Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

Felix Loos

Masterarbeit

A VISUAL UML-EDITOR FOR QDACITY

Eingereicht am 09.11.2017

Betreuer: Andreas Kaufmann, M. Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Happurg, 09.11.2017 _____

Felix Loos

License

Happurg, 09.11.2017 _____

Felix Loos

Abstract

QDAcity ist eine Webanwendung zur Unterstützung der qualitativen Datenanalyse (QDA) von Text-Daten. Bei der qualitativen Datenanalyse geht es darum Informationen und Zusammenhänge aus unstrukturierten Daten wie Interviewtranskriptionen zu gewinnen. Ein wichtiger Prozess bei der QDA ist die Kodierung, bei der Textstellen mit selbstdefinierten Codes versehen werden, um die Inhalte besser zu strukturieren. Die Anwendung QDAcity unterstützt die Kodierung von Texten mit einem Codesystem, welches die Codes hierarchisch strukturiert.

In dieser Arbeit wurde für QDAcity eine Software entwickelt, durch die es erstmals möglich ist, einzelne Elemente des konzeptuellen Modells (Klassen oder Beziehungen) mit Elementen aus qualitativen Daten (Codes) zu verknüpfen. Es wurde also ein Domänenmodell-Editor entwickelt, der auf Basis eines Codesystems arbeitet, und bei dem sowohl für das Klassendiagramm, als auch für das Codesystem ein gemeinsames Modell verwendet wird. Für die Umsetzung wurde das Codesystem um eine Codesystem-Language erweitert, die die Kategorisierung und damit die Abbildung in einen UML-Klassen-Diagramm ermöglicht.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Bei der qualitativen Forschung wird versucht die Frage zu beantworten, was, warum und wie Menschen, denken und handeln. Die qualitative Datenanalyse ist elementarer Bestandteil qualitativer Forschung und findet die weiteste Verbreitung in Forschungsfeldern wie z.B. Soziologie, Psychologie, Marktforschung oder Medizin. Qualitative Daten sind Informationen, die nicht auf eine numerische Darstellung reduziert werden können. Das können z.B. Transkriptionen aus Interviews oder Daten aus Fokus-Gruppen sein. Für die Strukturierung der Informationen gibt es verschiedene Prozesse, wie die Kodierung. Dabei werden Texte, bzw. Stellen im Text mit sogenannten Codes markiert. Diese Codes werden vom Benutzer je nach Anwendung selbst definiert.

Die Webanwendung QDAcity wurde entwickelt, um die qualitative Datenanalyse Softwaregestützt zu betreiben. In QDAcity können zu Projekten Textdokumente hochgeladen oder erstellt werden. Mithilfe von Codesystemen werden die Codes des Projekts hierarchisch verwaltet.

Neben QDAcity gibt es viele andere Projekte, welche die qualitative Datenanalyse softwareseitig unterstützen. Solche Tools sind zum Beispiel MaxQDA (Verbi GmbH, 2017), QDA-Miner (Provalis, 2017) oder RQDA (RQDA, 2016). Diese Anwendungen arbeiten ähnlich wie QDAcity mit Codes und Codesystemen. Allerdings bietet keines dieser Projekte die Möglichkeit, das Codesystem in andere Modelle (z.B. UML-konform) abzubilden.

In dieser Arbeit wurde für QDAcity ein Domänenmodell-Editor entwickelt, durch den es erstmals möglich ist das Codesystem auf ein UML-Klassendiagramm abzubilden. Dafür wird eine Codesystem-Language verwendet, durch die die Codes in Kategorien eingeteilt werden. Der UML-Editor nutzt als Modell das Codesystem und die Codesystem-Language um die Daten in Klassen und Beziehungen abzubilden. Weil sowohl für das Codesystem, als auch für den Domänenmodell-Editor das gleiche Modell verwendet wird, wird durch Bearbeitung des Modells im UML-Editor auch direkt das Codesystem verändert. Dies ist besonders bei großen Projekten nützlich, um zusätzliche Informationen aus dem Codesystem gewinnen zu können.

2. Technologie

Die Software QDAcity ist eine klassische Client-Server Anwendung. Der Server wird auf AppEngine (Google Inc., 2017a) betrieben, welches ein von Google betriebener Service für Server- und Webanwendungen ist.

2.1. Google App Engine (Backend)

Die Google AppEngine Cloud besteht aus folgenden Komponenten (Siehe Abbildung 1). Der Load Balancer (Google Inc., 2017g) verteilt die Anfragen auf die verschiedenen Instanzen der AppEngine Anwendung. Die Cloud Endpoints (Google Inc., 2017b) versenden und empfangen die Anfragen an die Clients. Datastore (Google Inc., 2017d) und Memcache (Google Inc., 2017e) sind für das Speichern der Daten zuständig. Die Task Queues werden in AppEngine für asynchrone Hintergrundaufgaben verwendet (Google Inc., 2017f).

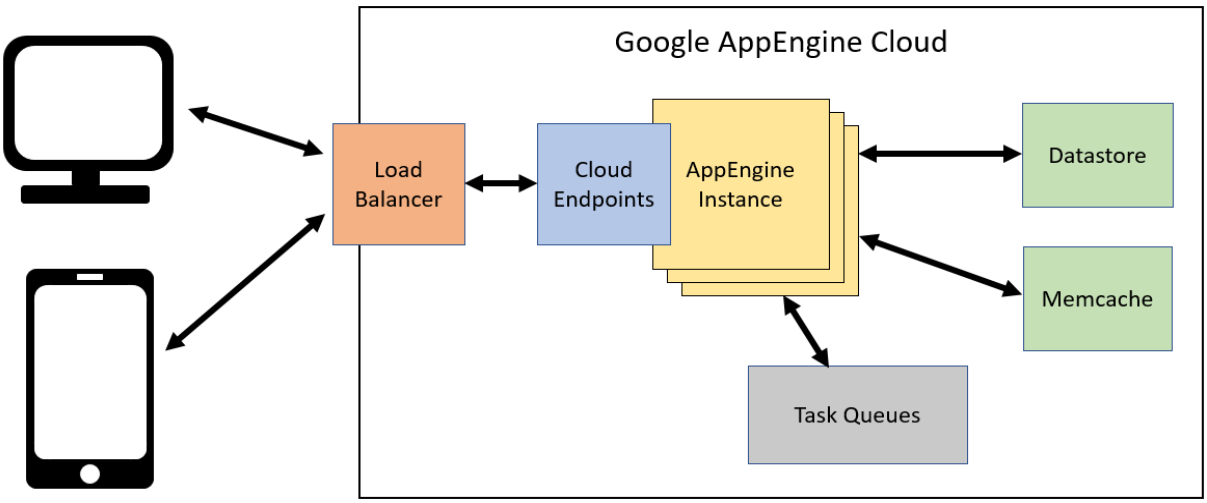


Abbildung 1: Vereinfachte Ansicht der Google AppEngine Cloud Architektur

Der Sever kommuniziert mit dem Client über die Cloud Endpoints, welche ein Representational State Transfer (REST) (Jakl, 2005) Interface bereitstellen. Für die Übertragung der Daten wird das Hypertext Transfer Protocol Secure (HTTPS), sowie als Übertragungsformat die JavaScript-Object-Notation (JSON) verwendet.

2.1.1. Datastore

Der Google AppEngine Datastore ist eine “Not only structured query language” (NoSQL) Datenbank. Das Datenmodell der Datenbank wird durch Annotationen der Java Data Objects JDO (Apache, 2015) definiert. Elemente in der Datenbank werden Entitäten genannt und werden über einen Key eindeutig zugeordnet. Die Entitäten sind, ähnlich wie die Ordner-Struktur im Dateisystem, in einer

hierarchischen Struktur gespeichert. Der Speicher wird deshalb auch „NoSQL document database“ genannt (Google Inc., 2017d).

2.1.2. Memcache

Der Google AppEngine Memcache ist ein über alle AppEngine Instanzen verteilter in-memory Cache. Dieser Cache bietet signifikant schnellere Zugriffszeiten als der Datastore der Google Cloud Platform (GCP). Es können beliebige Daten zwischengespeichert werden, d.h. sowohl einfache Werte, als auch komplexe Objekte können im Memcache abgelegt werden. Der Memcache wird besonders bei Operationen, die häufig auf dieselben Objekte im Datastore zugreifen, eingesetzt. QDAcity benutzt den Memcache im best-effort Modus (Shared Memcache), d.h. es wird versucht alle Daten so lange wie möglich bereit zu halten. Wenn der Cache allerdings wenig Speicher zur Verfügung hat oder neue Elemente eingefügt werden, werden alte Elemente aus dem Cache gelöscht. Das bedeutet, es gibt keine Garantie, wie lange die Objekte im Memcache zur Verfügung stehen. Wird der „Dedicated Memcache“ (kostenpflichtig) verwendet, gibt es die Möglichkeit die Daten mit einem Auslauf-Datum (expiration date) zu versehen. Der Cache sorgt dann dafür, dass die Daten für die angegebene Zeit verfügbar sind (Google Inc., 2017e).

2.2. Frontend

2.2.1. React

Für die Entwicklung des Frontends wird der Javascript Sprachstandard ECMA-Script 2016 (ES6) (ECMA International, 2016) verwendet und im Build-Prozess mit Babel (Babel, 2017) in ECMA-Script 2015 (ES5) (ECMA International, 2015) übersetzt. Bei der Gestaltung der Benutzeroberflächen-Elemente wird die von Facebook entwickelte Software-Bibliothek React (Facebook Inc., 2017) eingesetzt. Durch die Verwendung von React wird die Anwendung in einzelne gekapselte Komponenten zerlegt. Diese Komponenten werden deklarativ implementiert (Llody, 1994). React’s wichtigstes Merkmal ist, wie Manipulationen am Domain-Object-Model (DOM) gehandhabt werden. Anstatt das DOM direkt zu

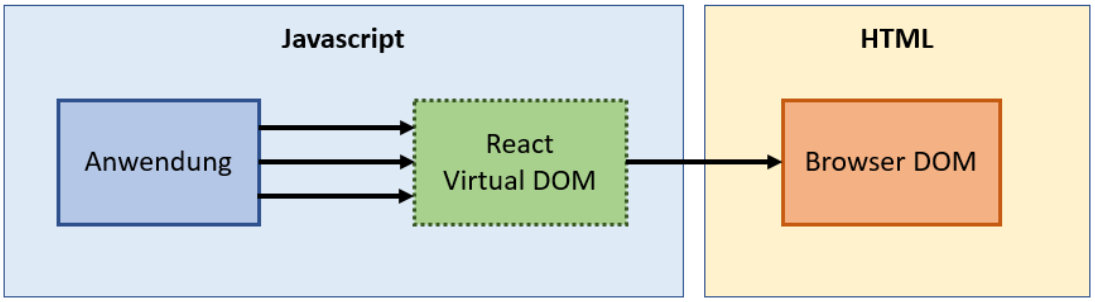


Abbildung 2: React Manipulation des Domain-Object-Models (DOM). Änderungen der Anwendung werden vom React Virtual-DOM gebündelt und verglichen und nur notwendige Operationen werden ausgeführt.

verändern, verwendet React eine abstrahierte Form des Standard-DOM, das Virtual-Domain-Object-Model. Durch die Einführung dieses Virtual-DOMs wird bei Änderungen der Anwendung ein Diff berechnet und nur die minimal notwendigen Operationen werden von React auf dem eigentlichen DOM ausgeführt (Siehe Abbildung 2) (Fedosejev, 2015).

2.2.2. Styled Components

Mit Styled-Components können kleine, und leicht testbare React-Komponenten erstellt werden. Genau wie bei React basiert der Grundgedanke von Styled-Components auf wiederverwendbaren Komponenten (Styled Components, 2017). Styled Components abstrahieren die Verbindung zwischen Komponenten und Styles. Dadurch muss der CSS-Code nicht mehr in einer separaten CSS-Datei geschrieben werden, sondern kann direkt in der Komponente bereitgestellt werden. Besonders die Eigenschaft, dass Styled-Components mit CSS und nicht mit inline-styles arbeitet, ist ein wesentlicher Unterschied zu vergleichbaren Bibliotheken wie Radium (Formidable Labs, 2017) oder Aphrodite (Khan Academy, 2017). Über einen Theme-Provider kann für Styled-Components ein Design-Schema übergeben werden, welches konsistentes Gestalten der Komponenten erleichtert.

