Universität Leipzig

Wirtschaftswissenschaftliche Fakultät

Institut für Wirtschaftsinformatik

Professur Softwareentwicklung für Wirtschaft und Verwaltung

Prof. Dr. Ulrich W. Eisenecker

David Baum

Thema

Entwurf eines Datenmodells zur Speicherung von Softwarevisualisierungsartefakten

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science – Informatik

Universität Leipzig
Fakultät für Mathematik und Informatik

vorgelegt von: Lisa Vogelsberg

Leipzig, den 27. September 2018

Abstract

Das System Getaviz der Forschungsgruppe *Visual Software Analytics* des Instituts für Wirtschaftsinformatik der Universität Leipzig stellt mehrere Werkzeuge zur Erzeugung und Analyse von Softwarevisualisierungen bereit. Da sich der Einsatz des Frameworks Xtext zur Generierung einer Visualisierung bezüglich Flexibilität und Speicherperformance als problematisch herausgestellt hat, wurde untersucht wie eine Ablösung dieser Technologie in Zukunft erreicht werden kann.

Zu diesem Zweck wurde ein Datenmodell entwickelt, welches es ermöglicht, Softwarevisualisierungsartefakte mittels Neo4j in einer Datenbank abzulegen und von dort wieder abzurufen. Der Entwurf dieses Datenmodells basiert auf einer vorangegangenen Analyse der bisher genutzten Modelle für die Metaphern City und Recursive Disk, um deren Bestandteile auf die neuen Modelle abzubilden. Um mögliche Auswirkungen einer Migration auf die Performance und Flexibilität des Transformationsprozesses untersuchen zu können, wurde anschließend das Datenmodell und der Zugriff auf die Datenbank innerhalb des Generators in Form von zusätzlichen Komponenten implementiert. Dadurch konnte erreicht werden, Transformationen ohne den Einsatz von Xtext durchzuführen. Als Eingabemodell wird ein von jQAssistant generierter Graph verwendet, der die Struktur des Softwaresystems abbildet.

Inhaltsverzeichnis

Inhaltsverzeichnis I

Abbildungsverzeichnis II

Tabellenverzeichnis III

Listings IV

Abkürzungsverzeichnis V

1 Einleitung 1
1.1 Motiviation und Problemstellung 1
1.2 Zielstellung 1
1.3 Aufbau der Arbeit 2

2 Grundlagen 3
2.1 Getaviz 3
2.1.1 Extraktoren 3
2.1.2 Generator 4
2.2 Xtext und Modeling Workflow Engine 2 4
2.3 Neo4j 7
2.4 jQAssisstant 8

3 Analyse der Modelle 9
3.1 Modelle zur Beschreibung der Struktur 9
3.1.1 FAMIX 9
3.1.2 Datenmodell von jQAssisstant 9
3.2 City 14
3.3 Recursive Disk 16

4 Neo4j-Integration 18
4.1 Konzipierung des Datenmodells 18
4.2 Auswahl der Migrationsstrategie 21
4.3 Implementierung 24
4.3.1 Allgemeine Beschreibung 24
4.3.2 Auswirkungen der erfolgten Migration auf den Transformationsprozess 26

5 Zusammenfassung und Ausblick 29

Literaturverzeichnis 31

Abbildungsverzeichnis

2.1 Komponenten der Famix2City- und Famix2RD-Workflows 6

3.1 Reduzierte Metamodell-Hierarchie von FAMIX 3.0 10
3.2 Reduzierter jQAssisstant (jQA)-Graph 11
3.3 Recursive Disk (RD) Konfigurationszusammenstellung. Der rot markierte Pfad stellt die Standardkonfiguration dar 17

4.1 Überblick über die Verbindung der Visualisierungsknoten mit den Elementen aus dem jQA-Graph 20
4.2 RD-Teilgraph 22
4.3 City-Teilgraph 23

Tabellenverzeichnis

3.1 Mapping von FAMIX-Namespaces auf jQA-Knoten mit dem Label Package 12
3.2 Mapping von FAMIX-Classes und FAMIX-ParameterizableClasses auf jQA-Knoten mit den Labels Class oder Interface 12
3.3 Mapping von FAMIX-Enums auf jQA-Knoten mit dem Label Enum 12
3.4 Mapping von FAMIX-AnnotationTypes auf jQA-Knoten mit dem Label Annotation 12
3.5 Mapping von FAMIX-Methods auf jQA-Knoten mit dem Label Method . . 13
3.6 Mapping von FAMIX-Attributes auf jQA-Knoten mit dem Label Field . . . 14
3.7 Mapping von FAMIX-EnumValues auf jQA-Knoten mit dem Label Field . 14
3.8 Mapping von FAMIX-LocalVariables auf jQA-Knoten mit dem Label Variable 14
3.9 Mapping von FAMIX-Parameters auf jQA-Knoten mit dem Label Parameter 15
3.10 Mapping von FAMIX-Invocations auf jQA-Relationen mit dem Namen INVOKES 15
3.11 Mapping von FAMIX-Accesses auf jQA-Relationen mit den Namen READS oder WRITES 15
3.12 Mapping von FAMIX-Inheritances auf jQA-Relationen mit den Namen EXTENDS oder IMPLEMENTS 15

Listings

2.1 Beispiel zur Belegung von Slots in Workflows 4
2.2 Beispiel für den Zugriff auf einen Workflow-Slot 5

4.1 Cypher-Statement zur Rückgabe von Getter-Methoden 24
4.2 Cypher-Statement zur Rückgabe von primitiven Typen 25
4.3 Verwendung des ResourceIterator 26
4.4 Funktion zur Rückgabe des Endknoten eines bestimmten Labels zu einer gegebenen Relationship 27

Abkürzungsverzeichnis

jQA	jQAssisstant
mwe2	Modeling Workflow Engine
RD	Recursive Disk

1 Einleitung

1.1 Motiviation und Problemstellung

Die Nutzung von Software über einen langen Zeitraum erfordert eine stetige Weiterentwicklung und Wartung des Systems. Dafür ist es nötig, dieses System analysieren und verstehen zu können. Die Dokumentation ist dabei ein hilfreiches Mittel, um dieses Ziel zu erreichen. Eine Möglichkeit der Dokumentation bietet die Softwarevisualisierung, welche die Struktur, das Verhalten und die Historie von Software grafisch darstellt. Einige der möglichen Formen zur Visualisierung werden mit Hilfe des Systems Getaviz[1] der Forschungsgruppe *Visual Software Analytics* des Instituts für Wirtschaftsinformatik der Universität Leipzig untersucht und entwickelt. Neben weiteren Werkzeugen beinhaltet Getaviz einen Generator, der über eine Folge von Transformationsschritten Visualisierungen für verschiedene Metaphern wie City oder RD erstellt. Die Transformation – beginnend beim Auslesen des Quelltextes bis hin zur fertigen Visualisierung – findet modellgetrieben statt, sodass mit jedem Schritt ein neues Modell erzeugt wird. Zur Verarbeitung der einzelnen Modelle wird derzeit das Framework Xtext – ein Teil des Eclipse-Modelling-Frameworks – verwendet. Die Verwendung dieses Frameworks geht einher mit der Verwendung von Workflows der Modeling Workflow Engine (mwe2), in denen die Abfolge der einzelnen Transformationsschritte festgelegt wird.
Da die Schwierigkeit für das Verständnis eines Systems mit dessen Größe skaliert, besteht ein besonderes Interesse, Visualisierungen unabhängig von der Größe des Quellcodes erstellen zu können. Der Einsatz von Xtext ist hierbei jedoch problematisch, da die Verarbeitung der einzelnen Modelle zu einem sehr hohen Arbeitsspeicherverbrauch führt, da jedes erstellte Modell während des gesamten Transformationspozesses vollständig im Speicher gehalten wird (vgl. [Baum et al. 2017]). Zusätzlich schränkt die Verwendung der Workflows die Flexibilität der Transformationen ein, da für jede gewünschte Transformation ein eigener Workflow angelegt werden muss, wodurch es nicht möglich ist, eine einzige flexible Datei zu definieren, die beispielsweise anhand einer Konfiguration unterschiedliche Folgen von Transformationsschritten auslöst. Des Weiteren ist es nicht möglich, Transformationsfolgen, die für mehrere Transformationen benötigt werden, wiederzuverwenden. Auch die Xtext-Modelle selbst bieten wenig Flexibilität bei der Formulierung von Abfragen.

1.2 Zielstellung

Um diese Einschränkungen zu lösen, gilt es zu untersuchen, wie eine schrittweise Ablösung von Xtext und mwe2 ermöglicht werden kann. Ziel der Arbeit ist es daher, über die Verwendung der Graphdatenbank Neo4j die Modelle, die während einer Transformation erzeugt werden, in einer Datenbank zu speichern und von dort wieder auszulesen. Zu diesem Zweck wird ein Datenmodell zur Speicherung des Strukturmodells und für die Modelle der Metaphern Recursive Disk und City konzipiert und anschließend prototypisch implementiert. Dadurch

[1] https://github.com/softvis-research/Getaviz

werden die Modelle nicht mehr im Speicher gehalten, da sie physisch auf der Festplatte gespeichert werden. Da nach jeder Ausführung einer Komponente ein (neuer) persistenter Zustand der Datenbank existiert, wird das Ausführen einzelner Komponenten ermöglicht, ohne, dass diese an einen Workflow gebunden sind. Da die Modelle neben einfachen Beziehungen auch Hierarchien von Beziehungen abbilden, bietet die Repräsentation als Graph durch die von Neo4j bereitgestellten Zugriffswege effiziente Möglichkeiten der Formulierung von Abfragen über rekursive Beziehungen, sowohl in Bezug auf Rechenzeit als auch Formulierung (vgl. [Vukotic 2015, S.13]). Weitere Vorteile des Einsatzes von Neo4j ergeben sich durch eine einfache Modellierung und somit Anschaulichkeit. Zudem bietet Neo4j eine gute Anbindung und API für Java und wird bereits von Teilen des Systems verwendet. In Bezug auf das Strukturmodell gilt es zu beachten, die Modelle jQAssisstant-kompatibel zu halten. jQAssistant (jQA) ist ein Werkzeug zur Validierung von Software und wird ebenfalls bereits vom System eingesetzt, um Quelltexte von Softwareprojekten einzulesen und eine Repräsentation der Struktur in einer Neo4j-Datenbank abzulegen. Damit die Migration schrittweise erfolgen kann, wird die Nutzung der Datenbank zunächst als zusätzliche Alternative implementiert, sodass weiterhin auch Xtext und Workflows verwendet werden können. Wird die Implementierung künftig ausreichend verfeinert und getestet, können auf Grundlage des Datenmodells und der Implementierung weitere Metaphern umgesetzt und Workflows abgesetzt werden. Dies ist jedoch nicht mehr Bestandteil dieser Arbeit.

1.3 Aufbau der Arbeit

Im Anschluss werden in Kapitel 2 die Grundlagen vorgestellt, auf denen die Implementierung aufbaut. In den ersten Abschnitten wird hierfür eine Einleitung zu Getaviz und dessen Bestandteile gegeben. Abgeschlossen wird das Kapitel, indem die für diese Arbeit relevanten Technologien vorgestellt werden. Kapitel 3 beschäftigt sich mit der Analyse der Modelle, um eine Basis für den Entwurf des neuen Datenmodells zur Verfügung zu stellen. In Abschnitt 3.1 werden zunächst das Modell FAMIX und das Datenmodell von jQAssisstant untersucht und miteinander verglichen, um festzustellen, welches der Modelle zur Beschreibung der Struktur verwendet wird. Die Abschnitte 3.2 und 3.3 beinhalten die Analyse zu den Visualisierungsmodellen City und RD. Kapitel 4 befasst sich dann mit der Integration von Neo4j innerhalb des Generators. Dafür werden zu Beginn in Abschnitt 4.1 die gewonnenen Erkenntnisse aus der Analyse genutzt, um das Datenmodell zu entwerfen und vorzustellen. In Abschnitt 4.2 werden dann zwei Strategien diskutiert, wie das entworfene Datenmodell integriert werden kann, bevor in Abschnitt 4.3 die Vorgehensweise der Implementierung für die gewählte Strategie genauer beschrieben wird und welche Auswirkungen sich aufgrund der Implementierung ergeben. Im letzten Kapitel der Arbeit werden die Ergebnisse noch einmal zusammengefasst und ein Ausblick über künftige Erweiterungen und Verbesserungen gegeben.

