möglichst effizient zu nutzen, Containment-Hierarchien darzustellen und die variablen Proportionen der Gebäude zu berücksichtigen (Wettel et al., 2010).

Die Anforderungen dieser Arbeit an das Layout divergieren aufgrund des veränderten Anwendungsfalls. Zusätzlich zur effizienten Flächennutzung wird für jedes District eine Margin für die Labels benötigt. Die Größe des Labels soll dabei abhängig von der Fläche der zugehörigen Bodenplatte sein, um die Lesbarkeit sicherzustellen.

Auch die geordnete Hierarchie der Anforderungsdokumente soll in der Anordnung von Buildings und Districts repräsentiert sein. Weiterhin wurde sich gegen ein Mapping von Eigenschaften der Anforderungsartefakte auf die Proportionen der *Buildings* entschieden, sodass sie alle über die gleiche Fläche verfügen. Durch das Wegfallen der variablen Proportionen der *Buildings* ist es möglich, einen *Treemap-Algorithmus* zu nutzen. Optimal wäre eine umfassende Evaluation der potenziellen Algorithmen mit anschließender Implementierung; dies konnte aber im Rahmen der Arbeit nicht umgesetzt werden.

Anstelle dessen wurde der existierende Algorithmus angepasst, um die genannten Anforderungen bestmöglich umzusetzen. Die Margin für die Labels wird durch prozentuales Erhöhen der District-Fläche erreicht. Um die Reihenfolge der Anforderungen und Kapitel in die Visualisierung einzubinden, wird die Sortierung der Rechtecke im k*d-tree-Algorithmus* nicht nach Größe, sondern nach der Ordnung entsprechend der <Spec-Hierarchy> vorgenommen. Die Elemente werden somit zwar in der Nähe ihres Vorgängers

platziert, jedoch werden alle verfügbaren Knotenpunkte systematisch durchlaufen und derjenige Knotenpunkt ausgewählt, bei dem die Flächennutzung am effizientesten erfolgt. Dadurch ist ein intuitives „Lesen“ von links nach rechts nicht möglich. Zusätzlich sinkt die Performanz des Algorithmus, da zusätzliche Schnitte und das daraus folgende Erstellen und Durchlaufen von zusätzlichen Knoten im Baum notwendig werden. Weiterhin wird rechts eines Knotens A mit geringer Höhe auch eine freie Fläche mit derselben Höhe geschaffen, in welche keine Knoten positioniert werden können, die höher als Knoten A sind. Dies verringert potenziell die Flächeneffizienz.

3.3.3.3 Navigation und Interaktion im User Interface

Nach der Erstellung des 3D-Modells und der Metainformationen durch den Generator werden die beiden Output-Dateien an das User Interface übergeben. Als Schnittstelle zum Endnutzer bietet *Getaviz* ein konfigurierbares, modulares und browserbasiertes User Interface (Baum et al., 2017).

Im folgenden Abschnitt werden die Bestandteile des User Interfaces sowie die Navigations- und Interaktionsmöglichkeiten konzipiert. Der Rahmen wird durch das Habilitäts-Prinzip vorgegeben, also den Zielen, eine intuitive Navigation in und Interaktion mit dem erzeugten 3D-Modell zu ermöglichen.

Zur Navigation in der Visualisierung werden die Möglichkeiten des *A-Frame-Frameworks* genutzt. Mittels des Maus-Cursors können sowohl die Kameraposition als auch die Orientierung des 3D-Modells manipuliert werden. Die Interaktionen zwischen Nutzer und 3D-Modell werden durch ein *Publish-Subscribe-Event-Modell* bereitgestellt. Dabei werden Position und Aktionen der Maus im Bezug zum 3D-Modell nachverfolgt und in Form von Events veröffentlicht. Controller verfolgen diese und manipulieren die Nutzersicht entsprechend. Im Prototypen soll dies genutzt werden, um die Metapherelemente auswählbar zu machen. Angewendet werden soll dies, um Beziehungen zwischen Anforderungsartefakten auf Abruf einzublenden. Im vom Generator erzeugten 3D-Modell wird die Höhe eines Gebäudes durch die Anzahl der Kontextbeziehungen einer Anforderung bestimmt. Diese Kontextbeziehungen sollen durch Auswahl eines Gebäudes eingeblendet werden. Umgesetzt wird dies durch das Ausblenden nicht beteiligter Metapherelemente, das Hervorheben der beteiligten Elemente sowie das Darstellen einer farblich hervorgehobenen Kante zwischen der Quelle und dem Ziel.

Im Falle der Auswahl eines Kapitels oder einer Spezifikation, in der Visualisierung als Bodenplatte dargestellt, sollen zusätzlich alle Beziehungen der Kind-Elemente visualisiert

werden. Die Evaluation der Mockups ergab, dass zu viele Kanten die Usability negativ beeinflussen. In diesem Fall wird daher auf das Einblenden von Kanten verzichtet.
Weiterhin soll die Möglichkeit geschaffen werden, die Struktur des Anforderungsdokuments in Form eines ausklappbaren Baumes darzustellen. Durch Auswahl eines Baum-Elements soll die Kamera das entsprechende Element der Visualisierungsmetapher in den Mittelpunkt setzen. Diese Darstellungsvariante wird bereits im *Windows-File-Explorer*, aber auch in einigen Requirements-Management-Softwareprodukten verwendet. Durch das Bereitstellen einer vertrauten Repräsentation soll die Orientierung und Navigation in der Metapher erleichtert werden.
Zusätzlich sollen, im Fall der Auswahl einer Anforderung, die dazugehörigen Attribute und Werte eingeblendet werden. Durch das Ergänzen der bestehenden Makro-Ansicht um eine Detailansicht wird ein ganzheitlicher Blick auf alle Ebenen des Anforderungsdokuments ermöglicht.

4 Implementierung

Nach dem Entwurf des Prototyps wird in diesem Kapitel die Implementierung vorgestellt. Die Gliederung folgt den Verarbeitungsschritten des Prototyps. Zunächst wird der Extraktor beschrieben, welcher die *ReqIF-Dateien* verarbeitet und die Datenbank initiiert. Anschließend werden die Komponenten des Generators beschrieben. Dieser erstellt gemeinsam mit zusätzlichen Metainformationen das 3D-Modell. Das User Interface schließt den Prozess ab und bietet eine interaktive Nutzerschnittstelle. Im letzten Teil wird auf Tests und Evaluation eingegangen.