3. Anforderungsprofil

Die folgenden Kapitel beschreiben die funktionalen und nicht-funktionalen Anforderungen, welche der Domänen-Model-Editor in QDAcity erfüllen muss.

3.1. Funktionale Anforderungen

3.1.1. Implementierung der Codesystem-Language

Um das Codesystem in ein UML-konformes Modell abbilden zu können, müssen die Codes in QDAcity mit Meta-Informationen einer Codesystem-Language attributierbar sein. Diese Codesystem-Language wurde bereits von Salow (2014) definiert und ermöglicht die Kategorisierung der Codes. Anhand der Codesystem-Language soll das Codesystem in ein UML-konformes Modell abgebildet werden können, welches dann in einem UML-Editor dargestellt werden kann. Dafür müssen auch die von Salow (2014) definierten Mapping-Regeln implementiert werden.

3.1.2. Integration in das bestehende System

Der UML-Editor muss in die bestehende Anwendung integriert werden. Das bedeutet, bei z.B. Änderungen am Codesystem, muss dafür gesorgt werden, dass der Zustand anderer Komponenten konsistent bleibt.

Zusätzlich soll der Nutzer Codes aus der Codesystem-Komponente des Coding-Editors in QDAcity per Drag-and-Drop in den UML-Editor ziehen können. Dadurch sollen die Codes direkt als UML-Klasse abgebildet werden.

In der Codesystem-Komponente sollen die Codes, die im UML-Modell abgebildet werden, farbig hervorgehoben werden.

3.1.3. Kodierung von Beziehungen

Eine weitere Anforderung ist, dass nicht nur Klassen, sondern auch Beziehungen, Attribute und Methoden als Codes zum Text zurückverfolgbar sein sollen. Das bedeutet, dass in den Textdokumenten von QDAcity auch die Beziehungen zwischen Codes kodierbar sein müssen.

3.1.4. Bearbeiten und Verändern des UML-Klassen-Diagramms

Das Codesystem muss durch den UML-Editor veränderbar sein. Dies umfasst folgende Funktionen:

- Hinzufügen/Entfernen von UML-Klassen (Codes)
- Hinzufügen/Entfernen von Beziehungen zwischen den Klassen
- Hinzufügen/Entfernen von Attributen und Methoden

3.1.5. Funktionen zur Verbesserung des Arbeitsflusses

Um mit dem UML-Editor auch bei größeren Projekten sinnvoll arbeiten zu können, werden folgende Funktionen benötigt um den Arbeitsfluss zu verbessern.

- Layout: Automatisches Anordnen der Komponenten (UML-Klassen) für eine bessere Übersicht.
- Zoom: Vergrößern und Verkleinern der Ansicht.
- Panning: Verschieben der Ansicht in horizontaler und vertikaler Richtung.

3.2. Nicht-funktionale Anforderungen

Bei der Auswahl der nicht-funktionalen Anforderungen habe ich mich an die Norm ISO/IEC 25010:2011 (ISO, 2011) für Qualitätsmerkmale in der Softwareentwicklung gehalten. Die Norm teilt die Anforderungen in die folgenden acht Kategorien in Tabelle 1 auf.

Kategorien	Sub-Kategorien	
Functional Suitability	Functional completeness	Functional correctness
	Functional appropriateness	
Performance Efficiency	Time behaviour	Capacity
	Resource utilization	
Compatibility	Co-existence	Interoperability
Usability	Learnability	User error protection
	Appropriateness recognizability	User interface aesthetics
	Operability	Accessibility
Reliability	Maturity	Recoverability
	Availability	Fault tolerance
Security	Confidentiality	Accountability
	Integrity	Authenticity
	Non-repudation	
Maintainability	Modularity	Analysability
	Reusability	Modifiability
Portability	Installability	Adaptability
	Replaceability	

Tabelle 1: Nicht-funktionale Anforderungen nach ISO/IEC 25010:2011

Für die Bestimmung der Anforderungen dieser Arbeit habe ich mich auf die wichtigsten Anforderungen beschränkt und nicht den ganzen Anforderungskatalog der ISO/IEC 25010:2011 ausgewertet. D.h., die Qualitätsmerkmale, die im nachfolgenden nicht näher betrachtet werden, wurden mit einer niedrigen Priorität für diese Arbeit eingestuft.

3.2.1. Funktionale Angemessenheit (functional appropriatness)

Dieses Kriterium besagt, dass der Nutzer nur mit den Notwendigen Schritten konfrontiert werden soll, die auch für die Erfüllung einer bestimmten Aufgabe notwendig sind. Schritte die im Hintergrund passieren können und für den Nutzer irrelevant sind, sollen auch vor dem Nutzer verborgen werden. Konkrete Anforderungen für diese Arbeit sind:

1) Beim Ändern der Codesystem-Language Einträge für einen Code sollen die Änderungen direkt beim Auswählen gespeichert werden, d.h. es soll keinen extra Button zum Speichern geben.

2) Beim Hinzufügen einer neuen Klasse im UML-Editor wird ein neuer Code erstellt, für den der Nutzer lediglich einen Namen auswählen muss. Die Einträge der Codesystem-Language sollen vom System automatisch ausgewählt werden.

3) Das Hinzufügen neuer Klassen im UML-Editor darf nicht mehr als 3 Aktionen benötigen.

4) Das Hinzufügen neuer Beziehungen zwischen Klassen im UML-Editor darf nicht mehr als 4 Aktionen benötigen.

3.2.2. Zeitverhalten (time behaviour)

Die Benutzerschnittstelle muss schnell auf Nutzer-Eingaben reagieren und Aufgaben in angemessener Geschwindigkeit ausführen, weil lange Reaktionszeiten den Arbeitsfluss des Nutzers stören, bzw. beeinträchtigen können. Der UML-Editor in QDAcity benötigt zwar keine besondere Effizient, dennoch darf kein Arbeitsschritt in QDAcity länger als 1 Sekunde benötigen.

3.2.3. Kapazität/Skalierbarkeit (capacity)

Um große Projekte mit mehr als 50+ Klassen zu unterstützen, werden Funktionen benötigt, die den Arbeitsfluss des Nutzers erleichtern.

1) Vereinfachte Ansicht: Es soll die Möglichkeit geben, die Attribute und Methoden der Klassen auszublenden und nur die Klassen-Namen anzuzeigen.

2) Es soll eine Funktion geben, über die das System das Layout der Klassen neu anordnet. Dadurch sollen die Überschneidungen von Kanten minimiert werden.

3) Der Nutzer soll die Möglichkeit haben, sich das gesamte Klassen-Diagramm anzeigen zu lassen. Dafür muss die Ansicht entsprechend verschoben und verkleinert/vergrößert werden, sodass alle Klassen zu sehen sind.

4) Wenn Codes in anderen Komponenten selektiert werden (z.B. Codesystem-Komponente), soll die zugehörige Klasse im UML-Diagramm angezeigt und hervorgehoben werden.

3.2.4. Erlernbarkeit (learnability)

Um eine leichte Erlernbarkeit des Editors für neue Anwender zu ermöglichen, ist es wichtig auf eine einheitliche und intuitive Gestaltung zu achten. Dies bedeutet, dass überall die gleichen Styles für Buttons und andere Steuerelemente verwendet werden müssen. Weil unterschiedliche Nutzer verschiedene Herangehensweisen, bzw. Angewohnheiten haben, ist es sinnvoll für wichtige Funktionen auch unterschiedliche Lösungswege anzubieten. Zusätzlich sollen alle Buttons und Bedienelemente mit Tooltipps versehen werden.

3.2.5. Fehler-Prävention (user error protection)

Damit Anwender nicht aus Versehen wichtige Informationen löschen oder irreversible Änderungen durchführen, soll dies durch zusätzliche Maßnahmen verhindert werden. Wenn Codes, UML-Klassen oder Beziehungen gelöscht werden, muss der Nutzer vorher über Dialoge die Durchführung bestätigen.

3.2.6. Vertraulichkeit (confidentiality)

Nutzer ohne gültigen Zugang, oder Nutzer ohne entsprechende Berechtigung sollen nicht auf die jeweiligen Daten und Funktionen zugreifen dürfen.

3.2.7. Modularität (modularity)

Der UML-Editor der in dieser Arbeit entsteht, soll möglichst abgekoppelt vom restlichen System als eigene Komponente entwickelt werden. Zusätzlich sollen alle Komponenten, aus denen der UML-Editor besteht auch austauschbar und lose gekoppelt sein. Das bedeutet auch, dass eventuell verwendete externe Libraries ohne großen Aufwand durch andere Komponenten ersetzt werden können.

4. QDAcity

Dieses Kapitel erklärt den Aufbau, sowie die wichtigsten Funktionen und Eigenschaften von QDAcity.

4.1. Projekte

Projekte sind eine zentrale Komponente in QDAcity. Ein Projekt besitzt ein Codesystem, welches die Codes verwaltet. Zu einem Projekt können beliebig viele Teilnehmer hinzugefügt werden und Dokumente erstellt werden.

4.2. Dokumente

Zu jedem Projekt können entweder Dokumente hochgeladen werden, oder direkt neue Dokumente erstellt und mit Text befüllt werden. In den Dokumenten können dann Textstellen mit den selbstdefinierten Codes markiert werden. Die Abbildung 3 zeigt einen Ausschnitt aus einem Text-Dokument mit markierten Textstellen.

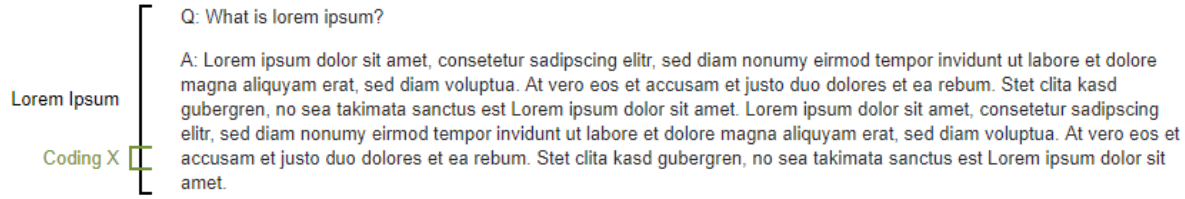


Abbildung 3: Ausschnitt eines Dokuments mit Kodierung

4.3. Codesystem

In QDAcity ist jedem Projekt ein eigenes Codesystem zugeordnet. Das Codesystem beinhaltet alle Codes die für Kodierungen verwendet werden können. Das Codesystem ist hierarchisch, wie bei einer

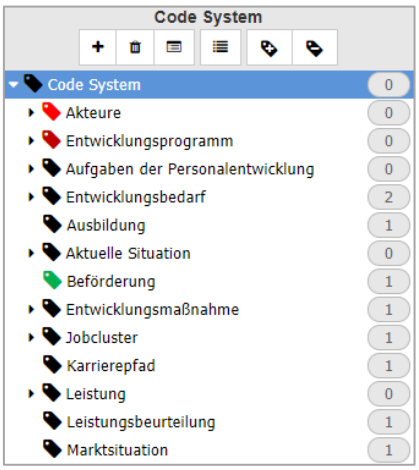


Abbildung 4: Ausschnitt des Codesystems aus einem QDAcity-Projekt

Ordnerstruktur aufgebaut. Es gibt genau einen Wurzel-Code der nicht entfernt werden kann. In Abbildung 4 sieht man einen Ausschnitt eines Codesystems.

Das Klassen-Diagramm in Abbildung 5 zeigt das Verhältnis zwischen Codesystem und Code. Die hierarchische Struktur der Codes wird durch die Attribute parentId und subCodeIds definiert. Jeder Code hat eine „id“ und eine „codeId“. Die „id“ wird benötigt um eine eindeutige Referenz auf den Code in der Datenbank zu haben. Die „codeId“ identifiziert den Code eindeutig innerhalb des Codesystems. Das bedeutet, dass Codes aus verschiedenen Codesystemen die gleiche „codeId“ besitzen können, aber niemals die gleiche „id“. Jeder Code besitzt außerdem ein CodeBookEntry, welches Informationen zur Verwendung des Codes bei der Kodierung speichert.