2 Grundlagen

Bevor das Datenmodell entworfen und implementiert wird, sollte sich zunächst in die verwendeten Technologien eingearbeitet werden. Zu diesem Zweck erfolgt eine Einführung in das Toolset Getaviz, innerhalb dessen die Implementierung erfolgt. Anschließend wird das Framework, das in Getaviz zur Verarbeitung der Modelle eingesetzt wird, betrachtet. Danach erfolgt eine Einführung in das verwendete Datenbanksystem, bevor eine Übersicht zu jQAssisstant gegeben wird, dessen Datenmodell bei der Konzipierung des Graphen zusätzlich Beachtung findet.

2.1 Getaviz

Getaviz ist ein Toolset, das von der Forschungsgruppe "Visual Software Analytics" des Instituts für Wirtschaftsinformatik der Universität Leipzig entwickelt wird und es ermöglicht, Softwarevisualisierungen zu entwickeln, zu erstellen und zu analysieren. (vgl. [Baum et al. 2017]) Die bisher implementierten Visualisierungen decken sowohl eine Sicht auf Strukturaspekte, als auch Verhaltens- und Entwicklungsaspekte einer Software ab. Zu den enthaltenen Werkzeugen gehören zunächst Extraktoren, die ein Softwareprojekt einlesen und die drei zuvor genannten Aspekte in jeweils einem Modell festhalten. Das zweite Werkzeug, der Generator, entnimmt anschließend dem Modell alle nötigen Informationen, um eine Visualisierung zu erzeugen. Das dritte Werkzeug stellt ein User Interface bereit, das über den Browser aufrufbar ist und Interaktionsmöglichkeiten für das Modell bereitstellt. Das letzte Werkzeug ist ein Evaluierungsserver, der – verknüpft mit dem User Interface – den Prozess der Evaluierung einer Visualisierung unterstützt. Da lediglich die ersten beiden Werkzeuge für diese Arbeit von Relevanz sind, werden diese im Folgenden vorgestellt.

2.1.1 Extraktoren

Bevor die Visualisierung einer Software erzeugt wird, werden ihre Bestandteile über einen Extraktor analysiert und in ein Modell überführt. Neben Extraktoren zur Ezeugung eines HISMO-Modells für Entwicklungsaspekte oder eines DYNAMIX-Modell für Verhaltensaspekte, existieren insbesondere sprachspezfische Extraktoren zur Überführung in ein FAMIX-Modell zur Beschreibung struktureller Informationen. Unterstützt werden bisher die Sprachen Java, C# und Ruby. Nach der Überführung stehen die Modelle jeweils als Textdokument zur Verfügung.
Nicht Teil des Toolsets, aber in Anwendung für Getaviz, befindet sich ein Parser für Quelltexte, um Stukturaspekte in einer Graphdatenbank zu hinterlegen. Näheres dazu wird in Abschnitt 2.4 zu jQAssisstant beschrieben. Eine Untersuchung des zugehörigen Datenmodells im Zusammenhang mit FAMIX erfolgt dann in Abschnitt 3.1.

2.1.2 Generator

Wurden die Informationen über das System extrahiert und in einer Datei gespeichert, wird diese für den Generator als Eingabe genutzt, um eine Visualisierung zu generieren. Hierfür steht eine Auswahl an Metaphern mit verschiedenen Varianten zur Verfügung. Neben den für diese Arbeit relevanten Metaphern City und RD sind dies noch Plant und Multisphere. Der Prozess der gesamten Transformation besteht aus einer Folge von mehreren Transformationsschritten, an deren Ende eine Visulisierung im X3D-, X3DOM- oder A-Frame-Format steht. Der Generator verfolgt dabei einen modellgetriebenen Ansatz, sodass mit jedem Transformationsschritt jeweils ein Modell eingelesen und daraus wieder ein neues Modell für den nächsten Transformationsschritt erzeugt wird (vgl. [Müller 2015]). Die Modelle werden über Xtext generiert und die Transformation über sogenannte Workflows der Modeling Workflow Engine 2 gesteuert.

2.2 Xtext und Modeling Workflow Engine 2

Bei Xtext handelt es sich um ein Framework zur Entwicklung von domänspezifischen Sprachen. Es ist ein Bestandteil des Eclipse Modeling Frameworks (EMF) und liefert neben einem Parser und einem Texteditor ein Eclipse-Plugin für die erstellte Sprache. Dafür wird eine neue Eclipse-Instanz erzeugt, die mit der Sprache arbeiten kann, wodurch zum Beispiel Syntax-Highlighting oder Codevervollständigung unterstützt werden (vgl. [*Xtext Documentation* 2018]). So gibt es für jedes Modell, insbesondere für die in dieser Arbeit relevanten Modelle FAMIX, City und Recursive Disk jeweils eine Xtext-Datei, die sie beschreibt. Über die Modeling Workflow Engine, die selbst eine domän-spezifische Sprache darstellt, werden die erstellten Sprachen beziehungsweise Modelle konfiguriert (vgl. [*Xtext Documentation* 2018]). Der Hauptaspekt von mwe2 besteht in der Definition von Workflows, welche aus mehreren Komponenten bestehen, über die Aufgaben wie das Einlesen von Ressourcen, Transformation oder das Erzeugen von Outputs angesteuert werden können. Die Komponenten eines Workflows werden in der Reihenfolge ausgeführt, in der sie in der Datei angelegt werden. Um die Modelle allen Komponenten eines Workflows zur Verfügung zu stellen, werden diese in Slots mit einem Key hinterlegt, über den sie wieder aufgerufen werden können. Listing 2.1 zeigt den Ausschnitt eines Workflows, indem ein Slot für das FAMIX-Model über den Key “famix“ bereitgestellt wird, um in der Komponente Famix2City auf das FAMIX-Modell zugreifen zu können. In Listing 2.2 Zeile 6 ist zu sehen, wie das Modell über den Key ausgelesen wird.

```
1  //...
2  component = org.svis.generator.city.s2m.Famix2City {
3     modelSlot = "famix"
4  }
5  //...
```

Listing 2.1: Beispiel zur Belegung von Slots in Workflows

```
1 //...
2 override protected invokeInternal(WorkflowContext ctx, ProgressMonitor
     monitor, Issues issues) {
3     log.info("Famix2City has started.")
4
5     // receive FAMIX-model from source
6     famixDocument = (ctx.get("famix") as org.svis.xtext.famix.Root).
         document
7     //...
```

Listing 2.2: Beispiel für den Zugriff auf einen Workflow-Slot

Workflows zur Erzeugung der City- und Recursive Disk-Visualisierung

So sind innerhalb des Generators mehrere dieser Workflows für die verschiedenen Metaphern und Software-Aspekte angelegt. Um im späteren Verlauf der Arbeit besser darzustellen, an welchen Stellen die Integration der Datenbank erfolgt, werden kurz die zwei bestehenden Workflows vorgestellt, die die Transformationen ausgehend von der Strukturbeschreibung bis hin zur Visualisierung für City und RD festlegen. Dies liefert einen Überblick über die Komponenten, in denen bisher Xtext-Modelle verarbeitet werden.

Famix2City

Zu Beginn des Workflows wird eine FAMIX-Datei eingelesen, um daraus das FAMIX-Xtext-Modell zu generieren und es der Komponente Famix2Famix zur Verfügung zu stellen. Diese wird anschließend aufgerufen und liest das generierte Modell aus, um nicht benötigte Elemente zu filtern und weitere Ergänzungen vorzunehmen. Während dieses Schrittes werden z.B. Attribute ergänzt, die nicht direkt aus der FAMIX-Datei ausgelesen werden. Ist die Transformation des FAMIX-Modells abgeschlossen, wird das neu erzeugte Modell an die nächste Komponente Famix2City weitergereicht. Wie zuvor wird nun das FAMIX-Modell ausgelesen und auf die Elemente des City-Modells abgebildet. Dabei werden nur Attribute gesetzt, die Metadaten darstellen und noch nicht formgebend sind. Am Ende dieser Transformation steht ein City-Xtext Modell, das an die Komponente City2City weitergereicht wird. Bevor diese Komponente allerdings ausgeführt wird, gibt es noch einen Zwischenschritt, bei dem mit Hilfe der Komponente Famix2JSON aus dem FAMIX-Modell Metadaten in einem JSON-Dokument gespeichert werden. Dieser Schritt ist nicht primär für die Erstellung der Visualisierung innerhalb des Generators nötig und erzeugt euch kein neues Xtext-Modell, sondern lediglich die Ausgabedatei. Die Metadaten werden aber beispielsweise für die Interaktion mit dem Modell innerhalb des User Interfaces benötigt. City2City erfüllt eine ähnliche Aufgabe wie Famix2Famix. Hier wird das bereits erzeugte City-Xtext-Modell ausgelesen und verfeinert. Dazu gehört vor allem das Setzen aller Attribute, die den Entitäten ihre Form, Farbe und Anordnung geben. Ist das City-Modell vollständig erzeugt, wird je nach gewähltem Ausgabeformat die entsprechende Klasse aufgerufen, die das Modell in das gewählte Format überführt.