4.1 Extraktor

Die bisherige Umsetzung des Extraktors in Form eines *jQAssistant-Plugins* wurde komplett abgelöst. Dafür wurde ein *Python*-Skript geschrieben, welches maßgeblich auf den Bibliotheken *TReqzmaster* und *Neo4J Java Driver* aufbaut. *Treqzmaster* ist eine Library zum Parsen und Verarbeiten von *ReqIF-Dateien*. Die *XML-Klassen* der *ReqIF*-Datei werden ausgelesen, dereferenziert und zu einem *Python Object* konvertiert, welches Inhalt und Struktur der *ReqIF-Datei* repräsentiert. Zusätzlich ermöglichen eine Vielzahl von Methoden das Zugreifen auf sowie das Manipulieren und Erstellen von Spezifikationen, Anforderungen, Attributen und Werten. Der *Neo4J Driver* wird zum Verbindungsaufbau für die Erstellung von Knoten, Attributen und Beziehungen in der Datenbank verwendet.

Zunächst werden alle Spezifikationen extrahiert. Für jede Spezifikation wird ein Knoten angelegt und die Methode *getDocumentHierarchicalRequirementId* verwendet, um die zugehörige <Spec-Hierarchy> zu speichern. Die Funktion *iterate_requirements* wird rekursiv aufgerufen, um die hierarchische Struktur der Spezifikationen zu durchlaufen und für jeden Eintrag das referenzierte <Spec-Object> samt Attributen und Werten in der *Neo4J*-Datenbank zu speichern. Die hierarchische Ordnung entsprechend der <Spec-Hierarchy> wird zusätzlich über *:IS_PARENT_OF*-Kanten abgebildet. Jeder dieser Knoten erhält ein Label entsprechend des *ReqIF-WF.Type-Attributes*. Sollte es sich um den Typ *Information* handeln, wird das <Spec-Object> ignoriert und die Beziehung des folgenden Knotens stattdessen zum letzten in *Neo4J* gespeicherten Knoten erstellt. Hinzugefügt wird jeweils noch das Attribut *getaviz_Order*, dessen Wert mit jedem Aufruf inkrementiert wird. Da die Verarbeitungsfolge der Knoten iterativ erfolgt und der <Spec-Hierarchy> entspricht, können andere Komponenten das Attribut nutzen, um die Ordnung der Anforderungen im ReqIF-Dokument nachzuvollziehen. Dies findet Anwendung in der Sortierung von Anforderungen mit gleichem Eltern-Knoten, bspw. in der Klasse *CityLayout.java* oder im *PackageExplorer.js*.

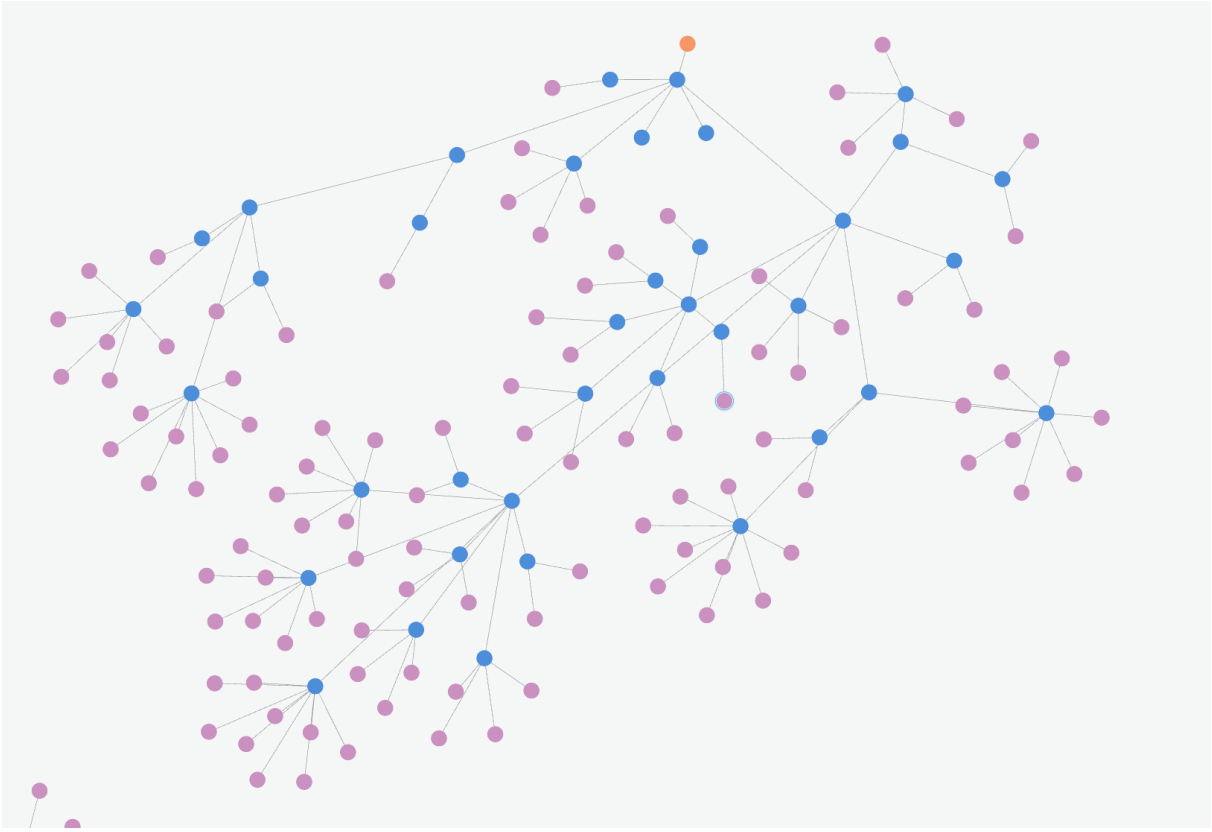

Abbildung 11 - Darstellung einer Spezifikation im Neo4J Browser

4.2 Generator

Der Generator nutzt die Repräsentation der ReqIF-Datei in *Neo4J,* um inkrementell die Datenbank zu erweitern. Als Output werden ein 3D-Modell sowie zusätzliche Metainformationen bereitgestellt und an die Benutzeroberfläche übergeben. Umfangreiche Änderungen im Generierungsprozess ergaben sich durch das neue Datenbankmodell und anwendungsfallspezifische Anpassungen am 3D-Modell. Die Verarbeitungsschritte wurden generell beibehalten, ebenso Komponenten- und Klassenbezeichnungen, um den Vergleich mit anderen *Getaviz*-Versionen zu vereinfachen. Namen von Funktionen und Variablen wurden angepasst, wenn es dem Verständnis des Codes dienlich ist. Um die Wartbarkeit zu gewährleisten, wurden nicht verwendete Codefragmente entfernt.

4.2.1 *Neo4J* Driver

Zunächst wird die Migration des *Neo4J* Drivers beschrieben. Dieser ist für die Kommunikation zwischen Generator und Datenbank zuständig. Daher ist nahezu der gesamte Generierungsprozess betroffen. Notwendig wurde es, da die bisher verwendete *Neo4J*-Datenbank Version 3.5.2 obsolet ist und von Entwicklerseite nicht mehr unterstützt oder zum Download angeboten wird (*Neo4J - Supported Versions*).