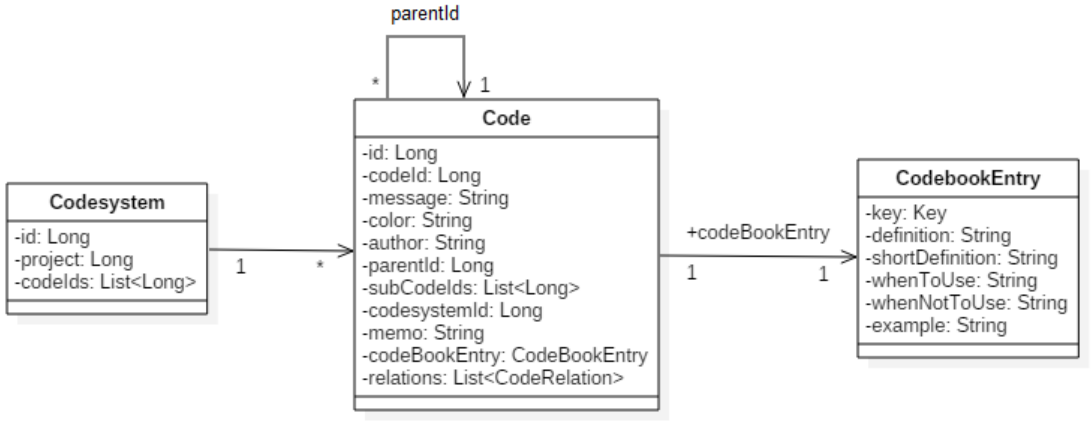


Abbildung 5: UML-Klassen-Diagramm des Codesystems

4.4. Beziehungen zwischen Codes

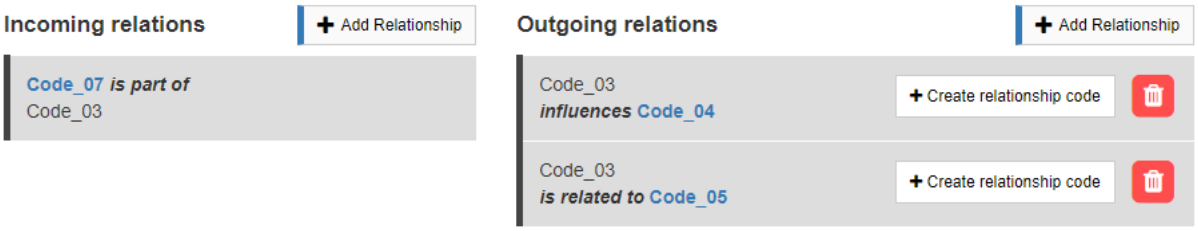


Abbildung 6: Code-Relations in QDAcity

Zwischen Codes können in QDAcity Beziehungen definiert werden. Abbildung 6 zeigt einen Ausschnitt der Benutzeroberfläche. In der linken Hälfte werden die eingehenden Beziehungen des Codes angezeigt, d.h. Beziehungen bei denen das Ziel der ausgewählte Code ist. Auf der rechten Seite werden Beziehungen angezeigt, bei denen der ausgewählte Code der Start-Code ist. Abbildung 7 zeigt das Verhältnis zwischen Code und CodeRelation. Der Ziel-Code der CodeRelation wird über das Feld „codeId“ bestimmt und der Start-Code wird durch den Key definiert.

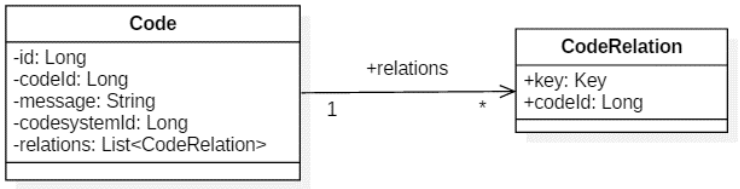


Abbildung 7: Beziehung zwischen Code und CodeRelation

4.5. Codesystem-Language

Die Codesystem-Language wird benötigt um das Code-System in ein UML-Klassendiagramm umzuwandeln. Jeder Code kann beliebig viele Codesystem-Language-Elemente besitzen (Siehe Abbildung 8).

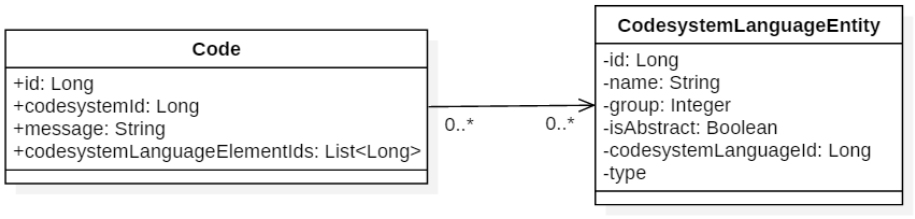


Abbildung 8: Verhältnis zwischen Code und Codesystem-Language (UML-Klassendiagramm)

Die Code-System-Language-Elemente sind als Baumstruktur aufgebaut. Die Klasse CodesystemLanguageRelation (Abbildung 9) markiert dabei immer ein Vater- (source) und Kind-Element (destination). Zusätzlich besteht die Struktur der Codesystem-Language-Einträge nicht aus einem einzigen Baum, sondern aus mehreren. Die einzelnen Bäume werden durch das Attribut „group“ unterteilt. Alle Elemente mit der gleichen group-id gehören zur gleichen Gruppe, wobei eine Gruppe auch aus mehreren Bäumen bestehen kann. Zu jedem Code können zwar beliebig viele Codesystem-Language-Elemente zugeordnet werden, allerdings kann aus jeder Gruppe nur ein Element ausgewählt werden. Jedes Codesystem-Language-Objekt hat außerdem ein Attribut „type“ vom Enum CodesystemLanguageType. Dieser Wert gibt an ob das Element für Codes (PROPERTY) oder für CodeRelations (RELATIONSHIP) ausgewählt werden kann. Durch die ID „codesystemLanguageId“, können verschiedene Codesystem-Sprachen definiert werden. Alle Elemente mit der gleichen ID gehören zur gleichen Codesystem-Language. So könnte man für unterschiedliche Projekte mit unterschiedlichen Codesystemen auch unterschiedliche Codesystem-Sprachen definieren. Im Rahmen dieser Masterarbeit wird aber nur mit einer Codesystem-Lanuage gearbeitet.

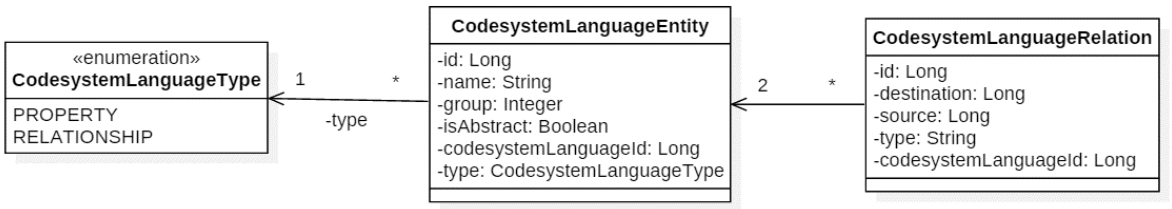


Abbildung 9: UML-Klassendiagramm der Codesystem-Language Klassen

In Abbildung 10 sieht man die Einträge der CodeSystemLanguageEntity aufgelistet. Die Einträge wurden aus einer vorherigen Masterarbeit „A MetaModel for Codesystems“ (Salow, 2016) übernommen, genauso wie die Einträge für CodeRelations (Siehe Abbildung 11). Die Einträge sind in zwei Gruppen unterteilt und die einzelnen Bäume sind farbig hervorgehoben.

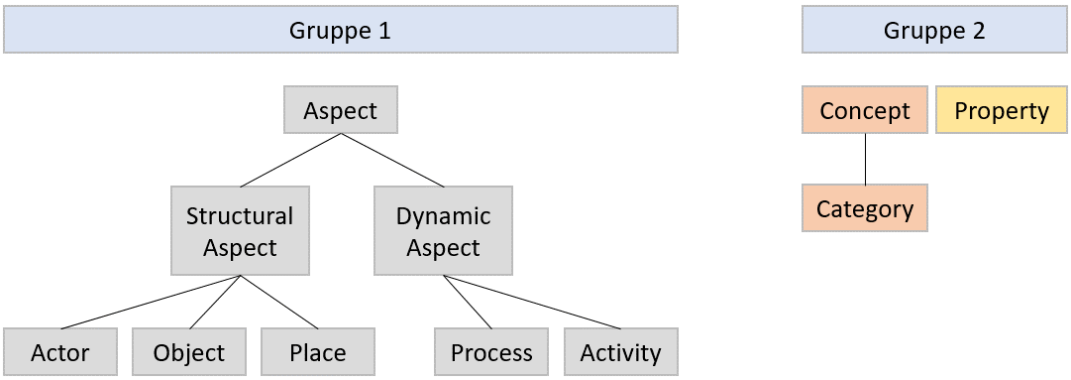


Abbildung 10: Einträge der Codesystem-Language für Codes

Einer Code-Relation, also einer Beziehung zwischen zwei Codes kann ebenfalls ein Codesystem-Language-Eintrag hinzugefügt werden, allerdings jeder Relation nur ein Eintrag (1:1-Beziehung). Die Elemente werden in Abbildung 11 dargestellt.

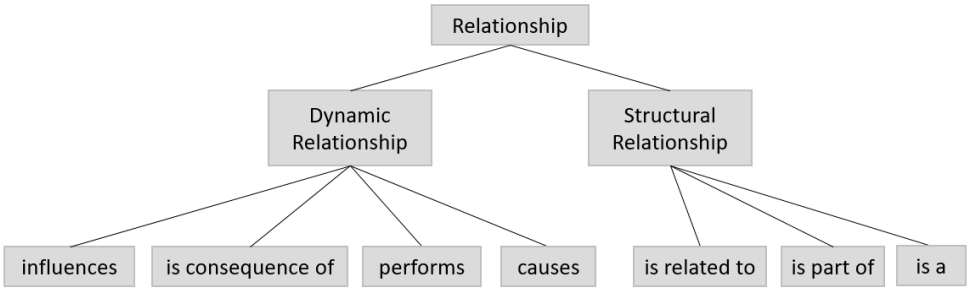


Abbildung 11: Codesystem-Language Einträge für Code-Relationships

5. Architektur

In diesem Kapitel werden die wichtigsten Komponenten des UML-Editors genauer erläutert. In Abbildung 12 sind die einzelnen Komponenten stark vereinfacht dargestellt und werden im Folgenden näher erläutert.

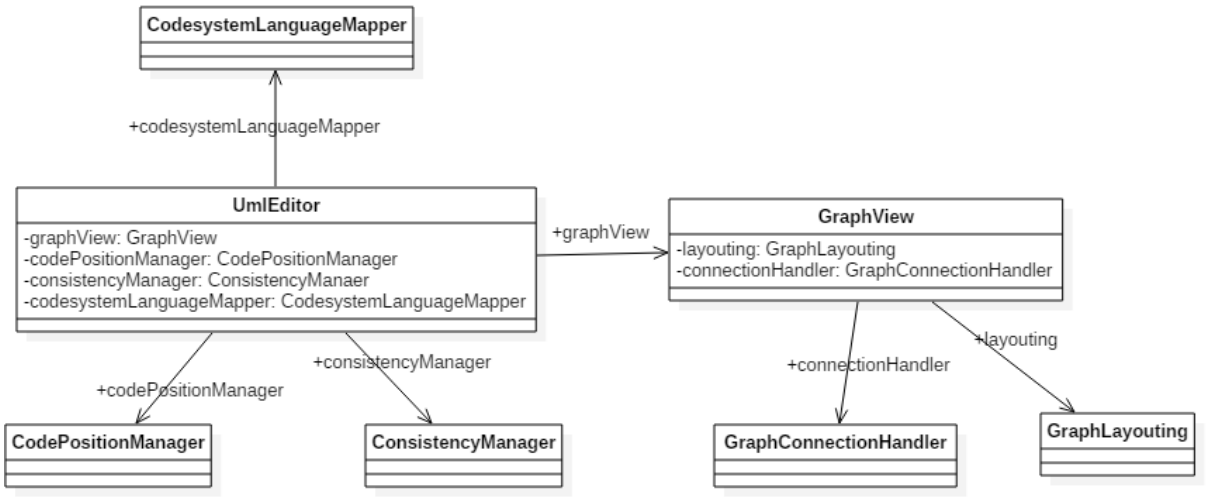


Abbildung 12: Klassen-Diagramm der wichtigsten Komponenten des UML-Editors

5.1. UmlEditor

Die Klasse UmlEditor bietet eine Schnittstelle zum UML-Editor an und ist als Fassade für diese Komponente implementiert. Diese Klasse kennt alle Komponenten des UML-Editors und leitet notwendige Funktionsaufrufe von außen an die jeweiligen Klassen weiter.