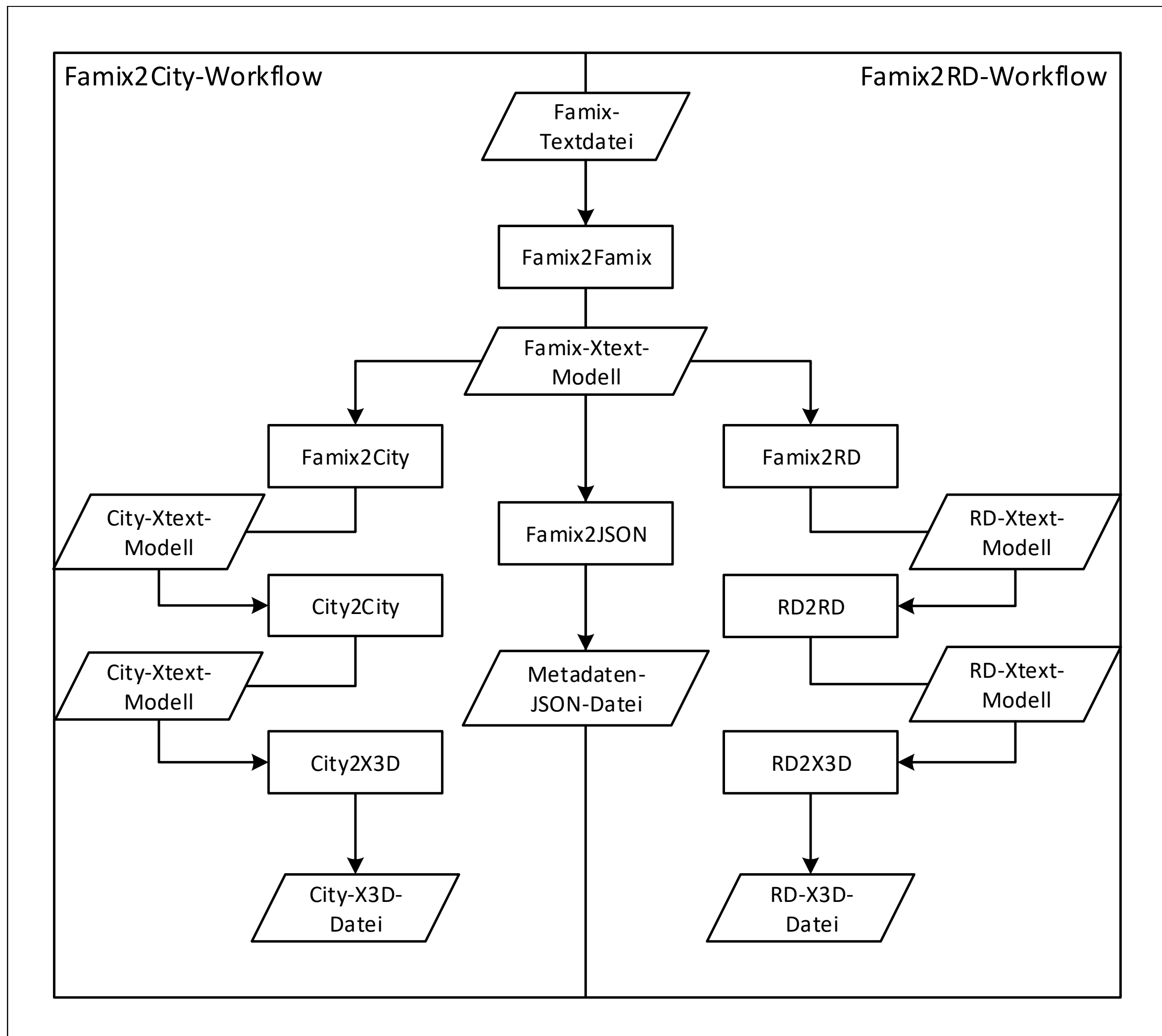

Abbildung 2.1: Komponenten der Famix2City- und Famix2RD-Workflows

Famix2RD

Die Schritte beginnend vom Einlesen der FAMIX-Datei bis zur Transformation des FAMIX-Xtext-Modells sind genauso definiert wie auch beim Famix2City-Workflow. Analog zu Famix2City und City2City werden dann die Komponenten Famix2RD und RD2RD aufgerufen. Sie übernehmen die gleichen Aufgaben wie schon jene des City-Workflows mit dem Unterschied, dass bereits in Famix2RD einige Attribute betreffend des Aussehens des Elements gesetzt werden, die aus einer Konfigurationsdatei entnommen werden. Danach erfolgt ebenso die Generierung des JSON-Dokuments und je nach gewählten Ausgabeformat generiert die entsprechende Komponente aus dem RD-Xtext-Modell den Output.

Beide Workflows haben gemeinsam, dass jede verwendete Komponente folgende drei Schritte ausführt:

1. Auslesen eines Modells aus einem Slot und Sammeln der Elemente in Listen
2. Verarbeitung der Elemente durch Iteration über die Listen
3. Weiterreichen des Modells oder Erzeugen einer Ausgabe-Datei

Abbildung 2.1 fasst den Ablauf der Komponenten und die erzeugten Artefakte grafisch noch einmal zusammen.

2.3 Neo4j

Neo4j ist ein graphbasiertes Datenbankmanagementsystem und der populärste Vertreter. Daten werden in Form von Knoten und Kanten eines Graphs gespeichert. Im Folgenden werden in Entsprechung zur Bezeichnung in Neo4j Knoten als Nodes bezeichnet und Kanten als Relationships. Letztere stellen also eine Beziehung zwischen Nodes dar. Um Nodes näher zu beschreiben, können diese mit Attributen versehen werden, die aus Key-Value-Paaren bestehen. Mit Hilfe von Labels ist es möglich, sowohl Nodes als auch Relationships mehreren Kategorien zuzuordnen, sodass Labels dadurch für die Suche im Graphen verwendet werden können, um eine bestimmte Gruppe von Nodes oder Relationships zu selektieren. Um Graphen zu erstellen, zu ändern oder zu durchsuchen, bietet Neo4j die eigene deklarative Sprache Cypher an. Ein Ziel bei der Entwicklung war nach eigenen Angaben die intutive Verständlichkeit für den Lesenden, sodass eine Abfrage auch mit geringen oder bestenfalls keinen Vorkenntnissen in Neo4j verstanden werden kann. (vgl. [Neo4j Graph Database Project 2018]) Sprachlich wurden deshalb auch Anleihen an SQL genutzt, wie zum Beispiel die Verwendung von WHERE-Klauseln zum Filtern von Ergebnissen oder andere Schlüsselwörter wie ORDER BY zum Sortieren. Über eine mitgelieferte Browser-Anwendung können bereits vorgegebene Cypher-Statements genutzt oder manuell eingegeben werden. Die Anzeige des Ergebnisses kann in grafischer oder textueller Form gewählt werden. Eine weitere Möglichkeit ist die Verwendung der API, die Interfaces und Funktionen für den Zugriff und die Manipulation des der Datenbank bereitstellt, es aber auch ermöglicht, Cypher-Statements auszuführen.
Bevor auf die Datenbank mit dem Graphen zugegriffen werden kann, muss diese für den Transformationsprozess ausgewählt und angebunden werden. Dafür gibt es zunächst zwei wesentliche Wege: den *embedded mode* und den *server mode*. *Embedded mode* bedeutet, dass die Datenbank und die sie nutzende Anwendung eine Einheit bilden. Der Begriff bedeutet allerdings nicht, dass es um die Einbettung der Datenbank geht, sondern er resultiert vielmehr aus der direkten Verwendung der Neo4j-API aus einer JVM-basierten Anwendung heraus (vgl. [Vukotic 2015, S.186]). Neo4j-Code und die Anwendung laufen also in der selben JVM. Im Gegensatz dazu läuft die Datenbank im *server mode* innerhalb eines eigenen Prozesses und wird über eine HTTP-basierte REST-API angesprochen. Vorteilhaft ist dies zum Beispiel für die Nutzung innerhalb eines Netzwerks, wobei zu beachten ist, dass im Vergleich zum *embedded mode* eine niedrigere Performance aufgrund von Latenzzeiten des Netzwerks auftreten kann (vgl. [Vukotic 2015, S.184]).
Um Neo4j in Getaviz für die Transformation zu nutzen, kann die bereits implementierte Klasse `Database` des Generators verwendet werden, die den embedded mode nutzt. Außerdem implementiert sie das Singleton-Entwurfsmuster und sichert so ab, dass genau eine Instanz der Datenbank zurückgegeben wird. Dafür stellt sie die statische Methode `getInstance` zur Verfügung, über deren Parameter der Pfad der Datenbank angegeben

wird. Die Angabe des Pfades wird in einer separaten Konfigurationsdatei über die Einstellung `database_name` vorgenommen, wodurch er über ein zugehöriges globales Konfigurationsobjekt wieder ausgelesen werden kann. Initialisiert wird die Datenbank mit Hilfe der Funktion `newEmbeddedDatabase` der Neo4j-Klasse `GraphDatabaseFactory` und in Form eines `GraphDatabaseService`-Objekts zurückgegeben. Über das zurückgegebene `GraphDatabaseService`-Objekt lassen sich nun alle Operationen, die die Datenbank betreffen, umsetzen – vom Öffnen und Beenden einer Transaktion bishin zum Suchen, Verändern und Speichern von Daten.

2.4 jQAssistant

jQAssistant ist ein Werkzeug zur Validierung von Software. Es scannt Software, um unter anderem ihre Struktur zu analysieren und hinterlegt die gewonnen Strukturinformation in einer Neo4j-Datenbank. In Getaviz wird jQAssistant bereits als Möglichkeit genutzt, um Repräsentationen von Java-Systemen in solch einer Datenbank abzulegen. Dabei werden Elemente wie Pakete, Klassen, Attribute oder Methoden als Knoten gespeichert und mit Kanten verbunden, die für Beziehungen wie Vererbungen, Aufrufe oder Enthaltensein stehen. Auf weitere Details des grundlegenden Datenmodells wird in Kapitel 3 zur Analyse der Modelle näher eingegangen. Weiterhin existieren zahlreiche Plugins, um die gescannte Struktur mit weiteren Elementen und Informationen anzureichern, zum Beispiel für JUnit und JSON-Dateien. Durch die Nutzung von Neo4j kann dessen Abfragesprache Cypher genutzt werden, um Informationen über die Software abzufragen. Diese Abfragen können genutzt werden, um Regeln für das System zu formulieren und diese in einer XML-Datei zu hinterlegen, um sie bei jedem Buildprozess zu überprüfen. Dies kann dazu genutzt werden, Warnungen zu generieren oder sogar den laufenden Buildprozess abzubrechen, wenn eine Architekturverletzung aufgetreten ist.

3 Analyse der Modelle

3.1 Modelle zur Beschreibung der Struktur

3.1.1 FAMIX

Um zunächst die Struktur des zu analysierenden Systems zu beschreiben, wird die Metamodell-Familie FAMIX verwendet. Das Kernstück bildet ein Metamodell zur sprachunabhängigen Beschreibung objektorientierter Software (vgl. [Ducasse et al. 2011]). In diesem Kernmodell sind typische Entitäten eines Systems wie *Class*, *Namespace*, *Attribute*, *Method* oder Beziehungen wie *Invocation*, *Inheritance* und *Access* definiert. Das Xtext-Modell in Getaviz beinhaltet weitestgehend die Entitäten des Kerns, fügt aber auch FAMIX-Entitäten aus dem Java-Umfeld wie *Enum*, *EnumValue* oder *ParameterizableClass* hinzu, die nicht mehr zum Kern gehören. Abbildung 3.1 zeigt einen Ausschnitt der FAMIX-Hierarchie. Alle unterstrichenen Entitäten werden im Xtext-Modell für FAMIX verwendet, wobei die Hierarchie nicht mit abgebildet wird. Außerdem wurde zusätzlich die Entität *Structure* hinzugefügt, die als Oberklasse für *Class*, *ParameterizableClass*, *Enum* und *AnnotationType* fungiert. Dies erleichtert an vielen Stellen den Transformationsprozess, da diese vier Elemente häufig gleich verarbeitet werden, wodurch nicht für jeden dieser Typen eine eigene Funktion notwendig ist. Structure wird – so wie die restlichen verwendeten Entitäten – der Oberklasse Element untergeordnet, die somit das Äquivalent zu Entity darstellt. Somit hält sich das verwendete Xtext-Modell noch recht nah an die ursprüngliche Definiton. Bei der Verwendung der Attribute wurde das Modell wesentlich freier von der Vorlage definiert. Da das Modell außerdem viele Elemente definiert, die innerhalb des Generators bisher nicht weiterverwendet werden, werden für die Modellierung nur die Elemente betrachtet, die auch tatsächlich verarbeitet werden, um den Fokus etwas enger zu ziehen. Die übrig gebliebenen Elemente sind somit *Namespace*, *Class*, *ParameterizableClass*, *Enum*, *AnnotationType*, *Method*, *Attribute*, *EnumValue*, *LocalVariable*, *Parameter*, *Type*, *Parameter*, *PrimitiveType*, *ParameterizedType*, *Invocation*, *Access* und *Inheritance*.