Bisher ist eine *Neo4J*-Datenbank Version 3.5.2 Bestandteil der *Getaviz Docker-Container*. Diese eignen sich jedoch nur bedingt zum Testen während des Entwicklungsprozesses, da Docker auf Nicht-Linux-Systemen eine virtuelle Maschine verwendet, um den notwendigen Linux-Kernel bereitzustellen. Dies wirkt sich negativ auf die Performance aus. Um eine lokale *Neo4J*-Datenbank Version 5.12.0 verwenden zu können, wird der Driver auf die Version 5.23.0 migriert, da die bisher verwendete Version 1.7.2 nicht mit Datenbanken ab Version 4.x kompatibel ist. Im Zuge der Migration ergeben sich einige *Breaking Changes*, hervorzuheben ist insbesondere, dass Ergebnisse von Datenbankabfragen nur einmal konsumiert werden können (*Neo4J - Breaking Changes*). Daraus resultieren Veränderungen in der Verarbeitung und Handhabung von Abfrageergebnissen.

```
88      public List<Record> executeRead(String statement) {
89          try (Session session = driver.session()) {
90              return session.readTransaction(tx -> {
91                  Result result = tx.run(statement);
92                  List<Record> records = result.list();
93                  return records;
94              });
95          } catch (Exception e) {
96              e.printStackTrace();
97              return Collections.emptyList();
98          }
99      }
```

Abbildung 12 - Methode executeRead aus DatabaseConnector.java

```
223         List<Record> childDistricts = connector.executeRead(
224             String.format("MATCH (n)-[:CONTAINS]->(c:District) WHERE ID(n) = %d RETURN c", nodeId));
225         for (Record result : childDistricts) {
226             setDistrictAttributes(result.get("c").asNode().id(), colorGradient, level + 1);
227             }
228         }
```

Abbildung 13 - Ausschnitt aus JQA2City als Beispiel für die Verarbeitung von List<Record>

4.2.2 nodeEnhancement

Der erste Verarbeitungsschritt des Generators erfolgt in der Klasse *JavaEnhancement.* Jedem Knoten in der Datenbank werden die Attribute *name*, *fqn* – kurz für *fully qualified name* – und *hash* erstellt und zugewiesen. Der Name soll als für den Menschen lesbarer Bezeichner fungieren, das Attribut *fqn* baut darauf auf, stellt aber die Bedingung, eindeutig zu sein. Der Hash-Wert wurde bisher aus dem *fqn* erzeugt.

Der ReqIF-Standard trifft keine Vereinbarungen bezüglich der Verwendung von Bezeichnungen für <Spec-Objects>, daher wird auf die getroffenen Einschränkungen im Kapitel 3.2 verwiesen. Die Verarbeitung erfolgt nach den Knotenklassen der Datenbank. Zuerst werden die Spezifikationen, dann Kapitel und abschließend die Anforderungen verarbeitet. Am Beispiel des Kapitels soll der Ablauf kurz skizziert werden. Zunächst werden alle Spezifikations-Knoten gesucht, um anschließend rekursiv alle damit verbundenen Kapitel-Knoten, sortiert nach ihrer Tiefe, zurückzugeben.

Anschließend wird die Methode *enhanceSection* aufgerufen. Zunächst wird geprüft, ob die Attribute *ReqIF.ChapterName* oder *ReqIF.ForeignID* existieren und nicht null sind; ansonsten wird die interne Datenbank-ID verwendet. Das Attribut *fqn* setzt sich zusammen aus dem Präfix *Reqif.*, dem Namen der zugehörigen Spezifikation und, falls vorhanden, den Namen der übergeordneten Kapitel. Durch die Verarbeitung, abhängig von der Tiefe, wird sichergestellt, dass der *fully qualified name* des Eltern-Knotens bereits gesetzt ist. Der Hash-Wert wurde bereits durch die ReqIF-Interne ID ersetzt.

Für Anforderungen wird die Methode *checkRequirementsRelations* verwendet, um zu prüfen, ob Eltern-Kind-Beziehungen zwischen Anforderungen existieren. Diese können im *City-Layout* nicht dargestellt werden. Anstelle dessen wird die *:IS_PARENT_OF*-Beziehung zu dem ersten Eltern-District-Knoten gezogen. Eine bessere Umsetzung wäre in der *City-Panel-Metapher* möglich, indem Eltern-Anforderungen als Würfel und Kind-Anforderungen als Zylinder darauf dargestellt werden.

```
70      private String collectAllSectios(){
71          return "MATCH (root:Specification) " +
72                  "CALL {" +
73                  "  WITH root" +
74                  "  MATCH (root)-[:IS_PARENT_OF*]->(child:Section)" +
75                  "  RETURN child" +
76                  "} " +
77                  "WITH root, collect(child) AS childs " +
78                  "UNWIND childs AS Section " +
79                  "RETURN Section " +
80                  "ORDER BY Section.depth ";
81      }
82
```

Abbildung 14 - Cypherquery zur geordneten Rückgabe der Kapitel

4.2.3 source2model & JQA2JSON

Die Komponente *source2model* wird genutzt, um den Knoten der Datenbank die entsprechenden Metapherelemente zuzuweisen. Dafür wird der Graph rekursiv entlang der *:IS_PARENT_OF*-Beziehungen traversiert. Für Spezifikationen oder Kapitel wird ein District-Knoten und für Anforderungen ein Building-Knoten erstellt. Jeder neue Knoten ist über eine *:VISUALIZES*-Beziehung mit dem entsprechenden *ReqIF-Knoten* und über eine *:CONTAINS*-Beziehung mit dem Eltern-District verbunden.

Building-Knoten erhalten zusätzlich das Attribut *numberOfLinks*. Dieses entspricht der Summe aller ausgehenden Kontextbeziehungen der repräsentierten Anforderung. Dieses Attribut wird zur Bestimmung der Höhe verwendet.