5.2. GraphView

Die Komponente GraphView abstrahiert das Zeichnen des Graphen, bzw. des Editors. Sie dient auch als Wrapper für die Software-Bibliothek mxGraph (JGraph, 2017a), die zum Zeichnen des Modells verwendet wird (Siehe Kapitel 6.1). Die Klasse bietet Methoden zum Hinzufügen/Entfernen von Nodes (UML-Klassen), Hinzufügen von Attributen, Methoden, Kanten, etc. Auch Funktionen für Zoom, Panning (Verschieben der Ansicht) oder das Auswählen (Focus) von Elementen werden implementiert (Siehe Abbildung 13).

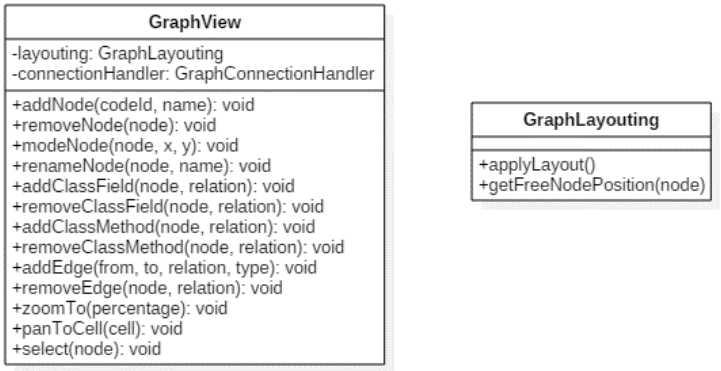


Abbildung 13: Methoden der Klassen GraphView und GraphLayouting

5.2.1. GraphConnectionHandler

Die Klasse GraphConnectionHandler kapselt die Funktionalität, die benötigt wird um neue Kanten mittels Drag-and-Drop im Editor hinzuzufügen. Diese Klasse wurde eingeführt, um Funktionalität und Aufgaben der Klasse GraphView zu reduzieren.

5.2.2. GraphLayouting

Für das Layouting der UML-Klassen wir die Klasse GraphLayouting verwendet. So kann z.B. mit applyLayout() automatisch ein neues Layout berechnet werden, oder mit getFreeNodePosition() sichergestellt werden, dass sich beim Einfügen keine Klassen überlappen (Sehe Abbildung 13).

5.3. CodesystemLanguageMapper

Das Codesystem-Language-Mapping, also das Übersetzen der Codes und Relations in ein geeignetes UML-Klassen-Diagramm wird von der Klasse CodesystemLanguageMapper übernommen. Sie definiert Mapping-Regeln die das Mapping steuern und auf die Codes angewendet werden können. Wie dieses System funktioniert, wird in Kapitel 6.2 beschrieben.

5.4. ConsistencyManager

Der ConsistencyManager sorgt dafür, dass der UML-Editor nach Änderungen am Codesystem immer den aktuellsten Stand zeigt. Dies ist dann besonders wichtig, wenn Änderungen nicht im UML-Editor geschehen, sondern in anderen Komponenten des Systems. Im Kapitel 6.1.3 wird die Funktionsweise des ConsistencyManagers genauer erläutert.

5.5. CodePositionManager

Für alle UML-Klassen im Editor muss die Position der Koordinaten in der Datenbank gespeichert werden. Die Klasse UmlCodePosition in Abbildung 14 erfüllt diese Aufgabe. Weil die Position nicht direkt mit dem Code verknüpft ist, wird jeweils die codeId und codesystemId gespeichert, um die UmlCodePosition eindeutig zu dem dazugehörigen Code zuordnen zu können. Die Klasse CodePositionManager verwaltet die Zugriffe auf die UmlCodePosition Einträge.

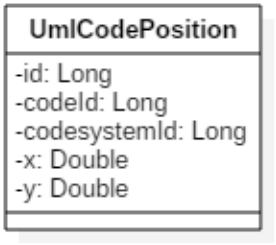


Abbildung 14: Die UmlCodePosition repräsentiert die Position, bzw. die Koordinaten eines Codes im Editor

6. Implementierung

In diesem Kapitel werden Details zur Implementierung des UML-Editors genauer erläutert.

6.1. Zeichnen des UML-Diagramms

Der zentralste Teil des UML-Editors ist das Zeichnen des UML-Graphen. Diese Komponente zeichnet die UML-Klassen mit Attributen und Methoden, sowie die Beziehungen zwischen den UML-Klassen. In diesem Kapitel wird diskutiert wie das Zeichnen des UML-Graphen realisiert wurde. Um die beste Entscheidung für die Implementierung treffen zu können, habe ich zunächst eine Open-Source-Bibliothek evaluiert und danach das Ergebnis mit dem Aufwand einer eigenen Entwicklung verglichen.

6.1.1. Evaluation verfügbarer Open-Source-Bibliotheken

In Tabelle 2 habe ich die Entscheidungskriterien für die Evaluation einer Open-Source-Bibliothek zusammengefasst und für zwei gängige Javascript-Software-Bibliotheken ausgewertet.

	mxGraph 3.7.4 (JGraph, 2017a)	**JointJS 2.0.0** (Client IO, 2017a)
Open Source Ist das Projekt Open-Source? Welche Lizenz wird verwendet?	Apache 2.0	Mozilla Public License (MPL) 2.0
Kosten Fallen Kosten für die Nutzung an?	Kostenlos	Kostenlos
Dokumentation Ist eine Dokumentation der Schnittstellen vorhanden?	Dokumentation der Schnittstellen Codebeispiele und Projektbeispiele für die Verwendung der Funktionen (JGraph, 2017b)	Demo-Projekte, Tutorials und Dokumentation vorhanden (Client IO, 2017b)
Abhängigkeiten/Dependencies	Keine	jQuery: 3.1.1 Lodash: 3.10.1 Backbone: 1.3.3
Browser Support Werden alle gängigen Browser unterstützt?	IE 11, Chrome 32+, Firefox 38+, Safari 7.1.x, 8.0.x, 9.1.x und 10.0.x, Opera 20+ (Siehe JGraph, 2017a)	Nur Browser die SVG unterstützen (konkrete Versionsnummern werden

		von JointJS nicht angegeben): - Latest Google Chrome (including mobile) - Latest Firefox - Safari (including mobile) - IE 9+ (Siehe Client IO, 2017a)
Funktionalität Panning/Scrolling Zoom	Panning/Scrolling durch Mausbewegung Zoom mit dem Mausrad	Panning/Scrolling durch Mausbewegung Zoom mit dem Mausrad
Layouting-Funktionen Automatisches Anordnen aller Komponenten	Bietet vordefinierte Layouting-Algorithmen und unterstützt auch selbst-implementierte Algorithmen.	Bietet vordefinierte Layouting-Algorithmen, die für den UML-Editor aber keine Verwendung finden. Unterstützt selbst-definierte Algorithmen.
Automatisches Verbinden von Kanten Können im Graphen neue Kanten per Drag & Drop hinzugefügt werden?	Wird unterstützt	Wird unterstützt
Vorgefertigte UML-Komponenten Unterstützt die Software UML-Komponenten? Z.B. vorgefertigte Klassen oder Beziehungen nach UML-Standard.	Unterstützt vordefinierte Kanten für Beziehungen, wie Generalisierung, Aggregation, usw. UML-Klassen-Objekte werden nicht bereitgestellt, können aber selbst implementiert werden.	Sowohl Kanten für Beziehungen, als auch UML-Klassen werden unterstützt

Tabelle 2: Evaluation einer Open-Source-Bibliothek für das Zeichnen des UML-Diagramms

Eine Anforderung an die Third-party-library war, dass als Lizenz keine Copy-Left-Lizenz verwendet werden darf, da sonst auch der Quell-Code von QDAcity veröffentlicht werden muss. Die Apache 2.0 Lizenz von mxGraph beinhaltet keine Copy-Left Klausel und die Mozilla-Public-Lizenz 2.0 von JointJS

erfordert das Copy-Left nur, wenn der Quell-Code von JointJS selbst geändert wird. Beide Projekte unterstützen die gängigsten Browser und bieten ausreichend Dokumentation, während von mxGraph mehr Demo-Projekte und Code-Beispiele angeboten werden. Von mxGraph werden keine zusätzlichen Dependencies benötigt, JointJS benötigt hingegen jQuery, Lodash und Backbone. Von beiden Software-Bibliotheken wird die Grundfunktionalität, die für die Implementierung des UML-Editors benötigt wird, unterstützt. Bei den Layouting-Algorithmen für das Anordnen der UML-Klassen im Editor bietet mxGraph bessere Unterstützung.

Weil mxGraph keine zusätzlichen Dependencies benötigt, mehr Funktionalität bietet und über eine bessere Dokumentation (besonders Beispiel-Projekte) verfügt, habe ich mich gegen JointJS und für mxGraph entschieden. Im Folgenden Kapitel wird nun diskutiert, ob für die Implementierung des UML-Editors eine eigene React-Komponente entwickelt wird, oder die Software-Bibliothek mxGraph eingesetzt wird.

6.1.2. Evaluation einer möglichen Neuentwicklung

In Tabelle 3 werden die Vor- und Nachteile einer eigenen Implementierung als React-Komponente, bzw. der Einsatz der externen Software mxGraph dargestellt. Das gesamte Front-End von QDAcity wurde in React implementiert und deswegen würde sich eine Implementierung als React-Komponente gut in das Projekt einfügen. Zusätzliche müsste nur Funktionalität implementiert werden, die vom Projekt tatsächlich benötigt wird. Bei der Third-party-libraries mxGraph werden viele Funktionen angeboten, die vom UML-Editor nicht benötigt werden. Der größte Nachteil der eigenen Implementierung ist, dass alle Funktionen die mxGraph bietet, selbst implementiert und getestet werden müssen. Besonders das Testen und Warten der Software, welches bei mxGraph nicht notwendig ist, ist ein großer Faktor. Da also der Implememtierungs-Aufwand einer eigenen React-Komponente für das Zeichnen des UML-Editors sehr hoch ist und auch das Testen und Warten sehr viel Zeit beanspruchen würde, habe ich mich für die Verwendung von mxGraph entschieden.

	Eigene React-Komponente	mxGraph
Vorteile	- Deklarative Entwicklung in React, fügt sich gut in das Projekt ein - Keine Überflüssige Funktionalität (light-weight)	- Hoher Funktionsumfang - Dokumentation - Open source - Gut getestete Software (wird z.B. von draw.io verwendet)
Nachteile	- Hoher Arbeitsaufwand für die Implementierung aller benötigten Funktionen	- Imperative Implementierung - Überflüssige Funktionalität

	- Software muss gewartet und getestet werden	- Aufwand die Bibliothek in das imperative React-Projekt einzufügen

Tabelle 3: Vergleich Eigene React-Komponente vs. mxGraph bei der Implementierung des UML-Editors

6.1.3. Implementierung mit mxGraph

Für die Implementierung mit mxGraph wird die Kopmonente „GraphView“ verwendet. Diese kapselt alle Zugriffe auf mxGraph im UML-Editor. Dadurch ist die Third-party-library mxGraph auch jederzeit durch eine andere Komponente für das Zeichnen des Editors austauschbar. Weil mxGraph imperativ arbeitet, d.h. der Zustand kann nur über Funktions-Aufrufe geändert werden, muss der UML-Editor selbst die notwendigen Änderungen durchführen.