3.1.2 Datenmodell von jQAssisstant

Durch den Scan von jQAssisstant wird ein Graph in Neo4j erstellt, der ebenso eine Beschreibung der Struktur zur Verfügung stellt. Somit ist zu analysieren, inwiefern sich der Informationsgehalt des Graphen mit dem des FAMIX-Modells deckt, ob also eine totale Abbildung vom FAMIX-Modell zum jQA-Modell möglich ist. Diese Analyse liefert dann eine Hilfestellung bei der Beantwortung der Frage, ob das FAMIX-Modell durch das jQA-Modell ersetzt werden kann oder das FAMIX-Modell eine eigene Graph-Repräsentation erhalten sollte. Der Typ einer Entität wird in jQA über entsprechende Label am Knoten angegeben. Die relevanten Typen von Knoten und deren Beziehungen zueinander können der Abbildung 3.2 entnommen werden. Eine vollständige Auflistung der Elemente und Properties kann der

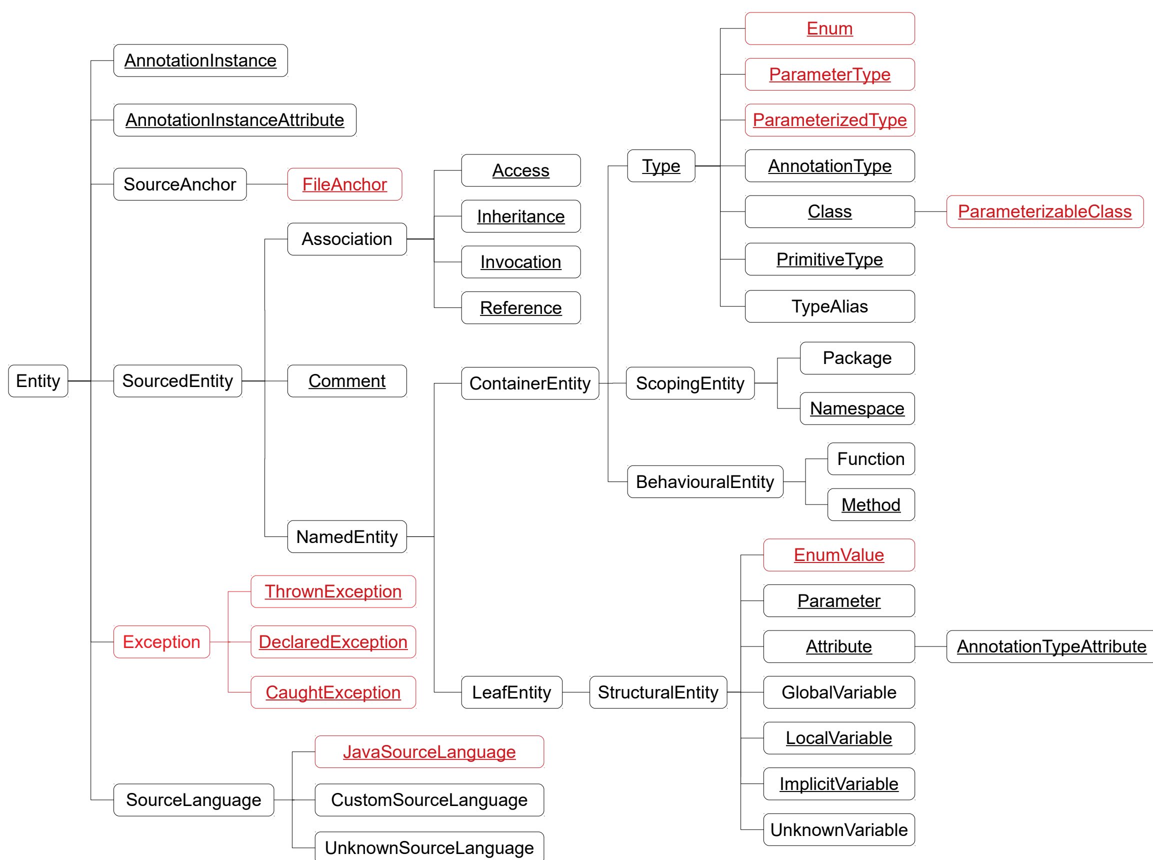

Abbildung 3.1: Reduzierte Metamodell-Hierarchie von FAMIX 3.0 (in Anlehnung an [Ducasse et al. 2011, S.13]) Schwarz sind alle Kernentitäten. Rot markierte Entitäten wurden den zusätzlichen Packages FAMIX-Java und FAMIX-Source-Anchor entnommen. Unterstrichene Entitäten sind im Xtext-Modell enthalten.

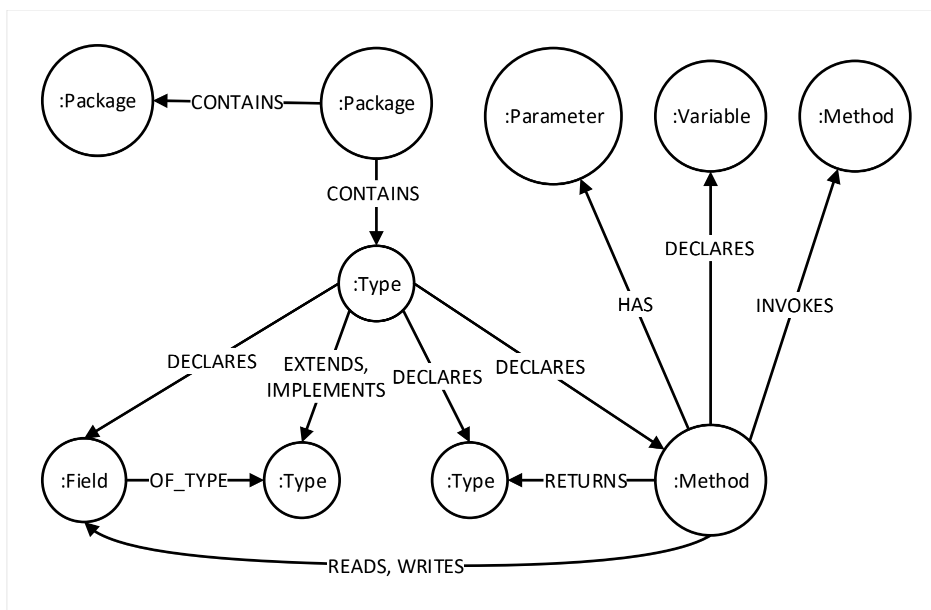

Abbildung 3.2: Reduzierter jQA-Graph

Dokumentation von jQAssistant entnommen werden[1]. Jeder Node des Standardplugins für Java besitzt zusätzlich auch noch das Label Java, das für jede Entität in den Betrachtungen allerdings weggelassen und implizit angenommen wird. Nodes mit dem Label Type werden weiter spezifiziert durch das Hinzufügen des Labels Class, Interface, Enum oder Annotation. Knoten ohne eine weitere Spezifizierung sind Typen, die von Java und externen Bibliotheken bereitgestellt wurden. Konstruktoren erhalten zusätzlich zum Label `Method` das Label `Constructor`.

Da nun die Entitäten beider Modelle bekannt sind, können diese schrittweise verglichen werden. Für jedes Element aus dem FAMIX-Modell wird zunächst die passende Entität aus dem jQA-Modell gesucht, auf die es abgebildet werden kann. Die Bezeichnung erfolgt in der Angabe der entsprechenden Labels. Danach wurde für jedes Attribut eines FAMIX-Elements geprüft, ob und wo die entsprechende Information im jQA-Graph zu finden ist. Häufig werden diese durch Properties eines Knotens oder über die Beziehung zu einem anderen Knoten gegeben.

Anmerkung zu Tabelle 3.2: Handelt es sich um eine innere Klasse, decken sich `name` und `fqn` nicht mehr mit den Attributen aus dem FAMIX-Modell. Name besteht dann aus dem Namen der übergeordneten Klasse gefolgt von einem Dollarzeichen und dem eigentlichen Namen der Klasse. Beim `fqn` verhält es sich ähnlich, indem statt eines Punktes ein Dollarzeichen vor dem Namen der inneren Klasse eingefügt wird.

Anmerkung zu Tabelle 3.5: Handelt es sich um einen Konstruktor, wird an Stelle des richtigen Namens in jQA der Ausdruck <*init*> verwendet sowohl für `name` als auch innerhalb von `signature`. Im Gegensatz zum FAMIX-Modell nutzt die Property `signature` für die Angabe des Typs eines Parameters die Property `fqn` des Typs statt `name`. Es kann außerdem vorkommen, dass eine Methode gar keine Property `name` besitzt. Dies kann zum

[1]http://buschmais.github.io/jqassistant/doc/1.4.0/

Attribut	jQA-Äquivalent
id	id
name	name
hash	-
fqn	fqn
parentScope	Startknoten der Relation CONTAINS

Tabelle 3.1: Mapping von FAMIX-Namespaces auf jQA-Knoten mit dem Label Package

Attribut	jQA-Äquivalent
id	id
name	name
hash	-
fqn	fqn
container	Startknoten der Relation CONTAINS
isInterface	:Interface statt :Class
modifiers	visibility, abstract, static, final
sourceAnchor	sourcefFileName

Tabelle 3.2: Mapping von FAMIX-Classes und FAMIX-ParameterizableClasses auf jQA-Knoten mit den Labels Class oder Interface

Attribut	jQA-Äquivalent
id, name, hash, fqn, container, modifiers, sourceAnchor	Analog wie für Class

Tabelle 3.3: Mapping von FAMIX-Enums auf jQA-Knoten mit dem Label Enum

Attribut	jQA-Äquivalent
id, name, hash, fqn, container, modifiers, sourceAnchor	Analog wie für Class

Tabelle 3.4: Mapping von FAMIX-AnnotationTypes auf jQA-Knoten mit dem Label Annotation

Beispiel bei einer Subklasse passieren, die die Methode von der Superklasse geerbt hat. In der Signatur ist der Name aber dennoch vorhanden. Außerdem ist der fehlende Name bei einer geerbten Methode unproblematisch, da diese Methoden in der Visualisierung nicht mit aufgenommen werden sollen. Auch wenn es für `numberOfStatements` kein Äquivalent

gibt, kann hierfür die Property `effectiveLineCount`, welches eine ähnliche Bedeutung besitzt, verwendet werden.

PrimitiveType, *ParameterType*, *ParamterizedType* und *Type* besitzen bei jQA kein eindeuti-

Attribut	jQA-Äquivalent
id	id
name	name
hash	-
fqn	Kombination aus fqn des Startknotens der Relation DECLARES und signature
cyclomaticComplexity	cyclomaticComplexity
declaredType	Endknoten der Relation RETURNS
kind	-
modifiers	visibility, static, final
numberOfStatements	-
parentType	Startknoten der Relation DECLARES
signature	signature
MethodType	-
sourceAnchor	sourceFileName des Startknotens der Relation DECLARES
accessesVariable	-

Tabelle 3.5: Mapping von FAMIX-Methods auf jQA-Knoten mit dem Label Method

ges Äquivalent. Besonders die Elemente *ParameterType* (der Platzhalter eines parametrisierten Typs) und *ParameterizedType* sind im jQA nicht aufgeführt. Ein typisches Beispiel für ein ParameterizedType im Java-Umfeld wäre `Map<String, String>`. jQA legt jedoch lediglich für die *ParameterizableClass* – in diesem Fall `Map` – einen Knoten mit dem Label `Type` an. Die Information, dass es sich um einen parametrisierten Typ handelt, geht veloren da jQAssisstant den Bytecode untersucht und beim Kompilieren von Java-Quelltext eine Typlöschung stattfindet (vgl. [Ullenboom 2018]). Das führt weiter zur Repräsentation eines PrimitiveTypes, dessen Äquivalent mit dem gleichen Label und Attributen angelegt werden müsste. Somit werden diese beiden Typen in jQA nicht unterschieden. Der jQA-Graph kann insofern erweitert werden, dass diese primitiven Typen gesucht werden und ein weiteres entsprechendes Label wie Primitive erhalten, da diese Information einmal während des City-Workflows gebraucht wird.