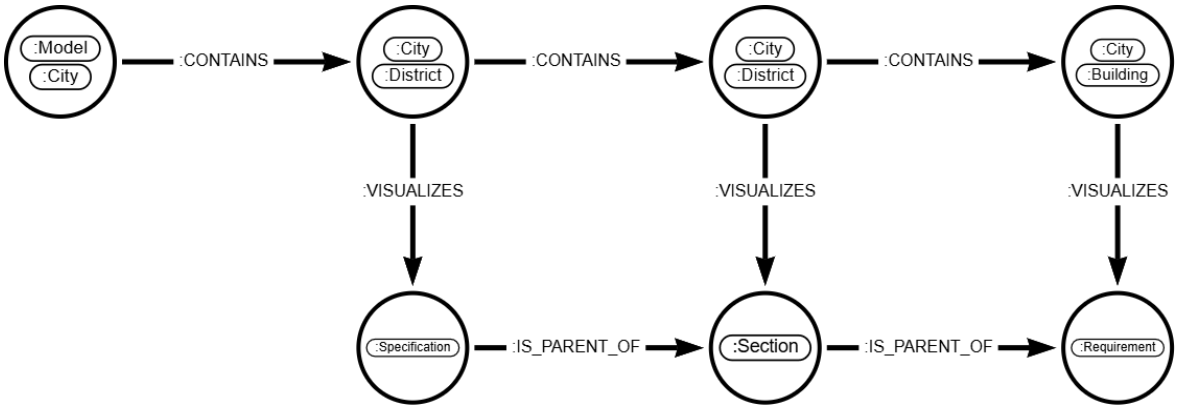

Abbildung 15 - Datenmodell nach Ausführung der Komponente source2model

Die Klasse *JQA2JSON* wird verwendet, um dem *User Interface* zusätzliche Informationen in Form einer *JSON*-Datei bereitzustellen. Diese werden zur Interaktion, Anzeige von Beziehungen, PackageExplorer und SourceCodeExplorer verwendet. Dafür erhalten alle Buildings und Districts die Attribute *id, qualifiedName, type, subClassOF, superClassOf, belongsTo* und *nodeID*. Die Attributsnamen wurden beibehalten, um die Integration der *metaData.json*-Output-Datei ohne umfangreiche Änderungen im *User Interface* verarbeiten zu können. Die ID wird durch die interne ReqIF-ID ersetzt. Das Attribut *type* entspricht noch der auf *FAMIX* basierenden Urversion des Generators. Gebäude erhalten den Wert *FAMIX.Class* und Districts den Wert *FAMIX.Namespace*. Über die Attribute *subClassOF* und *superClassOf* werden die ein- bzw. ausgehenden Beziehungen gespeichert. Im Falle der Districts werden zusätzlich alle Beziehungen der Kind-Elemente gespeichert. Eine ähnliche Funktion existiert bereits in der Klasse *model.js* des *User Interfaces*; eine Berechnung zur Renderzeit wird jedoch als suboptimal betrachtet. Das Attribut *belongsTo* beinhaltet die ID des Eltern-Elements. Gemeinsam mit dem neu eingeführten Attribut *nodeID* ermöglicht es die Repräsentation der geordneten Hierarchie.

4.2.4 model2model

Die Komponente *model2model* dient der Kalkulation notwendiger Attribute wie Proportionen und Positionen zum Erstellen des 3D-Modells. Die Klasse *City2City* wurde so angepasst, dass Gebäude fixe Dimensionen besitzen. Die Höhe der Gebäude repräsentiert die Anzahl der ausgehenden Kontextbeziehungen der visualisierten Anforderung. Spezifikationen und dazugehörige Kapitel sollen zur besseren Orientierung farblich unterschieden werden. Außerdem soll das Konzept der Topologie in Form eines Farbverlaufs umgesetzt werden.

Die Methode *setDistrictAttributes* hat die Aufgabe, die Attribute *height* und *color* für einen Spezifikations-District-Knoten und seine untergeordneten Kapitel-District-Knoten zu setzen. Dabei wird die hierarchische Struktur von District-Knoten durchlaufen und jedem Knoten eine Farbe aus einem Farbverlauf (colorGradient) zugewiesen. Zur dynamischen Bestimmung des Farbverlaufs wird die Methode *getDistrictsColors* eingeführt. Diese erstellt für jede Hierarchieebene eine Farbe. Die Farben werden mit zunehmendem Hierarchie-Level heller, um die Topologie abzubilden

```
275     private void getDistrictsColors() {
276         List<Record> rootDisricts = connector.executeRead
277         ("MATCH (n:Model:City)-[:CONTAINS]->(d:District) RETURN d");
278         for (Record root : rootDisricts) {
279             Node rootNode = root.get("d").asNode();
280             List<Record> districtMaxLevelResult = connector.executeRead(
281                 String.format("MATCH p=(n:District)-[:CONTAINS*]->(m:District)"  +
282                 "WHERE ID(n)=%d AND NOT (m)-[:CONTAINS]->(:District) RETURN length(p)" +
283                 "AS length ORDER BY length(p) DESC LIMIT 1", rootNode.id()));
284             int districtMaxLevel = 1; //Mit 0 funktioniert es nicht
285             if (!districtMaxLevelResult.isEmpty()) {
286                 districtMaxLevel = districtMaxLevelResult.get(0).get("length").asInt() + 1;
287             }
288             String[] colorPair = generateColorPair();
289             List<String> colorGradient = ColorGradient.createColorGradient(colorPair[0], colorPair[1], districtMaxLevel);
290             districtColors.put(rootNode.id(), colorGradient);
291         }
292     }
293
294     //*  Bis zu 6 Farbpaare, danach wird wieder von beginn an
295     private String[] generateColorPair() {
296         String[] colorPair = predefinedColors.get(colorPairIndex);
297         colorPairIndex = (colorPairIndex + 1) % predefinedColors.size();
298         return colorPair;
299     }
300
```

Abbildung 16 - Methode getDistrictsColors aus City2City.java

Die Klasse *CityLayout* positioniert Buildings und Districts und berechnet die dafür notwendige Fläche. Zur Berechnung wird ein *kd-Tree-Algorithmus* angewendet. Zunächst wird rekursiv, ausgehend vom Model-Knoten über die Abfrage der Kind-Knoten, eine Baumstruktur erstellt. Anschließend werden, von den Blattknoten aufwärts, für jeden District die Positionen seiner Kinder bestimmt. Dabei wird die Fläche des Districts zunächst aus der Summe der Längen und Breiten als Annäherungslösung berechnet. Anschließend werden die Kinder nach absteigender Größe sortiert und positioniert. Jedes Element erhält den noch freien Platz, welcher die nicht genutzte Fläche minimiert. Sollte die Fläche der Annäherungslösung nicht ausreichen, kann sie erhöht werden. Positionen der Kinder und die tatsächlich genutzte Fläche des Districts werden im Baum gespeichert.
Wurden alle Eigenschaften des Baumes berechnet, werden abschließend die noch voneinander unabhängigen Positionen angepasst, um sie in ein gemeinsames Koordinatensystem zu überführen.