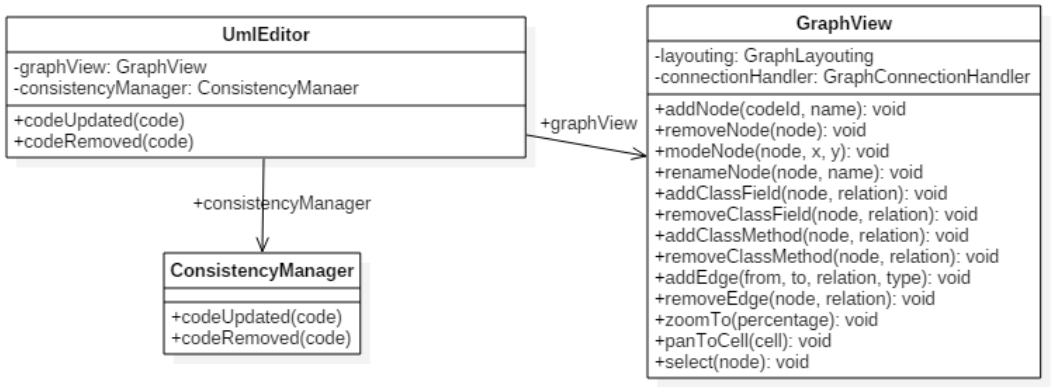


Abbildung 15: Überblick UmlEditor und ConsistencyManager

Im Beispiel in Abbildung 15 wird vereinfacht gezeigt wie der Update-Prozess des Editors funktioniert. Ein Code, der nicht im UML-Editor als UML-Klasse dargestellt wird (aufgrund des Codesystem-Language-Mappings, siehe Kapitel 6.2), wurde von einer anderen Komponente in QDAcity verändert und muss durch diese Änderung im UML-Editor dargestellt werden.

Zunächst wird die Methode codeUpdated() der Klasse UmlEditor aufgerufen und das aktuelle Code-Objekt als Parameter übergeben. Dieser Aufruf wird dann an die Klasse ConsistencyManger (Siehe Abbildung 16) weitergeleitet. Diese Klasse speichert immer den letzten Zustand eines Codes, bzw. einer CodeRelation und vergleicht dann ob das Update auch eine Änderung im UML-Editor zur Folge hat. Für die Überprüfung wird der CodesystemLanguageMapper verwendet (Siehe Kapitel 6.2). Dieser bestimmt anhand der Mapping-Regeln, was mit dem übergebenen Code passieren soll. In diesem Beispiel wird für die Evaluation des „alten“ Codes, also des Codes vor dem Update, der Wert „NOTHING“ zurückgegeben. D.h. der Code soll nicht im UML-Editor angezeigt werden. Danach wird der Code in seinem aktuellen Zustand (also nach dem Update) evaluiert und der Wert „CREATE_NODE“ wird zurückgegeben. Der Consistency-Manager vergleicht die beiden Aktionen und

übermittelt dem UmlEditor, dass eine neue UML-Klasse zum Editor hinzugefügt werden muss. Der UmlEditor ruft dann addNode(code) bei der Klasse GraphView auf und diese übernimmt das Ändern der Ansicht mithilfe von mxGraph. Die Klasse Consistency-Manager würde zusätzliche überprüfen, ob sich auch die Beziehungen (CodeRelation) des Codes geändert haben und für diese die notwendigen Aktionen ausführen. Dies wird im Sequenzdiagramm in Abbildung 16 zur Vereinfachung nicht dargestellt.

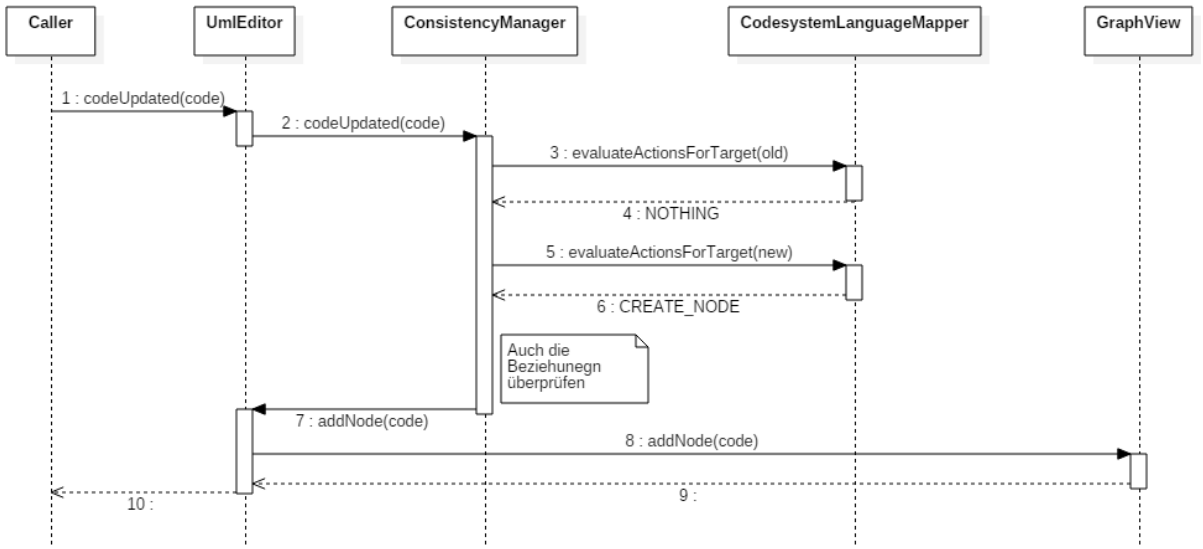
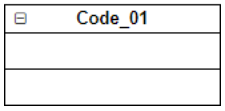
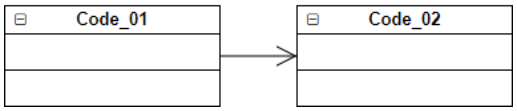
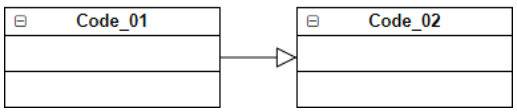
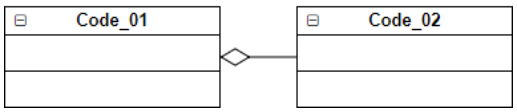
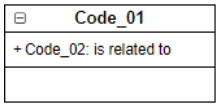


Abbildung 16: Update-Prozess des UML-Editors

6.2. Codesystem-Language-Mapping

6.2.1. Mapping der Codesystem-Language

Für die Umwandlung des Codesystems und der CodeRelations in ein UML-Klassen-Diagramm wurden sogenannte Mapping-Regeln definiert. Die Regeln wurden von Salow (2016) bereits erarbeitet und werden im Folgenden nochmal zusammengefasst. Der Begriff MetaModel hat hier die gleiche Bedeutung wie Codesystem-Language. Wenn eine der Regeln aus Tabelle 4 zutrifft wird die angegebene Aktion ausgeführt. Codes und CodeRelations mit Codesystem-Language-Einträgen für die es kein Mapping gibt, werden im UML-Editor nicht angezeigt.

Start-Code Codesystem-Language	**Ziel-Code** Codesystem-Language	**Relation** Codesystem-Language	**Mapping-Ergebnis**
Concept oder Category	-	-	Der Code wird als UML-Klasse angezeigt: 
Concept oder Category	Concept oder Category	is related to	Eine Beziehung vom Typ Assoziation wird angezeigt: 
Concept oder Category	Concept oder Category	is a	Eine Beziehung vom Typ Generalisierung wird angezeigt: 
Concept oder Category	Concept oder Category	is part of	Eine Beziehung vom Typ Aggregation wird angezeigt: 
Concept oder Category	Property und (Object, Place oder Actor)	is related to	Ein UML-Klassen-Attribut wird angezeigt: 

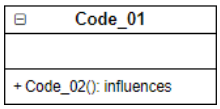
Concept oder Category	Property	influences	Eine UML-Klassen-Methode wird angezeigt: 

Tabelle 4: Codesystem-Language Mapping Regeln

6.2.2. Implementierung des Codesystem-Language-Mappings

Bei der Implementierung des Mappings werden die Mapping-Regeln auf Klassen abgebildet. Die Klasse „Rule“ (Siehe Abbildung 18) stellt eine solche Regel dar. Eine Regel besteht aus dem „targetType“, einer „condition“ und einer „action“. Der targetType ist ein Wert des Enums „Target“ und definiert, ob ein Code oder eine CodeRelation von der Regel gemappt werden soll. Die „condition“ hält die Bedingung die erfüllt sein muss, damit das Mapping durchgeführt wird. Die „action“ legt fest was passieren soll, wenn die Bedingung erfüllt ist (z.B. Erstellen einer neuen Beziehung). Mit condition und action sind hierbei die Klassen AbstractCondition und AbstractAction gemeint. Die Klassen Action und Condition bieten Methoden an um die jeweiligen Komponenten zu erstellen und werden verwendet um einen besser lesbaren Code beim Erstellen der Regeln (Rule) zu gewährleisten (Abbildung 17).

```
metaModelMapper.registerRule(
  Rule.create()
      .expect(Target.CODE)
      .require(Condition.or(
        Condition.hasCodesystemLanguageEntity('Category'),
        Condition.hasCodesystemLanguageEntity('Concept')
      ))
      .then(Action.createNode()));


metaModelMapper.registerRule(
  Rule.create()
    .expect(Target.RELATION)
    .require(Condition.and(
      Condition.hasCodesystemLanguageEntity('influences'),
      Condition.or(
        Condition.hasCodesystemLanguageEntity('Category', EvaluationTarget.SOURCE),
        Condition.hasCodesystemLanguageEntity('Concept', EvaluationTarget.SOURCE)
      ),
      Condition.hasCodesystemLanguageEntity('Property', EvaluationTarget.DESTINATION)
    ))
    .then(Action.createClassMethod()));
```

Abbildung 17: Code-Ausschnitt - Registrieren der Mapping-Regeln zur Laufzeit

In Abbildung 17 ist ein Code-Beispiel angegeben, wie Mapping-Rules zur Laufzeit erstellt werden können. Die erste Regel definiert, wann ein Code als UML-Klasse im Editor dargestellt werden soll. Es werden alle Codes, die als Codesystem-Language-Eintrag „Category“ oder „Concept“ haben abgebildet. Für das Mapping bietet die Klasse Rule die Methoden expect, require und then an (Siehe

Abbildung 18). Diese Methoden haben als Rückgabetyp jeweils den Typ Rule, weswegen die Aufrufe wie im Codebeispiel geschachtelt werden können.

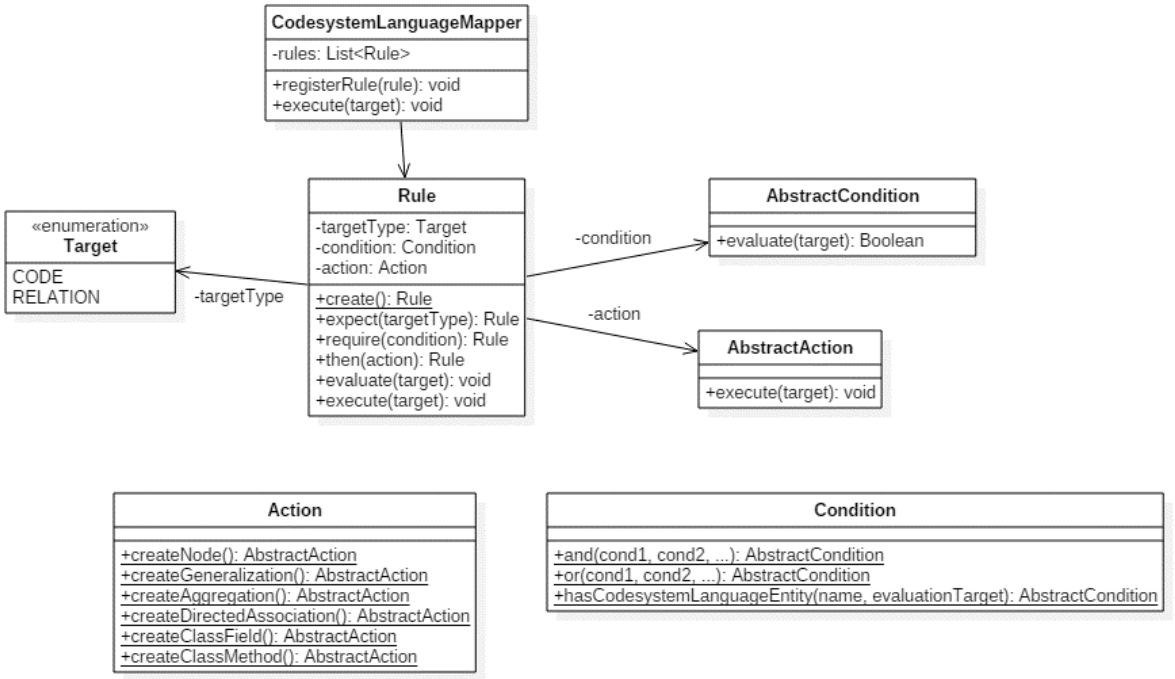


Abbildung 18: Klassendiagramm des Codesystem-Language-Mappings

Bei der Implementierung der Bedingungen einer Mapping-Rule wird die Klasse AbstractCondition eingesetzt. Sie definiert die Methode evaluate(target), welche als Argument den Code oder die Relation übergeben bekommt – je nachdem was für die Rule mittels expect() definiert wurde. Diese Methode gibt, wenn die Bedingung vom Code oder von der Relation erfüllt wird, true zurück. Um möglichst viele Szenarien für unterschiedliche Bedingungen abzudecken wurden verschiedene Klassen implementiert, die alle von AbstractCondition erben (Siehe Abbildung 19). Die Klassen AndCondition und OrCondition implementieren ein logisches UND bzw. ODER. Beide Klassen akzeptieren als Parameter mehrere Bedinungen vom Typ AbstractCondition. Die Klasse MultipleCondition dient als

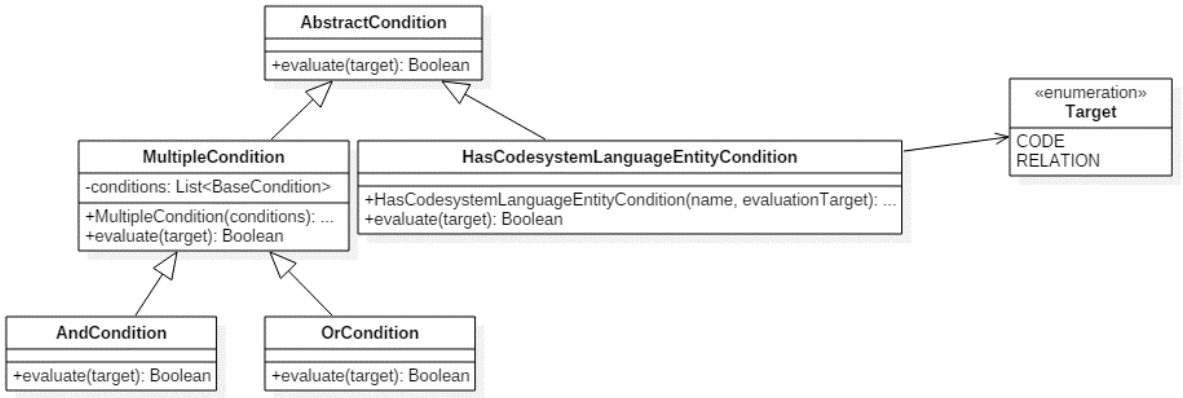


Abbildung 19: Klassen-Diagramm der Bedingungs-Klassen

Oberklasse und abstrahiert Code, der bei beiden Klassen gleich ist. Die letzte Bedingung HasCodeSystemLanguageEntityCondition prüft ob ein Code oder eine Relation eine Codesystem-Language-Entität mit einem bestimmten Namen besitzt. Optional kann als Parameter zusätzlich das EvaluationTarget definiert werden. Wenn bei der Mapping-Rule eine Relation ausgewertet wird, kann man mit diesem optionalen Parameter bestimmen, ob nicht die Beziehung, sondern der Start- bzw. Ziel-Code der Beziehung überprüft werden soll. Im Code-Beispiel in Abbildung 17 wird z.B. bei der zweiten Mapping-Rule überprüft, dass die Relation die Entität „influences“ besitzt, und der Start-Code der Relation die Entität „Concept“ oder „Category“. Dieses Konzept ist jederzeit um weitere Implementierungen von AbstractCondition erweiterbar, um weitere Funktionalitäten abzudecken.
Das gleiche Prinzip wird bei der Implementierung der Klasse AbstractAction verfolgt. Die Methode execute(target) führt das tatsächliche Mapping aus, d.h. das Bearbeiten des UML-Editors (Hinzufügen von Klassen, Kanten, Methoden, etc.). Ebenso wie bei den Bedingungen kann das System um weitere Aktionen, die von AbstractAction erben, erweitert werden (Siehe Abbildung 20Abbildung 20).

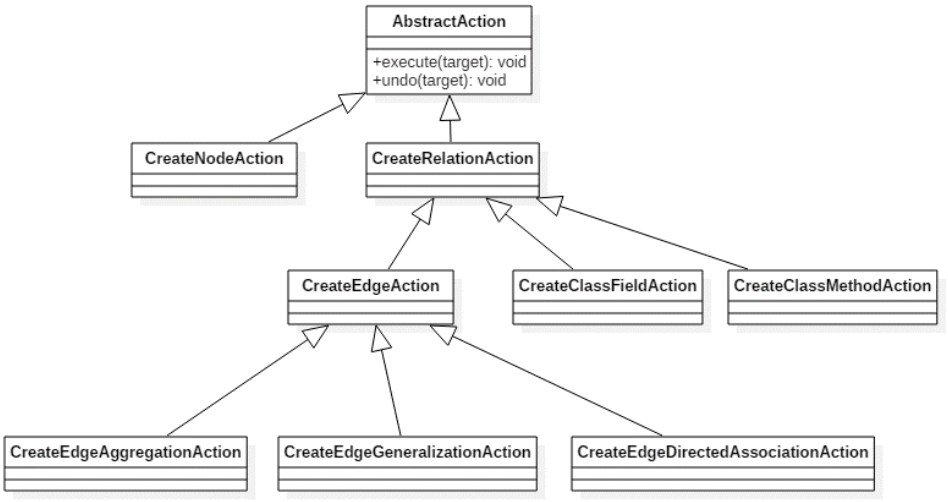


Abbildung 20: Klassen-Diagramm der Action-Klassen

Die Klasse CodesystemLanguageMapper kapselt das Mapping nach außen. Hier werden die Mapping-Rules mit registerRule(rule) registriert und verwaltet. Um das Mapping für einen Code oder eine Relation auszuführen, muss nur mapper.execute(code) ausgeführt werden. Das Sequenz-Diagramm in Abbildung 21 zeigt, welche Schritte beim Aufruf von execute() ausgeführt werden. In diesem Beispiel wird das Mapping eines Codes zu einer UML-Klasse gezeigt. Zur Vereinfachung wird hier auch nur eine Mapping-Rule ausgewertet. Beim Aufruf von execute() an der Klasse CodesystemLanuageMapper werden alle Mapping-Rules die auf das übergebene Objekt passen überprüft. D.h., je nachdem ob ein Code oder eine Relation übergeben wird, werden nur die Regeln ausgewertet, (bei denen mit expect() das Target (CODE oder RELATION) definiert wurde. Für die Auswertung einer Mapping-Rule ruft diese

die Condition auf, welche dann einen Boolean zurückgibt. Wenn dieser Wert wahr ist, also die Bedingung erfüllt ist, wird die Action aufgerufen. Im Beispiel in Abbildung 21 wird hier die CreateNodeAction ausgeführt, welche eine neue UML-Klasse zum Editor hinzufügt.

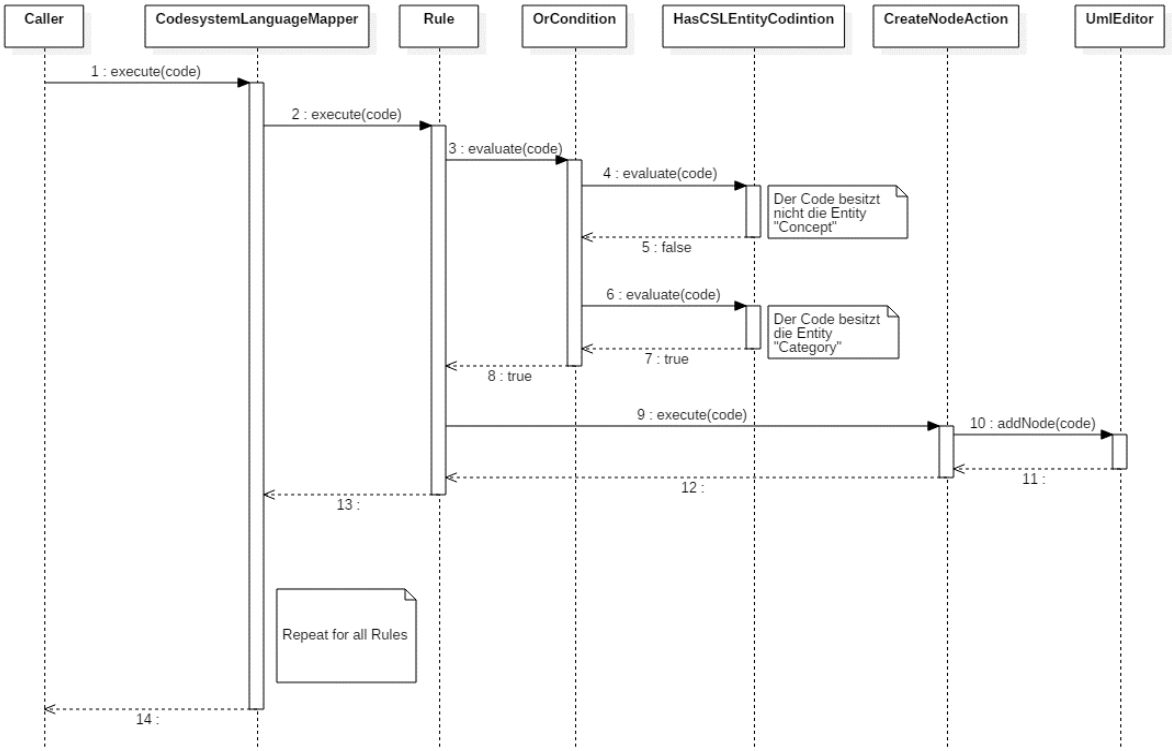


Abbildung 21: Aufrufe beim Mapping eines Codes

6.3. Kodierung von Beziehungen

Um Textstellen in QDAcity nicht nur mit Codes, sondern auch mit Beziehungen zwischen Codes markieren zu können, wurden sogenannte Relationship-Codes eingeführt. Ein Relationship-Code kann als Codesystem-Language nur Einträge vom Typ Relation (Siehe Kapitel 4.5) haben und kann selbst keine Beziehungen zu anderen Codes haben. In der Implementierung wurde die Klasse Code um das Attribut „relationshipCode“ erweitert, welches die zugehörige Beziehung speichert, falls es sich bei dem Code um einen Relationship-Code handelt (Siehe Abbildung 22).

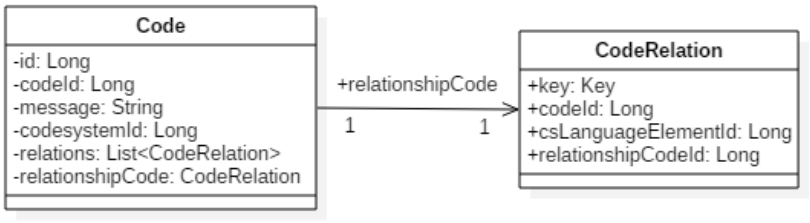


Abbildung 22: Klassen-Diagramm Relationship-Codes

7. Diskussion

7.1. Evaluation

In diesem Kapitel werden die Anforderungen, die in Kapitel 3 bestimmt worden, anhand der entwickelten Software ausgewertet.

7.1.1. Funktionale Anforderungen

Implementierung der Codesystem-Language

Die Funktionsweise und Implementierung der Codesystem-Language wurden bereits in Kapitel 6.2 ausführlich erläutert. Den Codes können über ein Menü die Codesystem-Language-Einträge hinzugefügt werden (Siehe Abbildung 23). Die Mapping-Regeln können dynamisch zur Laufzeit hinzugefügt und angepasst werden. Siehe Tabelle 4 für die Zusammenfassung der Mapping-Regeln.

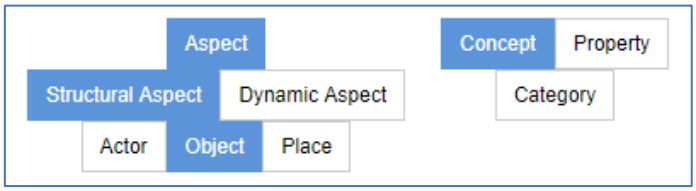


Abbildung 23: Codesystem-Language-Einträge eines Codes

Integration in das bestehende System

Um die Konsistenz der Daten im System sicherzustellen, wurde die Klasse ConsistencyManager entwickelt (Siehe Kapitel 5.4). Der ConsistencyManager sorgt dafür, dass Änderungen im UML-Editor an die anderen Komponenten des Systems weitergegeben werden, bzw. dass Änderungen in anderen Komponenten korrekt im UML-Editor verarbeitet werden.

Um existierende Codes, welche aufgrund ihrer Codesystem-Language-Einträge nicht im UML-Diagramm abgebildet werden, im Editor anzuzeigen, können diese per Drag-and-Drop in den UML-Editor geschoben werden. Dadurch werden die Codesystem-Language-Einträge des Codes auf „Concept“ gesetzt.

Zusätzlich wird in Abbildung 24 gezeigt, wie Codes, die im UML-Diagramm abgebildet werden, farbig hervorgehoben werden.

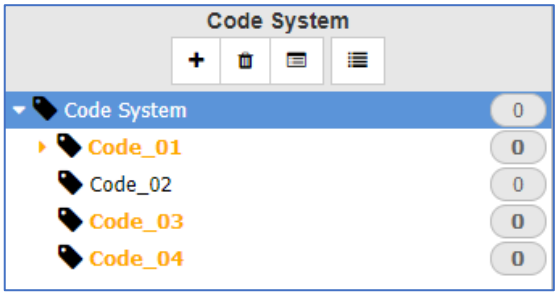


Abbildung 24: Hervorheben von Codes die im UML-Editor abgebildet werden

Kodierung von Beziehungen

In Kapitel 6.3 wurde erläutert, wie das Kodieren von Beziehungen realisiert wurde. Dadurch können Textstellen nicht nur mit Codes, sondern auch mit Beziehungen zwischen Codes, also z.B. Attributen oder Methoden, markiert werden.

Bearbeiten und Verändern des UML-Klassendiagramms

Hinzufügen/Entfernen von UML-Klassen (Codes)

Neue Klassen (Codes) können über einen Toolbar-Button des Editors hinzugefügt werden. Der Nutzer muss den Namen der Klasse angeben und es wird automatisch ein Code mit dem Codesystem-Language-Eintrag „Concept“ angelegt. Die einzelnen Schritte werden im Kapitel 7.1.2 noch deutlicher erläutert.

Klassen können über einen Hover-Button, der erscheint sobald die Klasse selektiert wird, entfernt werden.

Hinzufügen/Entfernen von Beziehungen zwischen den Klassen

Wenn eine Klasse selektiert wird, werden Hover-Buttons neben der Klasse angezeigt. Wählt man den Hover-Button zum Hinzufügen einer Kante, erscheinen drei neue Hover-Buttons. Bei diesen Buttons kann man wählen, ob man eine Generalisierung, Aggregation oder Assoziation hinzufügen möchte. Hat der Nutzer den Typ gewählt, muss er die Ziel-Klasse auswählen und die Kante wird hinzugefügt. Die einzelnen Schritte werden im Kapitel 7.1.2 noch deutlicher erläutert.

Wenn Kanten selektiert werden, können diese über einen Hover-Button gelöscht werden.

Hinzufügen/Entfernen von Attributen und Methoden

Attribute und Methoden können hinzugefügt, bzw. entfernt werden, sobald eine Klasse selektiert ist. Für die existierenden Attribute und Methoden erscheinen Buttons zum Löschen auf der rechten Seite (Siehe Abbildung 25).

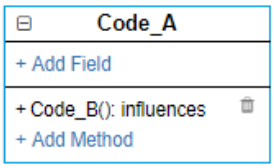


Abbildung 25: Hinzufügen und Entfernen von Attributen und Methoden

Funktionen zur Verbesserung des Arbeitsflusses

Layout

Der Editor bietet die Funktion ein neues Layout für das Klassendiagramm zu generieren. Dabei wird der mxGraph-Fast-Organic-Layout Algorithmus verwendet. Dieser Algorithmus ordnet alle Elemente abhängig von ihren Kanten neu aus und versucht dabei so wenig wie möglich Überlappungen von Kanten zu verursachen. Abbildung 26 zeigt auf der linken Seite ein Beispiel eines zufälligen ungeordneten Klassendiagramms. Auf der rechten Seite der Abbildung sieht man das Diagramm, nachdem der Algorithmus ausgeführt wurde.

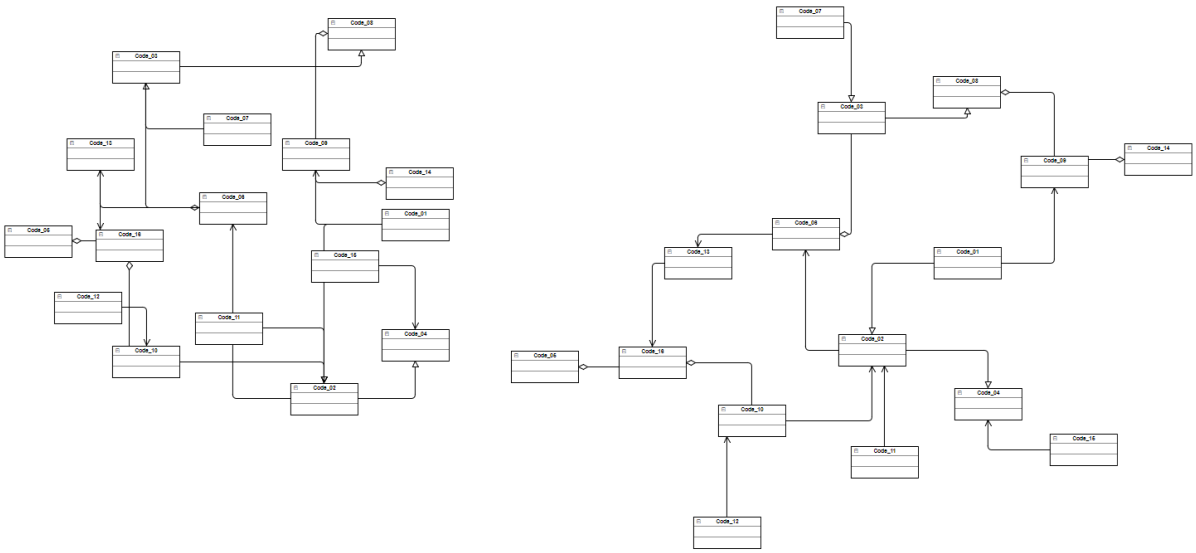


Abbildung 26: Vergleich des Layouting-Algorithmus; Auf der linken Seite sieht man ein zufälliges, ungeordnetes Layout; Auf der rechten Seite sieht man das Layout des Klassendiagramms nach der Ausführung des Algorithmus

Zoom

Um den Bildausschnitt im Editor zu verkleinern oder zu vergrößern, können die Toolbar-Buttons in Abbildung 27 verwendet werden. Auch durch Drehen des Mausrads kann die Ansicht verändert werden. Der Drop-Down-Button in der Abbildung zeigt immer den aktuellen Zoom-Faktor an.

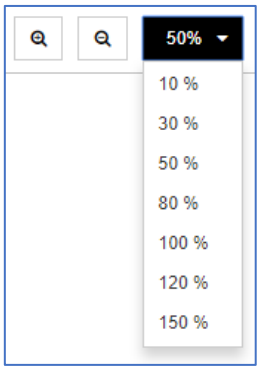


Abbildung 27: Zoom Toolbar-Buttons

Panning

Durch Halten der rechten Maustaste und bewegen der Maus kann die Ansicht horizontal und vertikal verschoben werden.

7.1.2. Nicht-funktionale Anforderungen

Funktionale Angemessenheit (functional appropriatness)

1) Abbildung 23 zeigt die Codesystem-Language-Komponente für Codes. Sobald ein Button, bzw. eine Codesystem-Language-Entität ausgewählt wird, wird die Änderung direkt in der Datenbank persistiert.
2) Wenn eine neue Klasse zum Editor hinzugefügt wird, muss der Nutzer einen Namen eingeben und ein Code mit der Codesystem-Language-Entität „Concept“ wird in der Datenbank gespeichert.
3) Um eine neue Klasse zum UML-Editor hinzuzufügen, muss der Nutzer zunächst auf den Button „New UML-Class“ in der Toolbar klicken. Dadurch öffnet sich ein Fenster, in dem der Nutzer den Namen der Klasse eingibt. Danach klickt der Nutzer auf „OK“ und die Klasse wird zum UML-Diagramm hinzugefügt (Siehe Abbildung 28). Für das Hinzufügen einer neuen Klasse werden also genau 3 Aktionen benötigt.

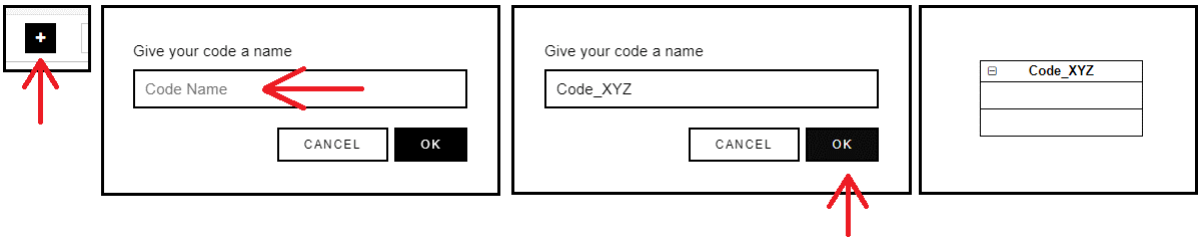


Abbildung 28: Teilschritte für das Hinzufügen einer Klasse

4) Damit der Nutzer eine neue Beziehung hinzufügen kann, muss er zunächst eine Start-Klasse auswählen. Durch die Selektion einer Klasse werden Hover-Buttons neben der Klasse angezeigt und der Nutzer wählt den Button zum Hinzufügen einer Kante „Add Edge“. Dadurch erscheinen drei neue Hover-Buttons, durch die der Nutzer auswählen kann, ob eine Generalisierung, Aggregation oder Assoziation hinzugefügt werden soll. Als letztes selektiert der Nutzer die Ziel-Klasse und die Kante wird hinzugefügt. Abbildung 29 zeigt die notwendigen Schritte nacheinander.

Abbildung 29: Tielschritte für das Hinzufügen neuer Kanten

Zeitverhalten (time behaviour)

Der wichtigste Faktor bei der Messung der Reaktionszeiten des Servers ist, dass der Server zum Zeitpunkt der Tests auf einem Google-Server in den USA gehostet wird. Weil es keine komplizierten Berechnungen oder Datenbank-Operationen geben, die für längere Antwort-Zeiten sorgen, ist hier lediglich die Distanz zum Server relevant. In Abbildung 30 sieht man die Verteilung der Anfragen über einen Zeitraum von 4 Wochen. Die meisten Anfragen benötigen weniger als 800ms. Es wurden aber auch Anfragen registriert, die zum Teil mehr als 10 Sekunden gebraucht haben. Das liegt daran, dass der Server während den Messungen neu gestartet wurde und die Endpoints bei den ersten Anfragen erst wieder hochfahren mussten.

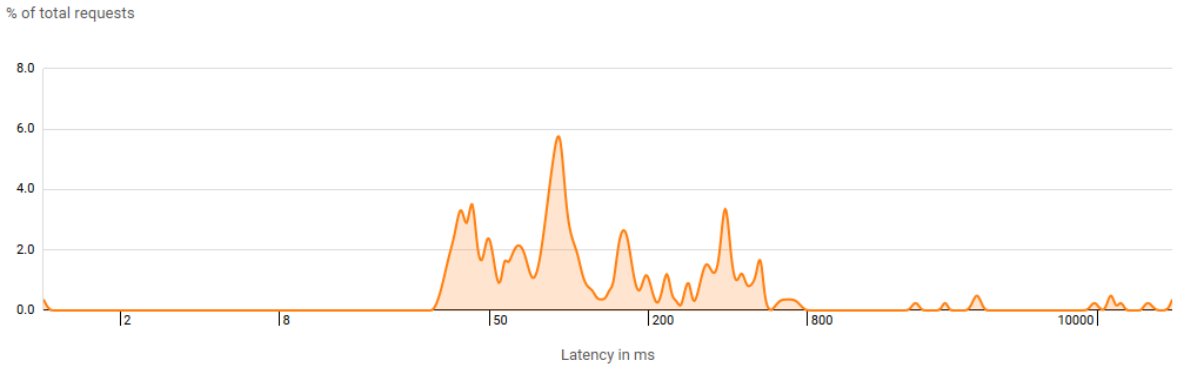


Abbildung 30: Messung der Reaktionszeiten des Servers

Kapazität/Skalierbarkeit (capacity)

(1) In der vereinfachten Ansicht wird bei den Klassen nur der Name angezeigt und die Attribute und Methoden werden ausgeblendet (Siehe Abbildung 32). Der Nutzer hat die Möglichkeit einzelne Klassen, oder alle Klassen des UML-Diagramms (Abbildung 31) ein- und auszuklappen.

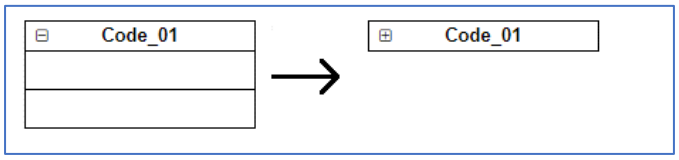


Abbildung 32: Vereinfachte Klassen-Ansicht

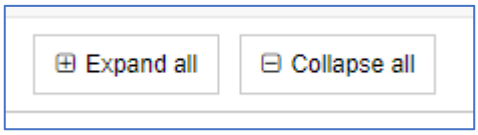


Abbildung 31: Toolbar-Buttons für das ein- und ausklappen aller Klassen

(2) Die Funktion, das Layout des Klassen-Diagramms neu zu generieren, wurde bereits im Kapitel 7.1.1 erläutert.

(3) Der Nutzer hat die Möglichkeit, sich das gesamte Klassen-Diagramm anzeigen zu lassen. Dafür wird die Ansicht verschoben und verkleinert, sodass alle Klassen zu sehen sind. Diese Funktion wird als Button in der Toolbar implementiert.

(4) Wenn Codes in der Codesystem-Komponente selektiert werden, dann wird auch die zugehörige Klasse (falls abgebildet) im UML-Editor selektiert. Wenn die Klasse nicht im sichtbaren Bereich ist, wird die Ansicht entsprechend verschoben um die Klasse anzuzeigen.
Wenn eine Klasse im UML-Editor selektiert wird, wird auch der zugehörige Code in der Codesystem-Komponente selektiert.

Erlernbarkeit (learnability)

Für den UML-Editor wurde versucht, für wichtige Aktionen mehrere Lösungswege anzubieten. Dabei wurde auch darauf geachtet, dass Menüs o.ä. durch die Redundanz nicht unübersichtlich werden. Wenn man z.B. einen Code im UML-Editor anzeigen möchte, der aufgrund des Mappings nicht abgebildet wird, kann man manuell die Codesystem-Language-Einträge ändern, oder den Code per Drag & Drop in den UML-Editor ziehen.
Alle Buttons wurden mit Tooltipps versehen, die für neue Anwender zusätzliche Informationen bereitstellen (Siehe Abbildung 33).

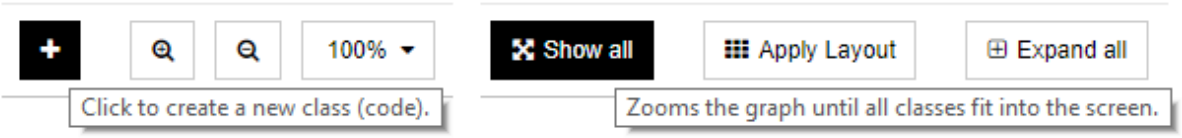


Abbildung 33: Beispiel-Tooltips

Fehler-Prävention (user error protection)

Wenn in QDAcity Codes oder Beziehungen zwischen Codes gelöscht werden, erscheint vorher ein Dialog-Fenster, welches bestätigt werden muss bevor ein Code gelöscht werden kann (Siehe Abbildung 34).

Abbildung 34: Bestätigungs-Dialog vor dem Löschen eines Codes

Vertraulichkeit (confidentiality)

Für die Authentifizierung der Nutzer wird in QDAcity das Protokoll O-Auth (Google Inc., 2017c) verwendet. Dadurch wird sichergestellt, dass nur registrierte und angemeldete Nutzer die Anwendung nutzen können. Um unberechtigte Zugriffe innerhalb der Anwendung zu verbieten, überprüfen die Endpoint-Methoden, ob der Nutzer berechtigt ist, die Operationen auszuführen. So dürfen z.B. nur die Teilnehmer eines Projektes das zugehörige Codesystem oder die zugehörigen Dokumente bearbeiten.

Alle API-Methoden wurden in Unit-Tests mit autorisierten und nicht autorisierten Nutzern auf die erwarteten Exceptions getestet.

Modularität (modularity)

Bei der Entwicklung des UML-Editors wurde darauf geachtet, den Editor möglichst wenig mit dem bestehenden System zu verschmelzen. D.h. es gibt nur eine Klasse UmlEditor die mit dem Rest des Systems kommuniziert. Alle anderen Klassen und Komponenten des UML-Editors werden hinter dieser Fassade verborgen (Siehe Kapitel 5.1).

Alle Zugriffe auf die externe Software-Bibliothek mxGraph werden von der Klasse GraphView gekapselt. Das bedeutet, wenn die Software zum Zeichnen des Graphen durch eine andere Komponente ausgetauscht werden soll, muss nur die Klasse GraphView angepasst werden.

7.2. Fazit

Im Rahmen dieser Masterarbeit wurde QDAcity um eine Software-Komponente erweitert, durch die es möglich ist, einzelne Elemente des konzeptuellen Modells mit Elementen aus qualitativen Daten zu verknüpfen. Das Codesystem wurde mit Elementen der Codesystem-Language erweitert und kann durch die definierten Mapping-Regeln in ein UML-konformes Diagramm abgebildet werden.

Auf Basis dieser Arbeit kann QDAcity noch um weitere Funktionen ergänzt werden. So wurde in der Arbeit „A Metamodel for Codesystems“ (Salow, 2016) auch vorgestellt, wie das Codesystem in Zustandsdiagrammen abgebildet werden kann. Das bedeutet, QDAcity könnte um einen weiteren Editor für Zustandsdiagramme erweitert werden. Zusätzlich kann auch der UML-Editor erweitert werden. So kann man z.B. für jedes Projekt in QDAcity eigene Codesystem-Language-Einträge bereitstellen, die dann vom Nutzer nach seinen Wünschen bearbeitet werden können. Zusätzlich können dann die Mapping-Regeln vom Nutzer selbst definiert werden.

Literaturverzeichnis

Apache (2015). Java Data Objects (JDO). https://db.apache.org/jdo/ [Zugriff am 30.09.2017]

Babel (2017). Babel JavaScript compiler. https://babeljs.io/ [Zugriff am 01.11.2017]

Client IO (2017a). Joint JS | Javascript Diagramming Library. https://github.com/clientIO/joint [Zugriff am 12.10.2017]

Client IO (2017b). Joint JS Demo-Projekte | Javascript Diagramming Library. https://resources.jointjs.com/demos [Zugriff am 12.10.2017]

ECMA International (2015). ECMAScript 2015 Language Specification. https://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf [Zugriff am 01.11.2017]

ECMA International (2016). ECMAScript 2016 Language Specification. https://www.ecma-international.org/ecma-262/7.0/ [Zugriff am 30.09.2017]

Facebook Inc. (2017). React.js | A JavaScript library for building user interfaces. https://reactjs.org/ [Zugriff am 30.09.2017]

Fedosejev, A. (2015). React.js essentials. A fast-paced guide to designing and building scalable and maintainable web apps with React.js. 18-19. Packt Publishing.

Formidable Labs (2017). Radium | Managing inline styles on React elements. https://github.com/FormidableLabs/radium [Zugriff am 04.11.2017]

Google Inc. (2017a). App Engine Overview | App Engine, Google Cloud Platform. https://cloud.google.com/appengine/?hl=de [Zugriff am 30.09.2017]

Google Inc. (2017b). Cloud Endpoints | App Engine, Google Cloud Platform. https://cloud.google.com/endpoints/docs/frameworks/legacy/v1/java/ [Zugriff am 30.09.2017]

Google Inc. (2017c). OAuth 2.0 | App Engine, Google Cloud Platform.
https://developers.google.com/identity/protocols/OAuth2 [Zugriff am 26.10.2017]

Google Inc. (2017d). Datastore | App Engine, Google Cloud Platform.
https://cloud.google.com/appengine/docs/standard/python/datastore/ [Zugriff am 30.09.2017]

Google Inc. (2017e). Memcache | App Engine, Google Cloud Platform.
https://cloud.google.com/appengine/docs/standard/java/memcache/ [Zugriff am 30.09.2017]

Google Inc. (2017f). Task Queues | App Engine, Google Cloud Platform.
https://cloud.google.com/appengine/docs/standard/python/taskqueue/ [Zugriff am 01.10.2017]

Google Inc. (2017g). Load Balancer | App Engine, Google Cloud Platform.
https://cloud.google.com/compute/docs/load-balancing/ [Zugriff am 30.09.2017]

ISO (2011). ISO/IEC 25010:2011. Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.

Jakl M. (2005). Representational State Transfer REST.
https://blog.interlinked.org/static/files/rest.pdf [Zugriff am 15.10.2017]

JGraph (2017a). mxGraph | A fully client-side JavaScript diagramming library.
https://github.com/jgraph/mxgraph [Zugriff am 12.10.2017]

JGraph (2017b). mxGraph | JavaScript Demo-Projekte.
https://jgraph.github.io/mxgraph/javascript/index.html [Zugriff am 12.10.2017]

Khan Academy (2017). Aphrodite | Colocating styles with JavaScript components.
https://github.com/Khan/aphrodite [Zugriff am 04.11.2017]

Llody, J.W. (1994). Practical Advantages of Declarative Programming

Provalis (2017). QDA-Miner | Qualitative Text Analysis Software.
https://provalisresearch.com/products/qualitative-data-analysis-software/ [Zugriff am 02.11.2017]

RQDA (2016). RQDA | Qualitative Data Analysis. http://rqda.r-forge.r-project.org/ [Zugriff am 02.11.2017]

Salow, S. (2016). A MetaModel for Codesystems. Masterarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg. https://osr.cs.fau.de/2016/10/28/final-thesis-a-metamodel-for-code-systems/ [Zugriff am 20.10.2017]

Styled Components (2017). Styled Components. https://github.com/styled-components/styled-components [Zugriff am 30.09.2017]

Verbi GmbH (2017). MaxQDA – The art of data analysis. http://www.maxqda.de/ [Zugriff am 02.11.2017]