Attribut	jQA-Äquivalent
id	id
name	name
hash	-
fqn	Kombination aus fqn des Startknotens der Relation DECLARES und name
declaredType	Endknoten der Relation OF_TYPE
modifiers	visibility, static, final
parentType	Startknoten der Relation DECLARES
sourceAnchor	sourceFileName des Startknotens der Relation DECLARES
getterSetter	-

Tabelle 3.6: Mapping von FAMIX-Attributes auf jQA-Knoten mit dem Label Field

Attribut	jQA-Äquivalent
id	id
name	name
hash	-
fqn, sourceAnchor	analog zu Attribute
parentEnum	Startknoten der Relation DECLARES

Tabelle 3.7: Mapping von FAMIX-EnumValues auf jQA-Knoten mit dem Label Field

Attribut	jQA-Äquivalent
id	id
name	name
parentBehaviouralEntity	Startknoten der Relation DECLARES

Tabelle 3.8: Mapping von FAMIX-LocalVariables auf jQA-Knoten mit dem Label Variable

3.2 City

Bei City handelt es sich um eine Metapher, die das zu analysierende System als Stadt darstellt. Als Vorlage diente die von Wettel und Lanza (vgl. [Wettel/Lanza 2007]) implementierte “CodeCity“, von welcher der Grundaufbau bestehend aus Stadtteilen und Gebäuden in Getaviz übernommen wurde. In der Standardkonfiguration werden Pakete auf ebene Stadtviertel

Attribut	jQA-Äquivalent
id	id
name	-
fqn	-
declaredType	Endknoten der Relation OF_TYPE
parentBehaviouralEntity	Startknoten der Relation HAS

Tabelle 3.9: Mapping von FAMIX-Parameters auf jQA-Knoten mit dem Label Parameter

Attribut	jQA-Äquivalent
id	id
candidates	Endknoten
sender	Startknoten

Tabelle 3.10: Mapping von FAMIX-Invocations auf jQA-Relationen mit dem Namen INVOKES

Attribut	jQA-Äquivalent
id	id
accessor	Startknoten
variable	Endknoten
isWrite	Durch Relation selbst abgedeckt

Tabelle 3.11: Mapping von FAMIX-Accesses auf jQA-Relationen mit den Namen READS oder WRITES

Attribut	jQA-Äquivalent
id	id
subClass	Startknoten
SuperClass	Endknoten

Tabelle 3.12: Mapping von FAMIX-Inheritances auf jQA-Relationen mit den Namen EXTENDS oder IMPLEMENTS

abgebildet und Klassen als Gebäude dargestellt, die sich auf die Stadtviertel entsprechend der Pakete, die sie beinhalten, verteilen. Die Breite und Höhe der Gebäude berechnet sich aus der Anzahl der Methoden und Attribute innerhalb der Klassen, diese Bestandteile werden

also nicht explizit dargestellt. Neben der Standardkonfiguration sind weitere Varianten hinzugekommen, mit denen auch Attribute und Methoden explizit visualisiert werden. Während bei der Variante "City Floor" Methoden als Etagen angelegt und Attribute auf dem Dach als kleine Zylinder platziert werden, setzt sich ein Gebäude bei der Variante "City Brick" aus nebeneinander und übereinanderliegenden Würfel, die Methoden und Attribute repräsentieren, zusammen, während die Klasse nur die Grundfläche für das Gebäude bildet. Eine weitere Variante ist "City Panel", bei der ein Gebäude aus übereinanderliegenden Quadern für Methoden entsteht, auf denen sich wiederum flache Zylinder für Attribute stapeln.
Trotz der vielfältigen Darstellungsmöglichkeiten der Strukturelemente, werden diese innerhalb des Xtext-Modells zu City jedoch nicht unterschieden, sondern erst auf Basis einer Konfigurationsdatei bei der weiteren Verarbeitung zur eigentlichen Visualisierung festgelegt. Das Metamodell definiert statt Elementen wie *Floor*, *Panel* oder *Brick* lediglich die drei Entitäten *District*, *Building* und *Building Segment*, um entweder ein Stadtteil, ein Gebäude oder ein Gebäudesegment zu beschreiben. In der Regel werden *Namespaces* auf *Districts*, *Structures* auf *Buildings* und *Attributes* sowie *Methods* auf *Building Segments* abgebildet. Ein Ausnahme ist die Variante "City Dynamic". Hier werden *Structures* zu *Districts* und *Methods* zu *Buildings*. Ob ein Bestandteil wie eine Methode als Block innerhalb der Variante "City Bricks" definiert wurde, lässt sich somit nicht anhand des Typs der Entität, auf die sie abgebildet wurde, bestimmen. Im Graphen kann diese Eigenschaft aber beispielsweise durch ein Label festgehalten werden, wodurch es möglich wäre, mehrere Varianten einer Visualisierung zu City in einem Graph zu halten und zu unterscheiden.

3.3 Recursive Disk

Im Vergleich zu City als Echtweltmetapher, stellt Recursive Disk eine abstraktere Form der Darstellung dar. Hier wird die Visualisierung in Form geschachtelter Scheiben aufgebaut. Pakete werden zu äußeren Ringen und können entweder weitere Ringe für enthaltene Pakete oder Klassen beinhalten (vgl. [Müller/Zeckzer 2015]). Im Metamodell werden dafür die beiden Entitäten *Disk* und *Disk Segment* vereinbart. Letzteres kann immer nur innerhalb einer *Disk* existieren. Auch für Recursive Disk gibt es verschiedene Konfigurationen, durch die unterschiedliche Varianten der Visualisierung erzeugt werden. Pakete und Klassen werden jedoch immer auf Disks abgebildet. Abbildung 3.3 zeigt die verschiedenen Konfigurationseinstellungen und auf welche Entitäten die Strukturelemente jeweils abgebildet werden.

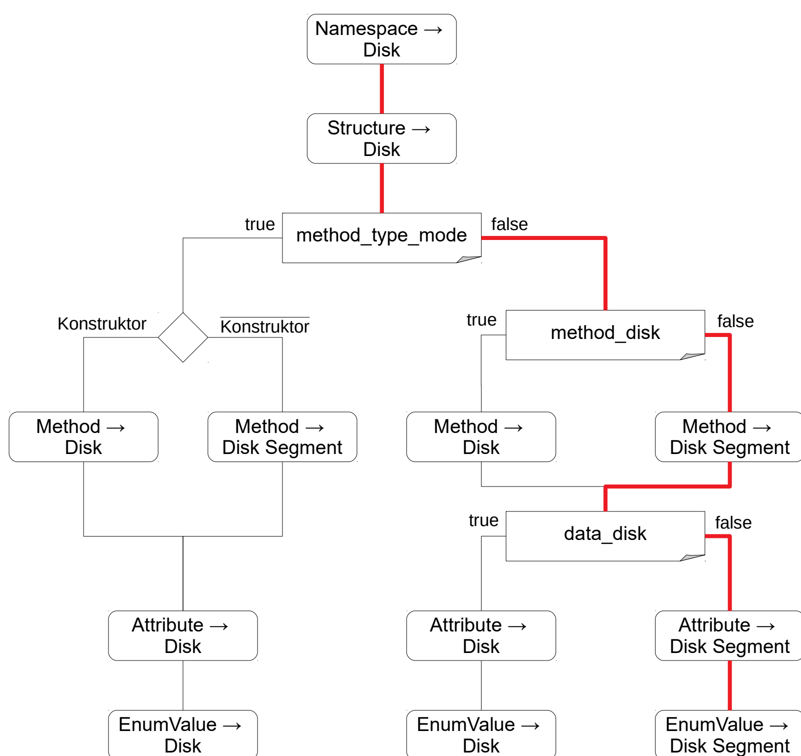

Abbildung 3.3: RD Konfigurationszusammenstellung. Der rot markierte Pfad stellt die Standardkonfiguration dar

4 Neo4j-Integration

4.1 Konzipierung des Datenmodells

Auf der Grundlage der getätigten Analyse erfolgen nun die Konsequenzen für das Datenmodell. Im Folgenden sollen drei Ansätze vorgestellt und diskutiert werden, wie die Infomationen aus dem FAMIX-Modell und dem jQA-Modell miteinander verbunden werden können.

1. **Alle FAMIX-Elemente hinzufügen:** In die Datenbank, die alle jQA-Enitäten enthält, werden für jedes FAMIX-Element entsprechende Knoten und Beziehungen samt aller Attribute eingefügt.
2. **Vollständig auf FAMIX verzichten:** Nur die Strukturinformationen, die der jQA-Graph liefert, werden für die Visualisierung genutzt.
3. **Fehlende FAMIX-Elemente und Attribute hinzufügen:** Es werden Knoten und Attribute hinzugefügt, die jQAssisstant nicht abbildet.

Zu 1.: Abschnitt 3.1.2 zeigt, dass die meisten Elemente und genutzten Attribute aus dem FAMIX-Modell ein entsprechendes Äquivalent in jQAssisstant besitzen. Dadurch würde das Hinzufügen aller FAMIX-Elemente zu einer hohen Redundanz von Strukturinformationen führen. Zudem stellt sich Frage, ob es dann überhaupt einen Mehrwert liefert, FAMIX und jQA-Graph zusammen in einer Datenbank abzulegen, da FAMIX mehr Strukturinformationen bereithält und es ausreicht, durch diesen Teilgraph zu navigieren, um die Visualisierung zu erstellen. Zu beachten ist jedoch, dass jQAssisstant lediglich auf die Analyse von in Java implementierten Systemen ausgelegt ist, während FAMIX sprachunabhängig genutzt werden kann. Somit ist gewährleistet, Strukturinformationen für alle Sprachen zur Verfügung zu stellen, für die es FAMIX-Extraktoren gibt.
Zu 2.: Dadurch, dass der Informationsgehalt des jQA-Graphen für die verwendeten FAMIX-Elemente und Attribute nur in vereinzelten Punkten geringer ausfällt, eröffnet sich auch die Möglichkeit, lediglich den jQA-Graphen zu verwenden. Das zuvor beschriebene Redundanzproblem tritt somit nicht auf. Außerdem entfallen weitere Zusatzschritte, um die Datenbank mit weiteren Informationen anzureichern. Momentan wird das `hash`-Attribut, das in jQAssisstant nicht erstellt wird, für die ID des Objekts im Visualisierungsmodell und in den Metadaten verwendet. Besonders hierfür muss ein Ersatz gefunden werden, mit dem die ID belegt wird oder ein Kompromiss getroffen werden, der das Attribut `hash` für jeden Node hinzufügt, auch wenn das nicht mehr dem Datenmodell von jQAssisstant entsprechen würde. Im Gegensatz zum ersten Ansatz können durch die alleinige Nutzung von jQAssisstant jedoch nur noch in Java implementierte Systeme analysiert werden. Es wurde weiterhin bei der Analyse gezeigt, dass vereinzelte Attribute nicht deckungsgleich zum FAMIX-Modell belegt werden. Ein Beispiel hierfür sind innere Klassen, die mit einem Dollarzeichen markiert werden. Für diese muss festgelegt werden, ob sie aus jQAssisstant übernommen werden oder innerhalb des Transformationsprozesses angepasst werden.

Zu 3.: Indem nur in jQAssisstant fehlende Elemente und Attribute aus dem FAMIX-Modell hinzugefügt werden, wird am wenigstens Redundanz bei größtmöglicher Menge an Informationen erreicht. Fehlende Attribute können direkt am zugehörigen Knoten aus jQAssisstant hinzugefügt werden. So wäre zum Beispiel das Problem des fehlenden `hash`-Attributs gelöst, ohne eine Alternative für die ID festlegen zu müssen. Auch bei dieser Variante können nur in Java implementierte Systeme analysiert werden. Es bedarf zwar weiterer Verarbeitungsschritte, um den jQA-Graph zu erweitern (zum Beispiel `hash`-Properties und ein Labeling für primtive Typen), für diese benötigt es aber nicht wie bei Variante 1 das Einlesen einer zusätzlichen Datei, da diese Informationen aus dem Graphen selbst generiert werden können.
Für die Konzipierung des Datenmodells wird in Bezug auf die Strukturinformationen der dritte Ansatz gewählt, um den gleichen Informationsgehalt zu behalten. Wie jedoch schon unter 1 angedeutet, würde durch eine volle Redundanz die Nutzung des jQA-Graphen für die Visualisierung obsolet werden. Beim dritten Ansatz kann so jedoch für die Extraktion der Strukturinformationen der jQA-Scanner verwendet werden. Danach kann dieser Graph mit den Properties für `hash` und `fqn` angereichert werden. Alle primitiven Typen im Graph werden gesucht und das Label Primitive hinzugefügt. Zuletzt werden die Methoden untersucht, um für Getter- und Setter-Methoden das Label `Getter` beziehungsweise `Setter` hinzuzufügen. Eine Schwierigkeit, die dieser Ansatz noch bietet, ist, dass jQAssisstant mehr Elemente anlegt, als in der Visualisierung verarbeitet werden sollen. Dieses Problem tritt insbesondere bei inneren (anonymen) Klassen auf. Zur Vereinfachung des Problems werden jegliche innere Klassen im Zuge dieser Arbeit nocht nicht betrachtet. Hier bleibt noch Raum zur genaueren Untersuchung, welche Elemente beim Auslesen des jQA-Graphen ignoriert werden müssen und wie ein passendes Muster zum Filtern gewählt werden kann, sodass nicht nur einzelne Spezialfälle abgedeckt werden.
In den Abschnitten 3.2 und 3.3 wurde gezeigt, wie die Strukturelemente auf die Elemente der jeweiligen Visualisierung abgebildet werden. Dies lässt sich analog auf den Graphen anwenden, indem für jedes Element einer Visualisierung ein Knoten angelegt wird, der über eine ausgehende Beziehung namens VISUALIZES mit dem Knoten verbunden wird, den er visualisiert. Über ein entsprechendes Label kann ein Knoten entweder zugehörig zur RD- oder City-Visualisierung markiert werden. Abbildung 4.1 gibt einen Überblick, wie die Visualisierungselemente mit den Strukturelementen verbunden sind. Dabei wurden Beziehungen der Visualisierungselemente untereinander noch nicht betrachtet, ebenso wie mögliche Properties. Deswegen folgt in Abbildung 4.2 und 4.3 ein detaillierterer Blick auf die Graphen der jeweiligen Modelle.
Jedes Visualisierungsmodell erhält einen Wurzelknoten mit dem Label `Model`. Dieser wird genutzt, um weitere Informationen über das Modell zu hinterlegen, wodurch mehrere unterscheidbare Modelle in einem Graph gespeichert werden können. Dies erweist sich zum Beispiel als nützlich, um für verschiedene Versionen einer Software oder Varianten einer Metapher jeweils ein Modell zu speichern. In den Abschnitten 3.2 und 3.3 wurde bereits erwähnt, dass eine Metapher verschiedene Varianten besitzt, die mit Hilfe einer Konfigurationsdatei aus-