Das Modul verfügt über weitere Hilfsklassen, bspw. zur Interaktion mit dem kd-Tree oder zur Berechnung von Rechteckattributen. Um die geordnete Hierarchie des ReqIF-Standards im Layout zu repräsentieren, wird die Klasse *sortChildrenAsRectangles* angepasst. Diese konvertiert Kinder eines Districts zu Rechtecken, um sie anschließend nach absteigender Fläche zu sortieren und wird im *kd-Tree-Algorithmus* verwendet. Die existierende Sortierung wurde an dieser Stelle entfernt. Stattdessen wird die Methode *getChildren* um eine Sortierung erweitert. Diese wird sowohl zur Traversierung des Baumes verwendet, indem untergeordnete *City-Knoten* eines gegebenen Knotens zurückgegeben werden, als auch zur Berechnung der Positionen der Kinder eines Districts. Hinzugefügt wird die Abfrage des jeweilig visualisierten *ReqIF*-Knotens, um das *getavizOrder*-Attribut zur Sortierung zu nutzen.
Eine Margin von 5% um die Bodenplatten wird hinzugefügt, indem zunächst die Methode *arrangeChildren* angepasst wird. Nach dem Ausführen des *kd-Tree-Algorithmus* wird die kalkulierte Fläche des Districts um 10% erhöht und dann im Baum gespeichert. Die Methode *adjustPosition* wird angepasst, um beim Zusammenführen der Positionen die Kinder eines Districts weiterhin mittig zu platzieren. Die X- und Y-Koordinaten werden dafür zusätzlich um 5% erhöht.

4.2.5 model2target

Der letzte Verarbeitungsschritt des Generators überführt die Informationen der Datenbank in ein A-Frame-Modell. Dieses soll um Labels erweitert werden. Buildings und Districts werden durch <a-box>-Elemente repräsentiert, zugehörige Labels werden in Form von <a-text>-Elementen in diese eingebettet. Die <a-text>-Attribute sind in der folgenden Abbildung zusammengefasst.

Buildings:

```
<a-text      class        = BuildingLabel
             position     = x= 0 | y= 0.5 x building.height + 0.001 | z= 0
             rotation     = -90 0 0
             color        = white
             width        = building.width *0.9
             value        = getBuildingLabel(ReqIF.ForeignID)
             wrap-count   = 10   > </a-text>
```

Districts:

```
<a-text      class        = DistrictLabel
             value        = getBuildingLabel(ReqIF.ChapterName)
             position=    x=     district.width *  0.45
                          y=     district.height * 0.5 + 0.001
                          z=     district.length * 0.475
             rotation     = -90 0 0
             color        = black
             width        = district.width *0.75
             wrap-count   = 25   > </a-text>
```

Abbildung 17 - Attribute der Text Komponenten zur Darstellung der Label

Zusätzlich werden Helfer-Methoden zur Bestimmung des Namens verwendet. Dabei werden Namen auf eine einheitliche maximale Länge gekürzt. Im Falle von Buildings werden, falls die *ReqIF.ForeignID* zu lang ist, nur die enthaltenen Zahlen dargestellt, da sich das Attribut *ForeignID* meist aus einem Spezifikations-Kürzel und einer eindeutigen Nummer zusammensetzt. Für Districts wird, wenn nötig, der Name gekürzt und die letzten 3 Zeichen durch „…“ ersetzt. Die Klasse *checkHeightLabel* wird verwendet, um die Höhe der Labels auf potenzielle Überlappungen zu überprüfen. Sollten sie die Höhe übersteigen, wird das Attribut *wordwrap* inkrementell erhöht. *Wordwrap* bestimmt die Anzahl von Buchstaben, die innerhalb der verwendeten Länge dargestellt werden können. Somit nimmt es Einfluss auf die Länge, die jedem Buchstaben zugewiesen wird, und damit auch auf die Höhe des Textes.

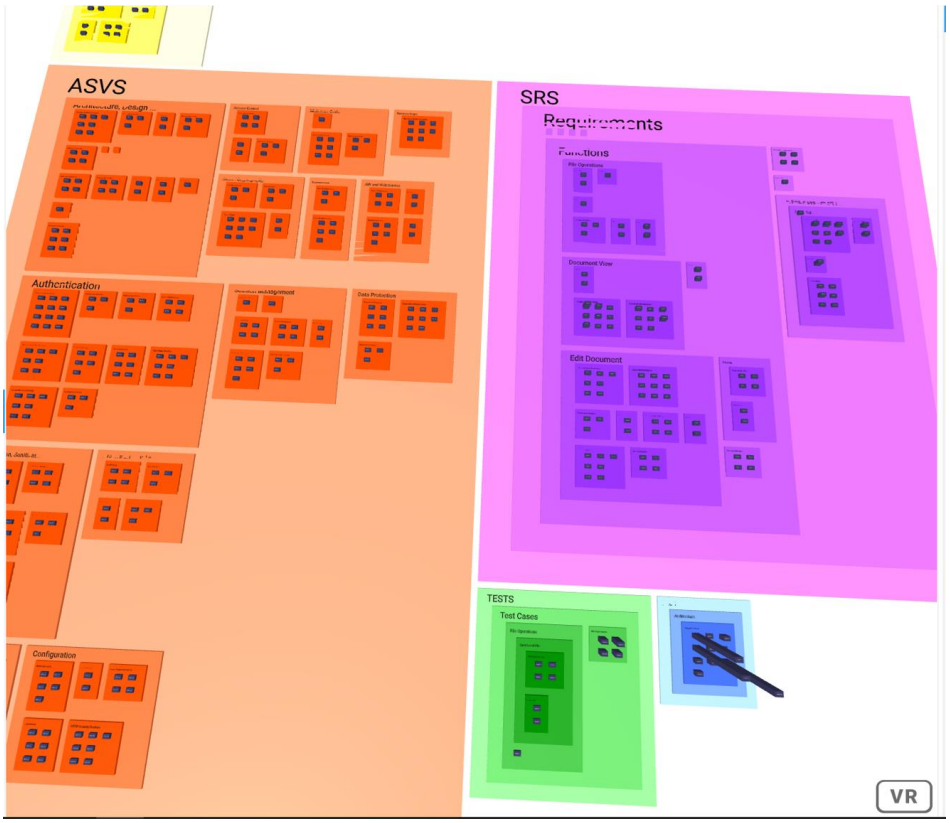

Abbildung 18 - Ausschnitt des fertigen 3D-Modells

4.3 User Interface

Das *User Interface* dient der Steuerung des Verarbeitungsprozesses des Generators sowie dem Rendern und Interagieren mit dem 3D-Modell. Die Datei *index.php* wird als Haupteinstiegspunkt für die browserbasierte Benutzeroberfläche genutzt. Sie lädt alle notwendigen Skripte und Stylesheets, die für das Funktionieren der Anwendung erforderlich sind. Das Modul *model.js* verarbeitet anschließend die *metaData.json*-Datei und initialisiert die Entitäten des internen Datenmodells. *application.js* ist für die Initialisierung und Verwaltung der Controller und des UI-Layouts verantwortlich. Erst nach erfolgreicher Initiierung des Datenmodells und aller verwendeten Controller wird die Benutzeroberfläche geladen. Durch die Erweiterungen im Zuge unterschiedlicher wissenschaftlicher Arbeiten steigt auch die Komplexität der Abhängigkeiten zwischen Modellen, Events und Controllern. Eine Kürzung des Quellcodes um nicht verwendete Codeartefakte wurde im Rahmen der Arbeit nicht umgesetzt.

Das Modul *model.js* wird um die Verarbeitung der neu integrierten Attribute der *metaData.json*-Datei erweitert, sodass ein Zugreifen auf die anwendungsfallspezifischen Informationen wie District-Beziehungen und der *NodeID* ermöglicht wird.

Der *PackageExplorerController* soll Districts und Buildings entsprechend ihrer geordneten Hierarchie in einem ausklappbaren Baumdiagramm anzeigen. Das existierende Modul wird lediglich um die Verarbeitung des Attributs *NodeID* erweitert. Zur Erstellung des Baumes wird die *jQuery*-Bibliothek *zTree* verwendet. Jedem Item des *zTrees* wird die *NodeID* der repräsentierten Model-Entität als Attribut zugewiesen. Nachdem es zur Sortierung der Items genutzt wurde, wird es gelöscht, um die Verarbeitung durch die *zTree*-Bibliothek nicht zu beeinträchtigen.

Die Detailsicht auf Attribute und Werte wurde unter Verwendung des *SourceCodeControllers* implementiert. Ursprünglich wurde das Modul zum Anzeigen von Quellcode verwendet. Neu implementiert wurde, dass der Name sowie Attribute und Werte der ausgewählten Anforderung tabellarisch dargestellt werden. Um auf diese Informationen zugreifen zu können, wurde dem Extraktor ein Skript hinzugefügt, welches das erstellte *Python*-Objekt in eine *JSON*-Datei umwandelt. Diese basiert maßgeblich auf einem in der *Treqz*-Bibliothek enthaltenen Beispiel. Über die Verwendung der *ReqIF*-Internen ID im UI-internen Entitätsmodell wird auf das entsprechende *JSON*-Objekt zugegriffen.

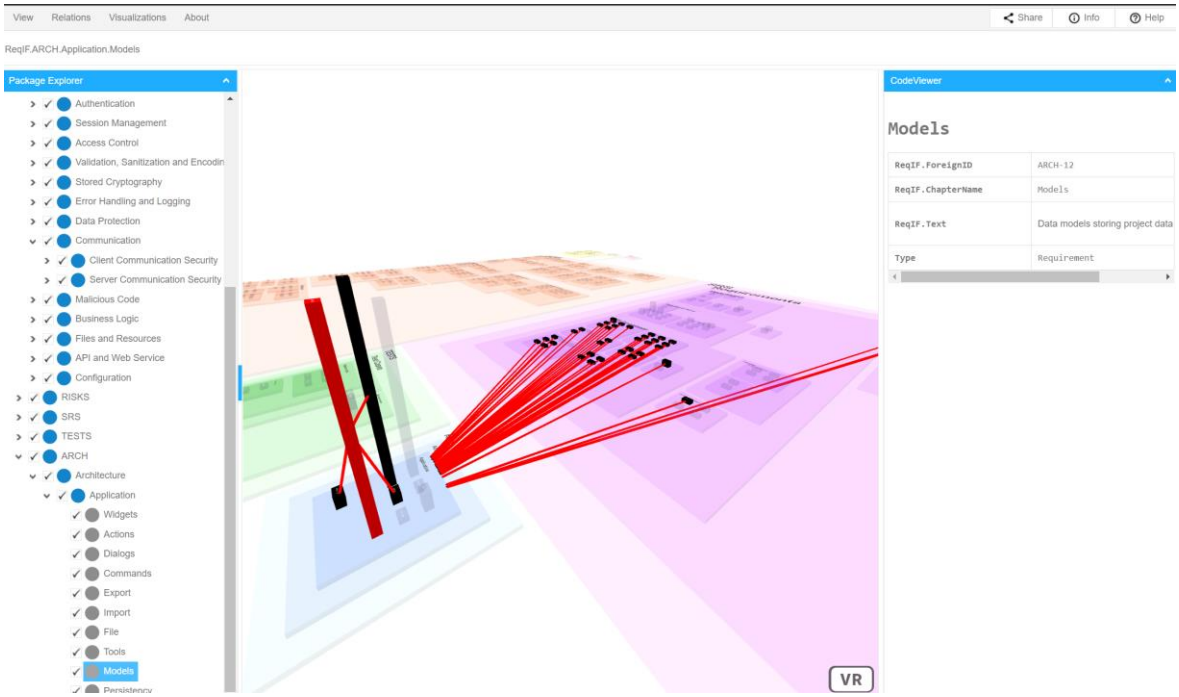

Abbildung 19 - Ausschnitt aus dem User Interface

4.4 Tests und Evaluation

Das Testen der Funktionalitäten ist ein wichtiger Bestandteil der Forschungsethos des Prototypings. Zur Durchführung der Tests wurde ein Anforderungsdokument im ReqIF-Standard verwendet und den Einschränkungen des Kapitels 3.2 angepasst. Die Ergebnisse des Extraktors wurden in der Datenbank sichtgeprüft. Zum Testen des Generators wurden die vom Extraktor erstellten Datenbanken verwendet. Um ein lokales Testen zu ermöglichen, wurde das Servlet angepasst, sodass der Generator nach der Kompilierung einen lokalen Server zur Anwendungsbereitstellung nutzen kann. Weiterhin wurden umfangreiche Logs verwendet, um Fehler im Verarbeitungsprozess nachvollziehen zu können. Die erzeugten Daten des Generators und Extraktors wurden zur Sichtprüfung vom User Interface verarbeitet und die Funktionalitäten getestet.

Die Evaluation schließt die Forschungsmethode des Prototypings ab. Die Durchführung von Interviews könnte wertvolle Einblicke in mögliche Verbesserungen und bestehende Mängel, insbesondere in Bezug auf die Benutzerfreundlichkeit, liefern. Ebenso wäre es denkbar, die erzeugte Visualisierung mit klassischer Requirements-Management-Software zu vergleichen. Eine mögliche Aufgabe könnte darin bestehen, Anforderungen wiederzufinden oder Kontextbeziehungen, beispielsweise in Form der Testabdeckung oder Auswirkungen von Änderungen einer Anforderung, nach einer kurzen Einarbeitungsphase wiederzugeben. Eine abschließende Nutzerevaluation des Prototyps konnte aufgrund des Umfangs der Arbeit jedoch nicht durchgeführt werden.

5 Fazit

5.1 Zusammenfassung

Die Arbeit untersucht die Visualisierung von Anforderungsdokumenten. Das Softwarevisualisierungswerkzeug *Getaviz* wurde erweitert, um ReqIF-Dateien zu verarbeiten und in einem 3D-Modell darzustellen. Dafür wurde die Forschungsmethode des Prototyping gewählt (vgl. (Wilde & Hess, 2007)). Zunächst wurden relevante Elemente des ReqIF-Standards identifiziert. Danach wurden die beiden Metaphern *Recursiv-Disk* und *City* vorgestellt und miteinander verglichen. Umgesetzt wurde anschließend die *City-Metapher*, welche um Aspekte der *nested Treemap* erweitert wurde, um Struktur und Kontextbeziehungen eines Anforderungsdokuments zu modellieren. Die Konzepte der Lokalität und Habitabilität wurden auf den Anwendungsfall übertragen und sollen eine

intuitive Navigation und Interaktion mit der Visualisierung ermöglichen. Die prototypische Erweiterung von *Getaviz* umfasst einen Extraktor, Anpassungen am Verarbeitungsprozess sowie im User Interface. Das Ergebnis der Arbeit lässt den Schluss zu, dass Werkzeuge und Konzepte der Softwarevisualisierung zur Metapher-basierten Modellierung von Anforderungsdokumenten verwendet werden können.

5.2 Kritische Würdigung

Ziel der prototypischen Erweiterung von *Getaviz* ist es, ein *Proof of Concept* zu schaffen. Die fehlende Evaluation hat zur Folge, dass *Usability* und Nutzen der Modellierung nicht abschließend geklärt werden können. Es fehlt außerdem die Zusammenführung der einzelnen Komponenten, sie müssen zurzeit manuell ausgeführt werden. Weiterhin fehlen einige konzipierte Funktionalitäten. Zum Beispiel fehlt dem User Interface zur Zeit die Funktionalität, die Beziehungen der Bodenplatten abzubilden.

Untersucht wurden nur textuelle, attributisierte Anforderungsdokumente im ReqIF-Format. Die für die Verarbeitung notwendigen Vorgaben an Struktur und Inhalt machen den Einsatz des Prototyps unter realen Bedingungen unplausibel. Anwendungsfall- und unternehmensspezifische Anpassungen sind zu erwarten.

Die Vereinbarungen zu den notwendigen Attributen, insbesondere die Begrenzung des Attributs *ReqIF-WF.Type* auf die Ausprägungen *Information*, *Requirement* und *Heading*, schränken die Verwendung des Prototyps zusätzlich ein. Weiterhin werden <Spec-Objects> vom Typ *Information* nicht in die Visualisierung eingebunden.

Die Verarbeitung und Visualisierung von Kontextbeziehungen wurden nur begrenzt umgesetzt. Im Prototypen besitzen alle Kontextbeziehungen die gleichen visuellen Charakteristika. Im ReqIF-Standard können Kontextbeziehungen Attribute und Werte besitzen und zwecks Klassifizierung in <SpecRelationGroups> gruppiert werden; dies wird weder verarbeitet noch dargestellt. Auch werden im 3D-Modell nur ausgehende, nicht aber eingehende Beziehungen angezeigt. Um die Anzahl der Kontextbeziehungen in Form der Höhe der Gebäude darzustellen, wurde ein lineares Mapping gewählt, um einen Vergleich zu ermöglichen. In der *CodeCity-Metapher* werden noch die Möglichkeiten zum Threshold- und Boxplot-Mapping untersucht. Diese würden kein direktes Vergleichen ermöglichen, würden aber zur Habitabilität beitragen.

Kritisch zu betrachten ist auch die Verwendung des angepassten Layout-Algorithmus. Entwickelt zur Verarbeitung von objektorientierten Quellcodes bildet er die Struktur des ReqIF-Standards nur bedingt ab. So ist die Repräsentation der geordneten Hierarchie im

City-Layout nur teilweise gegeben. Auch Eltern-Kind-Beziehungen zwischen Anforderungen werden nicht in dieser Form dargestellt.
Zusätzlich muss der Algorithmus bei jeder Veränderung im Layout neu durchlaufen werden und ist sehr rechenintensiv. Gemeinsam mit der aufwendigen Generierung des 3D-Modells wird die Nutzung des Prototyps eingeschränkt.

5.3 Ausblick

Nach dem Zusammenfassen der Ergebnisse dieser Arbeit und der Diskussion kritischer Punkte werden nun weiterführende Themen und mögliche nächste Schritte erörtert. Dieser Abschnitt ist in zwei Teile gegliedert: Zunächst werden Erweiterungen besprochen, die sich direkt auf die hier durchgeführte Implementierung beziehen. Anschließend werden verschiedene weiterführende Forschungsthemen vorgestellt.

Zunächst zu nennen ist die Durchführung und Planung einer ausführlichen Evaluation des Prototyps und die Untersuchung der Hypothese, dass sich 3D-Visualisierungen von Anforderungsdokumenten zur Erkenntnisgewinnung eignen.
Bezüglich der Layout-Berechnung des Prototyps wäre entweder die Anpassung oder die Neu-Implementierung eines Algorithmus zu empfehlen. Dazu notwendig ist die Analyse verschiedener potenzieller Algorithmen. Implementiert werden sollte ein Algorithmus, der sich zur Abbildung von geordneten Hierarchien eignet und strukturelle Änderungen umsetzen kann, ohne neu durchlaufen werden zu müssen. Wettel (2010) betont, dass kein klassischer *nested-Treemap*-Algorithmus verwendet wurde, da diese ungeeignet zur Darstellung der variablen Proportionen von Gebäuden seien. Da auf ein Mapping von Attributen auf die Länge und Breite von Gebäuden verzichtet wurde, ergeben sich potenziell effizientere Lösungen. Der *EvoSpaces*-Algorithmus ist, wie der aktuell verwendete, ein Containment-Treemap-Algorithmus und wurde entwickelt, um dynamisch auf Änderungen im Layout zu reagieren (Lanza et al., 2009). Auch vom *kd-Tree*-Algorithmus gibt es potenziell bessere Ausprägungen, um die Ordnung zu repräsentieren. Denkbar wäre die *Next-Fit*-Variante, um Elemente einer Bodenplatte rechts vom vorherigen Element zu platzieren und beim Erreichen des Limits eine weitere „Zeile“ zu beginnen.
Der *SourceCodeExplorer* sollte erweitert werden, mit dem Ziel, Attribute oder auch Werte auswählbar zu machen. Dies könnte genutzt werden, um Anforderungen mit gleichen

Attributen oder auch Werten hervorzuheben. Angewendet werden könnte dies, um eine Form der Sichtenbildung anzubieten.

Die Darstellung von Beziehungen im 3D-Modell sollte ebenfalls ausgebaut werden. Ausgangspunkte sind die Darstellung unterschiedlicher Kontextbeziehungsklassen sowie die Darstellung im Falle einer hohen Anzahl gleichzeitig dargestellter Beziehungen, umsetzbar zum Beispiel in Form eines Straßennetzes. Auch interessant wäre die Darstellung von Beziehungen über mehrere Verbindungsebenen hinweg, um Auswirkungen von Änderungen nachzuvollziehen.

Nach ausführlichem Testen, Evaluieren und Weiterentwickeln wäre es generell auch sinnvoll zu prüfen, ob eine Integration des 3D-Modells in ein Requirements-Management-Softwareprodukt oder die Implementierung ähnlicher Funktionen in die Benutzeroberfläche des Prototyps vorteilhaft ist.

Das zugrunde liegende 3D-Modell kann als die Basis für viele weitere Forschungen im Kontext der Darstellung von Traceability Links sein. Traceability entsteht, wenn Artefakte eines Entwicklungsprozesses mittels Kontextbeziehungen verbunden werden (Cleland-Huang et al., 2014). Zu erforschen wären bspw. Möglichkeiten, zusätzliche Informationen zu Anforderungen einzubinden, seien es erklärende Modelle und Grafiken, entwickelte Tests und Codefragmente oder auch zu erfüllende Akzeptanzkriterien, Standards und Verträge.

Literaturverzeichnis

Abad, Z. S. H., Ruhe, G., & Noaeen, M. (2016). Requirements Engineering Visualization: A Systematic Literature Review. *Proceedings - 2016 IEEE 24th International Requirements Engineering Conference, RE 2016*, 6–15. https://doi.org/10.1109/RE.2016.61

A-Frame Documentation. (n.d.). Https://Aframe.Io/Docs/1.6.0.

A-Frame Text Component. (n.d.). Https://Aframe.Io/Docs/1.6.0/Components/Text.Html.

Aurum, A., & Wohlin, C. (2003). The fundamental nature of requirements engineering activities as a decision-making process. *Information and Software Technology*, *45*(14), 945–954. https://doi.org/10.1016/S0950-5849(03)00096-X

Baum, D., Schilbach, J., Kovacs, P., Eisenecker, U., & Muller, R. (2017). GETAVIZ: Generating Structural, Behavioral, and Evolutionary Views of Software Systems for Empirical Evaluation. *Proceedings - 2017 IEEE Working Conference on Software Visualization, VISSOFT 2017*, *2017-October*, 114–118. https://doi.org/10.1109/VISSOFT.2017.12

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, *18*(9), 509–517. https://doi.org/10.1145/361002.361007

Cooper, J. R., Lee, S.-W., Gandhi, R. A., & Gotel, O. (2009). Requirements Engineering Visualization: A Survey on the State-of-the-Art. *2009 Fourth International Workshop on Requirements Engineering Visualization*, 46–55. https://doi.org/10.1109/REV.2009.4

Diehl, S. (2005). Software visualization. *Proceedings - 27th International Conference on Software Engineering, ICSE05*, 718–719. https://doi.org/10.1145/1062455.1062634

Ebert, C., & Jastram, M. (2012). ReqIF: Seamless Requirements Interchange Format between Business Partners. *IEEE Software*, *29*(5), 82–87. https://doi.org/10.1109/MS.2012.121

ebroeker. (2021). *pyreqif*. Https://Github.Com/Ebroecker/Pyreqif.

Gotel, O. C. Z., Marchese, F. T., & Morris, S. J. (2008). The Potential for Synergy between Information Visualization and Software Engineering Visualization. *2008 12th International Conference Information Visualisation*, 547–552. https://doi.org/10.1109/IV.2008.56

Hamou-Lhadj, A., & Pirzadeh, H. (n.d.). *Software Visualization Techniques for the Representation and Exploration of Execution Traces with a Focus on Program Comprehension Tasks*. https://www.researchgate.net/publication/329929672

Hans Jurgen Ohlbach. (n.d.). *NP-Probleme*.

Hundhausen, C. D. (1997). A Meta-Study of Software Visualization Effectiveness. *Journal of Visual Languages and Computing*.

Lanza, M., Gall, H., & Dugerdil, P. (2009). EvoSpaces: Multi-dimensional Navigation Spaces for Software Evolution. *2009 13th European Conference on Software Maintenance and Reengineering*, 293–296. https://doi.org/10.1109/CSMR.2009.14

layout-bmfont-text. (n.d.). Https://Github.Com/Experience-Monks/Layout-Bmfont-Text.

Lisa Vogelsberg. (2018). *Entwurf eines Datenmodells zur Speicherung von Softwarevisualisierungsartefakten*.

Michael Jastram. (n.d.). *ReqIF Quick Reference v.1.2* . Https://Www.Reqif.Academy/Reference/Reqif-Quick-Reference/.

Müller, R., & Zeckzer, D. (2015). The recursive disk metaphor: A glyph-based approach for software visualization. *IVAPP 2015 - 6th International Conference on Information Visualization Theory and Applications; VISIGRAPP, Proceedings*, 171–176. https://doi.org/10.5220/0005342701710176

Neo4J - Breaking Changes. (n.d.). Https://Neo4j.Com/Docs/Upgrade-Migration-Guide/Current/Version-4/Migration/Drivers/Breaking-Changes/.

Neo4J - Supported Versions. (n.d.). Https://Neo4j.Com/Developer/Kb/Neo4j-Supported-Versions/.

Object Management Group. (2016, June). *Requirements Interchange Format (ReqIF)*.

Pohl, K. (2008). *Requirements Engineering: Grundlagen, Prinzipien, Techniken* (Vol. 2). dpunkt-Verlag.

prostep ivip. (2021). *ReqIF Implementation Guide*. https://www.ps-ent-2023.de/fileadmin/prod-download/prostep-ivip_ImplementationGuide_ReqIF_V1-8.pdf

prostep ivip. (2024, July). *Recommendation PSI 18 V2.2*. Https://Www.Prostep.Org/Fileadmin/Prod-Pay-Download-8c1d/Recommendation_ReqIF_V2.2.Pdf.

tiluhlig. (2022). *TReqz*. Https://Github.Com/Tilluhlig/TReqz.

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

Wettel, R., & Lanza, M. (2007). Program comprehension through software habitability. *IEEE International Conference on Program Comprehension*, 231–240. https://doi.org/10.1109/ICPC.2007.30

Wettel, R., Lanza, M., & Robbes, R. (2010). *Università della Svizzera italiana USI Technical Report Series in Informatics Empirical Validation of CodeCity: A Controlled Experiment*. http://codecity.inf.usi.ch

Wilde, T., & Hess, T. (2007). Forschungsmethoden der Wirtschaftsinformatik. *WIRTSCHAFTSINFORMATIK*, *49*(4), 280–287. https://doi.org/10.1007/s11576-007-0064-z

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit mit dem Titel "Visualisierung von Anforderungsdokumenten" selbständig und ohne unerlaubte Hilfe angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Leipzig, den 12. November 2024

Benjamin Gahnz