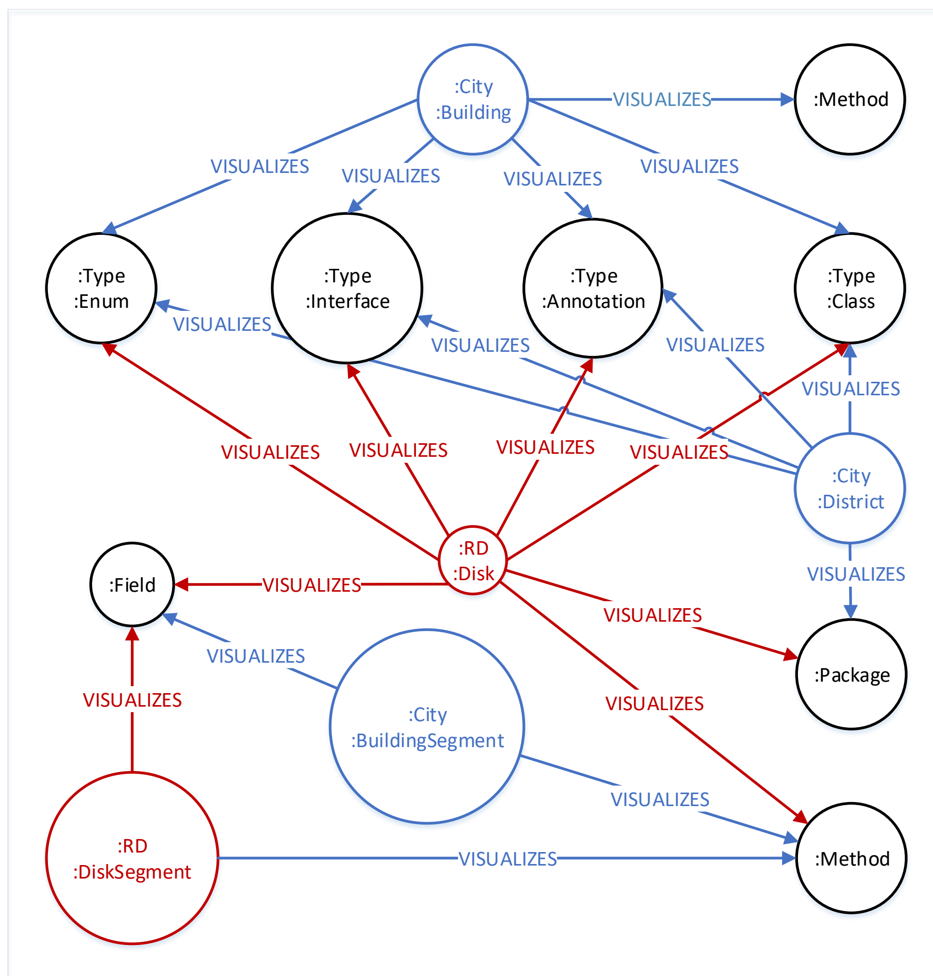

Abbildung 4.1: Überblick über die Verbindung der Visualisierungsknoten mit den Elementen aus dem jQA-Graph

gewählt werden. Da für die City-Metapher die Konfigurationseinstellung `building_type` maßgeblich die Variante bestimmt, wird dem Model-Knoten der City-Modelle die Property `building_type` hinzugefügt mit dem jeweiligen Wert aus der Konfigurationsdatei. Die erzeugte Variante bei Recursive Disk ist von der Wahrheitsbelegung mehrerer Konfigurationseinstellungen abhängig, weswegen hier zum Model-Knoten die drei Properties `method_type`, `data_type` und `data_disk` hinzugefügt werden. Ist `method_type` mit true belegt, kann die Angabe der anderen beiden Properties entfallen, da diese dann für die Abbildung der Strukturelemente auf die Visualisierungselemente nicht mehr relevant sind (s. Abbildung 3.3). Ein weiterer Vorteil des Wurzelknotens besteht darin, dass er über die Relation `CONTAINS` als Container für die Visualisierungselemente auf der obersten Ebene dient. Weitere Elemente sind über die Relation jeweils nur mit den direkten Elternelementen verbunden. Dies gewährleistet eine einfache Navigierbarkeit durch den Graph ausgehend vom Wurzelelement und eine einfache Berechnung der Leveltiefe anhand der Länge eines Pfades. Dadurch kann eine Property, die die Tiefe explizit für jedes Element speichern müsste, entfallen. Über ein Cypher-Statement kann auch der längste Pfad von einem Blatt zur Wurzel gefunden werden, sodass nicht mehr wie zuvor die Tiefe jeder Elemente miteinander verglichen werden muss, um das maximale Level zu bestimmen. Weitere Properties, die entfallen können, sind die für die Attribute `name`, `fqn`, `hash` und `type`, da diese ursprünglich aus dem FAMIX-Modell übernommen werden und nun über die Beziehung `VISUALIZES` das entsprechende Element abgefragt werden kann, das diese Properties besitzt. Desweiteren werden nun Attribute, die Beziehungen repräsentieren, wie `data`, `methods` oder `parent` auch über die Beziehung `CONTAINS` abgedeckt. Zu `data` gehören nun alle Endknoten dieser Relation, die ein Attribut visualisieren, also eine Relation `VISUALIZES` zu einem Attribut besitzen. Analog gilt das auch für Methoden. `Parent` wiederum sind immer die Startknoten der `CONTAINS`-Relationship. Da sich hinter den Attributen `position` und `separator` ein komplexes Objekt verbirgt, das aus weiteren Attributen besteht, wird hierfür jeweils ein einzelner Knoten angelegt und mit dem entsprechenden Element über eine Relation verbunden.

4.2 Auswahl der Migrationsstrategie

Für die zu erfolgende Integration von Neo4j innerhalb der beiden Workflows sind zwei grundlegende Vorgehensweisen denkbar.

1. Der erste Teilschritt jeder Komponente wird dahingehend verändert, dass alle Elemente, die in der Datenbank gespeichert sind, zunächst in Xtext-Objekte überführt werden, sodass wie bisher für den Workflow Listen mit allen Entitäten eines Xtext-Modells (zum Beispiel *Disk*, *District*, *FAMIXNamespace* usw.) vorliegen. Dadurch kann der zweite Teilschritt eines Transformationsschritts ohne weitere Anpassung des Ablaufs oder der bisher implementierten Funktionen erfolgen. Im dritten Teilschritt werden dann alle neuen Objekte in das Datenmodell für die Datenbank überführt beziehungsweise bereits in der Datenbank bestehende Elemente gesucht und mit den neu gesetzten Werten

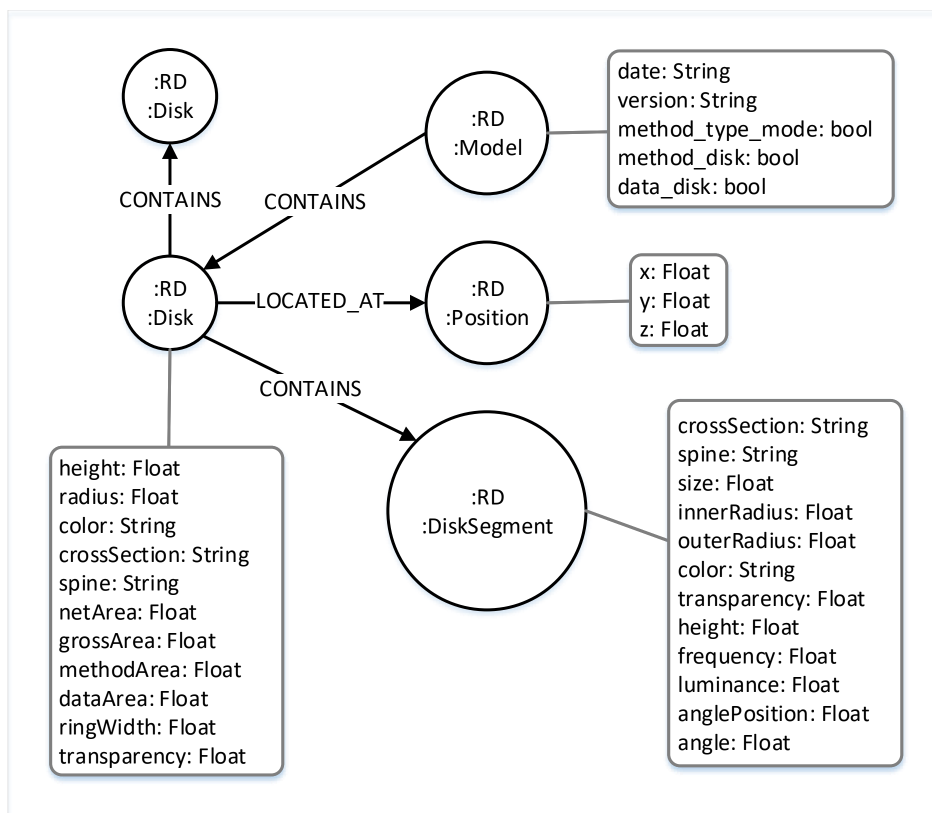

Abbildung 4.2: RD-Teilgraph

aktualisiert. Der Zugriff auf die Datenbank erfolgt also nur am Anfang und am Ende, während die eigentliche Verarbeitung unberührt bleibt.

2. Statt für die Datenbankelemente zunächst ein Mapping auf die Xtext-Objekte vorzunehmen, erfolgt der Zugriff auf die Datenbank direkt innerhalb des Verarbeitungsschrittes. Dafür muss der Ablauf der Verarbeitung verändert werden. Daraus folgt vor allem eine Neuimplementierung aller bisherigen Funktionen, angepasst auf die Zugriffswege, die durch Neo4j vorgegeben sind. Damit jede Funktion weiterhin für die Xtext-Objekte verwendet werden kann, wird ein Duplikat von ihr angelegt, welches dann angepasst wird. Das führt dazu, dass die klare Aufteilung in die drei Teilschritte aufgelöst wird, da die Elemente immer nur ausgelesen werden, wenn sie gerade innerhalb der Verarbeitung gebraucht werden und auch das Speichern des Modells nicht erst im dritten Teilschritt, sondern während des Verarbeitungsprozesses erfolgt.

Der Vorteil der ersten Strategie besteht in der klaren Aufteilung, an welchen Stellen der Datenbankzugriff erfolgt. Dadurch können durch die Integration entstandene Fehler besser lokalisiert werden. Wenn die Visualisierung nicht mehr korrekt aufgebaut wird, kann das nur an einem fehlerhaften Mapping oder Speichern der Elemente liegen, nicht aber an der Verarbeitung, da die Funktionen bereits korrekt arbeiten. Es werden zwar Objekte der Xtext-Modelle verwendet, es müssen aber keine Modelle mehr während des gesamten Transformationsprozesses im Speicher gehalten werden. Dadurch sind auch keine Workflows mehr notwendig, da die Komponenten auch mit Hilfe einer Startklasse nacheinander ausgeführt

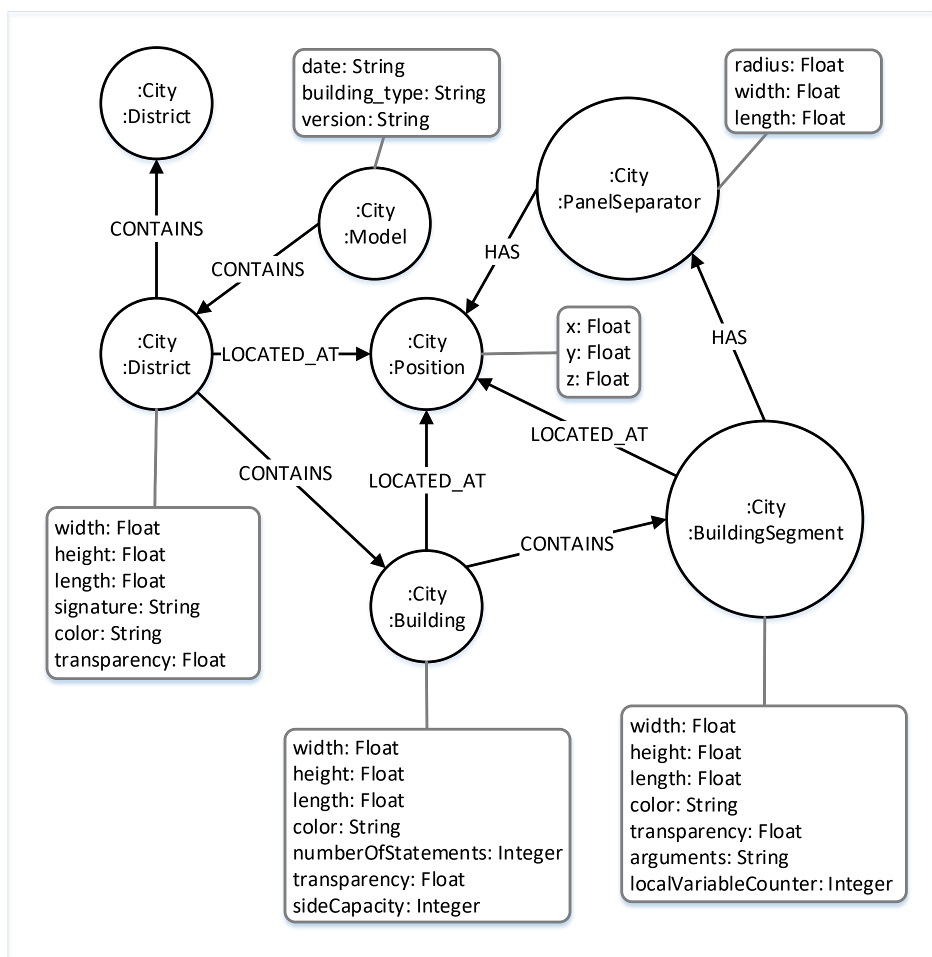

Abbildung 4.3: City-Teilgraph

werden können.
Dennoch gibt es stärkere Argumente für die Wahl der zweiten Strategie, weswegen diese auch umgesetzt wird. Die Integration des Datenbankzugriffs innerhalb der Verarbeitungsschritte ist zwar aufwendiger und komplexer, da jede Funktion neu implementiert und auch getestet werden muss. Der wesentliche Vorteil liegt aber darin, dass so bereits Xtext vollständig abgelöst werden kann. Die Umsetzung der ersten Strategie würde später zwangsläufig auch zur zweiten Strategie führen, wenn auch Xtext abgelöst werden soll. Möchte man Neo4j ausschließlich für die Generierung der Modelle einsetzen, gilt es frühzeitig zu klären, wie sich die veränderten Zugriffswege auf die Verarbeitung auswirken. Über die zweite Strategie kann bereits aufgedeckt werden, an welchen Stellen die Verarbeitung erleichtert oder sogar erschwert werden könnte. Besonders Letzteres sollte frühzeitig erkannt werden, um geeignete Maßnahmen zu entwickeln, das auszugleichen.

4.3 Implementierung

4.3.1 Allgemeine Beschreibung

Dadurch, dass die gewählte Migrationsstrategie vollständig auf die Xtext-Modelle verzichtet, müssen auch die Workflows Famix2City und Famix2RD nicht mehr verwendet werden, um eine Transformation auszuführen. Stattdessen werden neue Komponenten erstellt sowie eine Startklasse für jede Metapher, die die Komponenten für die einzelnen Transformationsschritte aufruft. Die neuen Komponenten, die den Datenbankzugriff implementieren, orientieren sich in Bezug auf Ausführung und Aufteilung sehr stark an den beiden Workflows. An Stelle der Komponente `Famix2Famix` tritt nun die Komponente `JQAEnhancement`, die den jQA-Graphen mit weiteren Properties und Labels anreichert. Die Berechnung des Wertes für die neu anzulegende Property `hash` wird von der Funktion `createID` aus `Famix2Famix` übernommen, ohne dass weitere Anpassungen erfolgen müssen. Für Getter- und Setter-Methoden werden – wie schon für Konstruktoren existent – Labels hinzugefügt, um diese entsprechend zu kennzeichnen. Für die Bestimmung des Methodentyps wird nicht wie in `Famix2Famix` eine programmatische Lösung implementiert. So werden für das Aufspüren der Getter-Methoden über ein Cypher-Statement (Listing 4.1) alle Relationen mit dem Namen `READS` ausgewählt, deren Methodenname mit der Zeichenfolge "get"beginnt gefolgt vom Namen des Attributs unter der Bedingung, dass sich beide innerhalb des gleichen Typs befinden.

```
1  MATCH (o:Type)-[:DECLARES]->(method:Method)-[getter:READS]->
2        (attribute:Field)<-[:DECLARES]-(q:Type)
3  WHERE method.name =~ "get[A-Z]+[A-Za-z]*"
4  AND toLower(method.name) contains(attribute.name) AND o.hash = q.hash
5  RETURN method
```

Listing 4.1: Cypher-Statement zur Rückgabe von Getter-Methoden

Für Setter-Methoden kann analog ein Statement über die Relation `WRITES` angewendet werden. Für des Setzen des Labels `Primitive` wird dieses einfache Cypher-Statement in Listing 4.2, da die primitiven Typen die einzigen sind, deren `fqn` lediglich aus Kleinbuchstaben bestehen.

```
1 MATCH (p:Type)
2 WHERE p.name =~ "[a-z]+"
3 RETURN p
```

Listing 4.2: Cypher-Statement zur Rückgabe von primitiven Typen

Um innere anonyme Klassen einfach von der Visualisierung auszuschließen, werden sie mit einem Label versehen, das sie als solche kennzeichnet, wofür die beiden Konzepte java:InnerType[1] und java:AnonymousInnerType[2] von jQAssisstant angewendet werden.
Die Komponente `Famix2JSON` wird ersetzt durch `JQA2JSON`. Da für den fqn, der dem Key `qualifiedName` zugewiesen wird, für Methoden und Attribute keine Property existiert, muss dieser aus der Signatur und dem `fqn` der Container-Klasse zusammengesetzt werden. Für Attribute und Werte von Enums ist es ausreichend, den Namen der Containerklasse mit dem Namen des Elements zu verbinden. Für `qualifiedName` und `name` einer inneren Klasse müssen noch Anpassungen vorgenommen werden, bevor sie verwendet werden. Für `fqn` wird das Dollarzeichen einfach durch einen Punkt ersetzt. Vom Namen wird alles entfernt, was vor dem Dollarzeichen steht inklusive dem Zeichen selbst.
Die Komponenten `JQA2City` und `JQA2RD` sind die Pendants zu `Famix2City` und `Famix2RD`. Von ihnen wurden Ablauf und Funktionen übernommen und angepasst, um die Graph-Objekte zu verarbeiten. Zu Beginn werden für die Modelle die Wurzelknoten angelegt und mit den Metadaten für Datum und Variante bestückt. Danach wird der jQA-Graph ausgehend von den Wurzelpaketen durchlaufen, für jeden zu visualisierenden Knoten ein Visualisierungsknoten der gewählten Metapher angelegt und über eine Relationship mit ihm verbunden. Die Visualisierungselemente, die eines der Wurzelpakete visualisieren, werden außerdem mit dem Model-Knoten verbunden. Der Graph wird so weiter durchlaufen, dass jeweils immer die Kindknoten des zuletzt abgebildeten jQA-Elements gewählt werden, bevor der nächste Knoten auf der gleichen Ebene gewählt wird. Wichtig ist hier, dass nicht jedes Element visualisiert werden soll. Dabei ist das Label für innere anonyme Typen hilfreich, um diese beim Durchlauf zu überspringen. Bei der Auswahl der zu visualisierenden Typen müssen für City nur die Typen mit dem Label `Class` und `Interface` gewählt werden, da `Enum` und `Annotation` dort nicht dargestellt werden. Außerdem müssen alle Methoden übersprungen werden, deren Name ein Dollarzeichen enthält, da diese sich nicht innerhalb des Typs befinden, obwohl sie über die `DECLARES`-Relation als solche gekennzeichnet werden. Für die Elemente *BuildingSegments* werden für die Variante "City Floors" zusätzlich entweder das Label `Floor` (für Methoden) oder `Chimney` (für Attribute) angehängt. Das erleichtert in folgenden Schritten das Filtern von *BuildingSegment*-Knoten ohne zusätzliche

[1] http://buschmais.github.io/jqassistant/doc/1.4.0/#java:InnerType
[2] http://buschmais.github.io/jqassistant/doc/1.4.0/#java:AnonymousInnerType

Prüfung, ob eine Beziehung zu einem Attribut oder einer Methode besteht, da stattdessen nur eines der beiden Label angegeben werden muss.
Um die Visualisierungsmodelle zu modifizieren, werden die Komponenten RDModification und CityModification in Anlehnung an die Komponenten City2City und RD2RD angelegt. Für die dort verwendeten Hilfs- und Layout-Klassen CityUtils, CityLayout, BrickLayout, Rectangle und CircleWithInnerCircles werden keine neuen Klassen implementiert, sondern Funktionen hinzugefügt. Zur Erzeugung der Ausgabe wird das Format X3D gewählt, da dies in der Standardkonfiguration verwendet wird. In Anlehnung an die Komponenten City2X3D und RD2X3D werden deshalb die Komponenten CityOutputX3D und RDOutputX3D angelegt und über eine neue Funktion in den schon vorhandenen Hilfsklassen CityOutput und RDOutput aufgerufen.

4.3.2 Auswirkungen der erfolgten Migration auf den Transformationsprozess

Aufgrund der gewählten Migrationsstrategie ergeben sich Veränderungen in der Verarbeitungslogik, aus denen Vorteile, aber auch Probleme resultieren. Anhand von Beispielen wird auf diese nun im Detail eingegangen.
Eine wesentliche Änderung besteht darin, dass vor Beginn der Verarbeitung nicht alle Elemente, die verarbeitet werden sollen, in Collections gehalten werden, um über sie zu iterieren. Wird eine Anfrage an die Datenbank gestellt, die mehrere Objekte zurückgibt, können die Interfaces Result und ResourceIterator<T> der Java-API verwendet werden, um über die Objekte zu iterieren. Die Besonderheit liegt hier darin, dass immer nur das Objekt in den Speicher geladen wird, auf das gerade zugegriffen wird. Danach wird es automatisch wieder freigegeben (vgl. [The Neo4j Graph Database Project 2018]). Listing 4.3 zeigt, wie der ResourceIterator in CityModification genutzt wird, um Properties für alle Building Nodes zu setzen. Die Funktion execute gibt zunächst ein Result-Objekt zurück, das über die map-Methode in RecourceIterator<Node> umgewandelt wird. Es wäre auch möglich, direkt über das Result-Objekt zu iterieren. Diese Variante wurde lediglich gewählt, um die Rückgabe der Objekte (Zeile 1-3) syntaktisch von der Verarbeitung (Zeile 4) zu trennen.

```
1 val buildingNodes = graph.execute(
2     "MATCH (m:Model:City)-[:CONTAINS*]->(b:Building) RETURN b"
3 ).map[return get("b") as Node]
4 buildingNodes.forEach[setBuildingProperties]
```

Listing 4.3: Verwendung des ResourceIterator

Eine weitere gravierende Änderung betrifft die Art und Weise des Zugriffs auf Elemente, die in Relation zu einem anderen Element stehen. In City2City wird das Statement methods.filter[parentType.ref === elem] genutzt, um die Methoden einer Klasse zurückzugeben. elem ist die gegebene Klasse, methods eine Liste mit allen Methoden aus dem Modell und parentType ein Referenzobjekt, dass die Klasse enthält, zu der die Methode gehört. Um also die Methoden zu finden, die zu der gegebenen Klasse gehören, muss

diese Liste komplett durchlaufen werden, um alle Methoden zurückzugeben, deren Referenzobjekte mit der Klasse übereinstimmen. Die in `CityModifikation` implementierte Variante nutzt den Funktionsaufruf `getEndNodes(Rels.DECLARES, Labels.Method, structure)`. Die Implementierung der Funktion ist in Listing 4.4 aufgeführt. Mit Hilfe der Funktion `traversalDescription` wird ein `TraversalDescription`-Objekt zurückgegeben, das genutzt wird, um ausgehend vom übergebenen Startknoten durch den Graph zu traversieren. Die Funktionen, die `TraversalDescription` bereitstellt, werden genutzt, um Einschränkungen zu treffen, welche Pfade und Knoten berücksichtigt werden sollen. Über die drei verwendeten Evaluatoren werden nur die nächsten Nachbarknoten gewählt, die das angegebene Label besitzen. Der Evaluator `fromDepth(1)` wird benötigt, damit der Startknoten nicht mit in die Auswahl genommen wird. Auch wenn der Aufruf über das `TraverserDescription`-Objekt aufwendiger aussieht als das Statement in `City2City`, werden dafür bei der Suche nur die Elemente untersucht, die zur Klasse gehören, statt alle Methoden im Graph durchzugehen. Außerdem werden die Knoten durch den Aufruf von `nodes` am Ende auch über einen `ResourceIterator` zurückgeben, wodurch nicht alle Elemente auf einmal in den Speicher geladen werden.

```
1  def getEndNodes(Rels relationship, Labels label, Node startNode) {
2      val endNodes = graph.traversalDescription().relationships(
           relationship, Direction.OUTGOING)
3      .evaluator(Evaluators.toDepth(1)).evaluator(Evaluators.fromDepth(1))
4      .evaluator(new EndNodeEvaluator(label))
5      .traverse(startNode).nodes
6      return endNodes
7  }
```

Listing 4.4: Funktion zur Rückgabe des Endknoten eines bestimmten Labels zu einer gegebenen Relationship

Trotz der Vorteile gilt es zu beachten, dass aufgrund des direkten Datenbankzugriffs Schwierigkeiten entstehen. Jeder Zugriff auf die Datenbank – sei er lesend oder schreibend – muss innerhalb eines Transaktionsblocks erfolgen. Eine Property wird über die Funktion `setProperty` eines `Node`- oder `Relationship`-Objekts gesetzt oder aktualisiert, sofern die Property schon existiert. Persistent ist dieser Wert erst, wenn die Transaktion, in der er gesetzt wurde, beendet wird. Das bedeutet, dass der Wert auch erst dann wieder über die Funktion `getProperty` ausgelesen werden kann. Das führt während der RD-Modifikation zu dem Problem, dass erst die Property `netArea` für jedes Element gesetzt wird, aber in der gleichen Transaktion dieser Wert noch nicht gespeichert ist und für die Berechnung der Property `radius` benötigt wird. An dieser Stelle kann das Problem noch recht einfach gelöst werden, indem nicht in einer einzigen Transaktionen alle Modifikationen durchgeführt werden, sondern die Transaktionen in mehrere aufeinanderfolgende Transaktionen aufgeteilt wird, sodass die Werte in der Datenbank gespeichert sind und über `getProperty` zurückgegeben werden können. Auch während der City-Modifikation tritt das Problem auf eine etwas andere Weise auf. Die Methode `cityLayout` der Klasse `CityLayout` erzeugt rekursiv

Rechtecke für die Visualisierungselemente und verwendet dafür die Properties `width` und `length`, die aber innerhalb der tieferen rekursiven Funktionsaufrufe verändert werden. Hier kann als Lösung keine Aufteilung in mehrere Transaktionen genutzt werden, um den aktuellen Wert der Properties zu speichern. Würde man versuchen, innerhalb der rekursiv aufgerufenen Funktionen Transaktionen anzulegen, würde dies zu einer Verschachtelung von Transaktionen führen (für jeden Aufruf eine weitere Schachtelung), was von Neo4j nicht unterstützt wird. Eine Begründung liegt darin, dass es bedeuten würde, eine eigentlich erfolgreich beendete innere Transaktion rückgängig machen zu müssen, wenn die äußere Transaktion fehlschlägt. Um das Problem zu lösen, werden die Werte für `width` und `length` für jeden Visualisierungselement in einer Datenstruktur festgehalten. Hierfür wird eine `HashMap` verwendet, mit der ID eines Knotens für den Key und als Value ein Array mit den Werten der Properties. An Position 0 steht der Wert für `width`, an der ersten Position der Wert für `length`. Bevor `cityLayout` ausgeführt wird, werden die Werte der Elemente in die `Map` übertragen und diese als Parameter der Funktion mit übergeben. Wann immer einer der Werte verwendet oder neu gesetzt wird, geschieht das über die Map. Erst, wenn keine Änderung mehr erfolgt, werden die Properties mit den Werten aus der Map aktualisiert. Hier ist zu betonen, dass diese vorgestellte Lösung lediglich eine Übergangslösung darstellen soll und es einer eleganteren Lösung bedarf.

5 Zusammenfassung und Ausblick

Innerhalb dieser Arbeit wurde ein Datenmodell für die Modelle der Metaphern City und Recursive Disk sowie für die Struktur einer Software konzipiert, um Neo4j zum Speichern der Modelle in den Generator zu integrieren. Dafür wurden zunächst die bestehenden Xtext-Modelle analysiert, um benötigte Entitäten und deren Eigenschaften festlegen zu können. Die Analyse bildete die Grundlage, um darauffolgend das Datenmodell zu entwerfen. Aufgrund des Vergleichs des FAMIX-Modells mit dem Datenmodell von jQAssistant ergab sich eine Ablösung von FAMIX durch den von jQAssisstant erzeugten Strukturgraphen, der dafür mit weiteren Information angereichert werden musste. Dadurch kann der Scanner von jQAssisstant genutzt werden, um Strukturgraphen von System zu erstellen, wodurch auf das Einlesen einer FAMIX-Datei zur Erstellung des Modells verzichtet werden kann. Als Nachteil wird jedoch hingenommen, zunächst nur Java-basierte Systeme analysieren zu können. jQAssisstant arbeitet jedoch auf Bytecode, sodass prinzipiell alle Sprachen unterstützt werden, die sich in Java-Bytecode überführen lassen. Nach dem Entwurf erfolgte die Implementierung des entworfenen Modells in Form neuer Komponenten, die sich sehr nah an den in den Workflows Famix2City und Famix2RD verwendeten Komponenten orientieren. Durch den Zugriff auf die Datenbank innerhalb der Verarbeitungsschritte kommen die beiden Transformationen bereits vollständig ohne Xtext und Workflows aus, indem die Komponenten nun über eine einfache Startklasse nacheinander ausgeführt werden. Aufgrund der gewählten Migrationsstrategie wurden außerdem weitere Vor- und Nachteile aufgedeckt und anhand von Beispielen erläutert. Eine Evaluation der auf diesem Weg erzeugten Visualisierungen konnte aus Zeitgründen nur in Grundzügen getätigt werden.

Hier besteht weiterer Bedarf, eine ausführlichere Evaluation vorzunehmen. Das bedeutet vor allem, dass alle bisherigen Tests neu erstellt werden müssen. Dafür braucht es jQA-Graphen der Testsysteme für den Input, da die neuen Komponenten keine FAMIX-Dateien mehr verwenden. Weiterhin müssen die Soll-Dateien für die erzeugten Visualisierungsmodelle und Metadaten angepasst werden, da durch die veränderte Paketstruktur mehr Elemente hinzukommen und die enthaltenen Elemente anders angeordnet sind. Dass die Nutzung der Datenbank den Arbeitsspeicherverbrauch verbessert, wurde bisher nur an anhand eines Beispielsystems überprüft. Hier müsste die Performance anhand weiterer Systeme getestet werden. Besonders interessant wären in dem Zuge Systeme, die aufgrund ihrer Größe bisher nicht verarbeitet werden konnten.

Da die implementierte Lösung nur für das Strukturmodell und die Metaphern City und Recursive Disk umgesetzt wurde, müssen zum Einen noch die fehlenden Metaphern Plant und Multisphere und zum Anderen die Modelle der Verhaltens- und Entwicklungsaspekte hinzugefügt werden. Für die beiden Modelle HISMO und DYNAMIX müssen neue Extraktoren entwickelt werden, die einen Neo4j-Graphen erzeugen, statt Textdokumente zur Verfügung zu stellen. Denkbar wäre statt eines neuen Extraktors als Zwischenlösung ähnlich wie in der

Komponente Famix2Famix ein erzeugtes Xtext-Modell zu nutzen, um daraus den Graph zu erstellen, sodass Xtext nur zu Beginn des Transformationsprozess verwendet wird.

Literaturverzeichnis

[Baum et al. 2017] David Baum, Jan Schilbach, Pascal Kovacs, Ulrich Eisenecker und Richard Muller. „GETAVIZ: Generating Structural, Behavioral, and Evolutionary Views of Software Systems for Empirical Evaluation“. In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2017, S. 114–118. DOI: 10.1109/VISSOFT.2017.12.

[Ducasse et al. 2011] Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval und Tudor Girba. *MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family: Research Report*. 2011. URL: https://hal.inria.fr/hal-00646884.

[Müller 2015] Richard Müller. „Software Visualization in 3D - Implementation, Evaluation, and Applicability“. Dissertation. Leipzig: Universität Leipzig, 2015.

[Müller/Zeckzer 2015] Richard Müller/Dirk Zeckzer. „The Recursive Disk Metaphor - A Glyph-based Approach for Software Visualization“. In: *Proceedings of the 6th International Conference on Information Visualization Theory and Applications*. SCITEPRESS - Science and and Technology Publications, 2015, S. 171–176. DOI: 10.5220/0005342701710176.

[Neo4j Graph Database Project 2018] Neo4j Graph Database Project. *Chapter 3. Cypher - The Neo4j Developer Manual v3.4*. 2018. URL: https://neo4j.com/docs/developer-manual/current/cypher/ (besucht am 26. 09. 2018).

[The Neo4j Graph Database Project 2018] The Neo4j Graph Database Project. *Neo4j 3.4.7 API*. 2018. URL: https://neo4j.com/docs/java-reference/current/javadocs/ (besucht am 26. 09. 2018).

[Ullenboom 2018] Christian Ullenboom. *Java ist auch eine Insel: Einführung, Ausbildung, Praxis*. 13., aktualisierte und überarbeitete Auflage. Bonn: Rheinwerk, 2018. ISBN: 9783836258692.

[Vukotic 2015] Aleksa Vukotic. *Neo4j in action*. Shelter Island, NY: Manning Publications, 2015. ISBN: 978-1-617290-76-3. URL: http://proquest.tech.safaribooksonline.de/9781617290763.

[Wettel/Lanza 2007] Richard Wettel/Michele Lanza. „Program Comprehension through Software Habitability“. In: *Proceedings of ICPC 2007 2017*. 2007, S. 231–240. URL: http://wettel.github.io/download/Wettel07a-icpc.pdf.

[2018]. *Xtext Documentation*. URL: https://www.eclipse.org/Xtext/documentation/102_domainmodelwalkthrough.html (besucht am 20. 09. 2018).

[2018]. *Xtext Documentation*. 2018. URL: https://www.eclipse.org/Xtext/documentation/302_configuration.html (besucht am 20. 09. 2018).

Ehrenwörtliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann. Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.

Ort: Datum: Unterschrift: