HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Entwicklung eines grafischen Editors zur Metamodellierung sowie Validierung von Modell-Instanzen

Tobias Droth

Konstanz, 22.07.2015

BACHELORARBEIT

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

an der

Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Fakultät Informatik
Studiengang Angewandte Informatik

Thema: **Entwicklung eines grafischen Editors zur Metamodellierung sowie Validierung von Modell-Instanzen**

Bachelorkandidat: Tobias Droth
Schneckenburgstraße 48
78467 Konstanz

1. Prüfer: Prof. Dr. Marko Boger
2. Prüfer: M. Sc. Markus Gerhart

Ausgabedatum: 22.04.2015
Abgabedatum: 22.07.2015

Abstract

Thema:	Entwicklung eines grafischen Editors zur Metamodellierung sowie Validierung von Modell-Instanzen
Bachelorkandidat:	Tobias Droth
Institution:	HTWG Konstanz
Betreuer:	Prof. Dr. Marko Boger M. Sc. Markus Gerhart
Abgabedatum:	22.07.2015
Schlagworte:	Modell, Metamodell, Meta-Metamodell, Modellierung, Metamodellierung, Validierung, Modell-Instanzen, JSON, MoDiGen-Metamodell, Meta Object Facility, Ecore

Im Rahmen dieser Arbeit wurde ein grafischer Editor für die Modellierung von Metamodellen entwickelt. Der Editor wird in einem beliebigen Web-Browser ausgeführt und ist somit plattformunabhängig nutzbar. Er implementiert das an der HTWG Konstanz im Projekt *Progress in Graphical Modeling Frameworks* entwickelte MoDiGen-Metamodell, und erlaubt die Modellierung von Metamodellen, die zu diesem Meta-Metamodell konform sind.
Als Ausgabeformat nutzt der Editor eine JSON-Struktur, was die Datenhaltung mit Hilfe von JSON-basierten nicht-relationalen Datenbanken ermöglicht und die Implementierung des Editors in JavaScript erleichterte.
Zusätzlich wurde ein Werkzeug entwickelt, mit welchem die Instanzen des modellierten Metamodells, die Modelle, gegen das Metamodell geprüft werden können. Dieses Programm ist sowohl für die Clientseite im Web-Browser zur Prüfung eines Modells, als auch für die Serverseite zur Prüfung der Modelldaten vor der Persistierung relevant, weshalb die Validierung in JavaScript bzw. CoffeeScript entwickelt wurde. Im Web-Browser kann diese Implementierung direkt ausgeführt werden, serverseitig wurde die von der Mozilla Foundation in Java geschriebene JavaScript-Implementierung Rhino verwendet, um das JavaScript-Programm aufzurufen.
Der Theorieteil der Arbeit beschäftigt sich mit den Meta-Metamodellen, die die Grundlage des MoDiGen-Metamodells bilden, sowie ausführlich mit dem MoDiGen-Metamodell selbst. Im Praxisteil wird die Entwicklung und der Aufbau des Editors und des Validators erläutert.

Ehrenwörtliche Erklärung

Hiermit erkläre ich *Tobias Droth, geboren am 15.02.1992 in Singen*, dass ich

(1) meine Bachelorarbeit mit dem Titel

Entwicklung eines grafischen Editors zur Metamodellierung sowie Validierung von Modell-Instanzen

an der HTWG Konstanz unter Anleitung von Prof. Dr. Marko Boger selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 22.07.2015

(Unterschrift)

Inhalt

Abbildungen

Abkürzungen

CMOF	Complete Meta Object Facility
EMF	Eclipse Modeling Framework
EMOF	Essential Meta Object Facility
JSON	JavaScript Object Notation
MoDiGen	Model Diagram Generator
MOF	Meta Object Facility
OMG	Object Management Group
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 Einleitung

1.1 Vorarbeiten und Hintergründe

Die vorliegende Arbeit ist Teil des Projektes *Progress in Graphical Modeling Frameworks* an der Hochschule Konstanz Technik, Wirtschaft und Gestaltung (HTWG Konstanz). Im Rahmen dieses Projektes wird ein Toolkit entwickelt, welches die Spezifikation von domänenspezifischen Metamodellen aufgrund eines festgelegten Meta-Metamodells, sowie die Modellierung konkreter Sachverhalte auf Basis der spezifizierten Metamodelle erlaubt.

Grundlage des gesamten Projektes ist ein eigens entwickeltes Meta-Metamodell, das Model Diagram Generator (MoDiGen)-Metamodell, das in Anlehnung an den Ecore des Eclipse Modeling Frameworks (EMF) (siehe Eclipse Foundation, 2015) entwickelt wurde. Auf Grund dieses festgelegten Meta-Metamodells können branchen- und domänenspezifische Metamodelle und Modelle entwickelt werden, welche die zu modellierenden Sachverhalte anhand von Klassen, Referenzen und Vererbungsstrukturen mit Attributen und Abhängigkeiten beschreiben.

Metamodelle und Modelle werden bei MoDiGen in grafischen Editoren modelliert, die für den Anwender auch ohne tiefgreifende spezifische Kenntnisse über die Metamodellierung zu bedienen sind.

Der Vorgang der Modellierung bewegt sich auf vier Metamodellierungsebenen:

M0 Das reale Objekt, der reale Sachverhalt.

M1 Das Modell, welches den realen Sachverhalt abstrahiert und eine Instanz des Metamodells darstellt.

M2 Das Metamodell, welches das Modell abstrahiert und eine Instanz des Meta-Metamodells darstellt.

M3 Das Meta-Metamodell, welches sehr abstrakt die Grundlage aller Metamodelle darstellt.

Wie in Abbildung 1.1 zu sehen ist, beschreiben sich die Ebenen gegenseitig formal, bzw. abstrahieren einander. Das Meta-Metamodell ist reflexiv definiert, es beschreibt sich selbst.

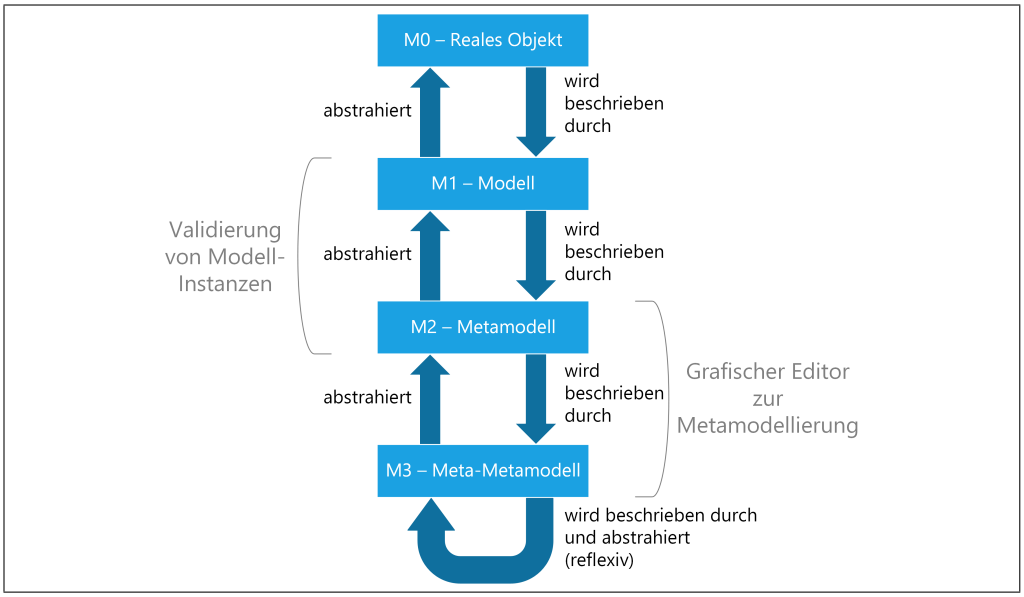

Abbildung 1.1: Übersicht über die Ebenen der Metamodellierung, eigene Darstellung, Stand: 15.06.2015

Serialisiert werden die Modelle bei MoDiGen in das JavaScript Object Notation (JSON)-Format, was gegenüber gängigeren Formaten wie Extensible Markup Language (XML) einige Vorteile besitzt. Sie besitzt wenig Overhead, ist auch für den menschlichen Betrachter sehr einfach zu erfassen, und mithilfe von nicht-relationalen Datenbanken wie bspw. CouchDB (siehe Apache Software Foundation, 2015) oder MongoDB (siehe MongoDB, 2015) ist die Datenhaltung einfach und gut horizontal skalierbar.

1.2 Ziel der Arbeit

Ziel der Arbeit ist die Entwicklung eines grafischen Editors zur Modellierung von Metamodellen. Außerdem soll ein Tool entwickelt werden, mit welchem die Konformität von Modellen gegenüber des ihnen zu Grunde liegenden Metamodells geprüft werden kann. Zu besseren Übersicht ist in Abbildung 1.1 eingezeichnet, auf welchen Ebenen sich die beiden Bestandteile dieser Arbeit bewegen: Der grafische Editor zur Metamodellierung nutzt das Meta-Metamodell als Grundlage und erlaubt die Entwicklung von Metamodellen, das Tool zur Validierung von Modell-Instanzen prüft, ob Modelle zu einem vorgegebenen Metamodell passen.

Es wäre möglich, Metamodelle mit einem Text-Editor direkt im JSON-Format zu spezifizieren, allerdings ist diese Vorgehensweise sehr fehleranfällig und nicht praktikabel. Einfacher, schneller und sicherer lassen sich Metamodelle mit Hilfe eines grafischen Editors modellieren. Dieser grafische Editor soll im Rahmen

dieser Arbeit entwickelt werden. Er soll in verschiedenen Web-Browsern lauffähig sein, was ihn plattformunabhängig und ohne Installation nutzbar macht. Die Modellierung soll für den Benutzer einfach und ohne lange Einarbeitungszeit durchführbar sein. Wichtig hierfür ist auch eine optisch ansprechende und leicht zu bedienende grafische Benutzeroberfläche. Für den Editor soll die in JavaScript implementierte Bibliothek für die Erstellung von grafischen Modell– und Diagramm-Editoren JointJS (siehe client IO, 2015a) mit deren Erweiterung Rappid (siehe client IO, 2015b) genutzt werden. Die Serialisierung und der Export des Metamodells im JSON-Format soll mit wenigen Mausklicks durchgeführt werden können.

Des Weiteren soll ein Tool entwickelt werden, mit welchem die Validität eines Modells gegenüber des ihm zu Grunde liegenden Metamodells geprüft werden kann. Die Schwierigkeit hierbei liegt unter anderem darin, dass diese Prüfung sowohl für den Benutzer im Web-Browser, als auch auf der Serverseite vor der Persistierung in einer Datenbank durchgeführt werden können soll. Hierfür soll deshalb eine Technik gefunden werden, die die Validierungslogik von beiden Seiten aus zugreifbar macht.

2 Theorie

In den folgenden Abschnitten wird die Theorie hinter der Metamodellierung erläutert. Hierfür werden erst die Begriffe Modell, Metamodell und Meta-Metamodell geklärt. Im Anschluss werden die Ideen und der Aufbau von Meta Object Facility (MOF) und Ecore beschrieben. Diese bilden die Grundlage für das in Abschnitt 2.4 vorgestellte MoDiGen-Metamodell, worauf der in dieser Arbeit entwickelte grafische Editor aufbaut.

2.1 Modelle, Metamodelle und Meta-Metamodelle

In vielen Branchen gehört es bereits zum Standardvorgehen, dass Probleme und Projekte vor der praktischen Umsetzung modelliert werden. In Ingenieursdomänen ist es unverzichtbar Sachverhalte als Planung vor der Umsetzung möglichst genau zu modellieren, da Fehlplanungen häufig mit hohen Kosten verbunden sind. In der Softwareentwicklung wurden Modelle lange Zeit hauptsächlich zum schnellen Skizzieren von Ideen und Strukturen, oder zur Dokumentation von bereits bestehender Software verwendet. Mittlerweile geht der Trend allerdings immer mehr in Richtung der modellgetriebenen Softwareentwicklung, was die Modellierung zu einem wesentlichen Bestandteil des Entwicklungsvorganges macht.

Allgemeine Modellierungssprachen wie Unified Modeling Language (UML) versuchen, eine einheitliche Sprache zur domänenübergreifenden Modellierung verschiedenster Sachverhalte zu schaffen. Allerdings ist die Nutzung von UML als Metamodell, also Modell das die tatsächlichen domäneneigenen Modelle direkt abstrahiert, nicht immer sinnvoll. Um syntaktische und semantische Feinheiten der verschiedenen Anwendungsgebiete deutlicher hervorzuheben und zu beschreiben sind häufig nur einige festgelegte und für das Fachgebiet spezifische Elemente notwendig; UML ist hierfür zu allgemein. Beispielsweise benötigt ein Softwareentwickler andere Elemente zur Modellierung als ein Architekt oder ein Heizungsinstallateur. Die Verallgemeinerung auf eine Modellierungssprache kann somit das domänenspezifische Verständnis der Modelle stark einschränken.

Aus diesem Grund werden allgemeine Modellierungssprachen wie UML häufig nicht als direktes Metamodell verwendet, sondern als Beschreibungssprache für das domänenspezifische Metamodell, als Meta-Metamodell. Das domänenspe-

zifische Metamodell stellt somit genau die Elemente zur Verfügung, die in der spezifischen Domäne notwendig und verständlich sind.

Die Architektur der domänenspezifischen Modellierung bzw. Metamodellierung wird deshalb von Seidewitz (2003, S. 30) in vier Ebenen beschrieben:

M0 Was modelliert wird

M1 Die Modelle

M2 Die Metamodelle

M3 Das Meta-Metamodell

Ebene M1, das Modell, ist eine direkte Abbildung aller wichtigen Aspekte des realen Objektes aus Ebene M0. Das Modell ist eine Sammlung von Aussagen, welche auf das Objekt zutreffen. Sollte eine Aussage, die das Modell über dessen Instanz trifft, unwahr sein, hängt es von der Art des Modells ab wo der Fehler liegt:

- Wird das Modell als Spezifikation für ein reales Objekt angefertigt, dann existiert das Modell zuerst, und das Objekt wird nach dem Modell erstellt. Trifft nun eine Aussage des Modells nicht auf das Objekt zu, ist das Objekt fehlerhaft.

- Wird das Modell hingegen als Dokumentation des Objektes genutzt, so hat sich das Modell nach dem Objekt zu richten. Stimmen Objekt und Modell nicht überein, ist das Modell fehlerhaft.

Beide genannten Fälle kommen in der Praxis vor. Der erste Fall tritt bspw. in Ingenieursdomänen auf, wo es üblich ist, einen Sachverhalt erst zu modellieren bevor er umgesetzt wird. In der Informatik kommen beide Fälle regelmäßig vor. Oftmals wird ein bestehendes Softwareprojekt mit Hilfe von Modellen dokumentiert, in der modellgetriebenen Softwareentwicklung wird ein Modell vor der Programmierung zur Spezifizierung des Softwareprojektes erstellt.

Metamodellierungsebene M2, das Metamodell, abstrahiert das Modell. Kühne (2006) beschreibt die Anwendung der Vorsilbe *meta* folgendermaßen:

> „In summary, the prefix ‚meta' is used before some operation f in order to denote that it was applied twice. Instead of stating ‚f–f', as in ‚class–class' one states ‚meta-f', e.g., ‚meta–class'. For any further application of the operation, another ‚meta' prefix is added to yield ‚meta–meta-class', etc." (Kühne, 2006, S. 377)

Das „Modell-Modell", *Metamodell*, wird als Grundlage für viele weitere Modelle genutzt. Ein Modell ist die Instanz eines Metamodells. Das Metamodell

abstrahiert das Modell und beschreibt dieses formal. Es legt fest, welche Typen von Knoten und Verbindungen es im Modell gibt, welche Eigenschaften diese haben, und wie sie zusammenhängen. Zu beachten ist, dass sich ein Metamodell zwar zwei Abstraktionsebenen über dem realen Objekt, Ebene M0, befindet, trotz dieser Abstraktion allerdings im Allgemeinen noch stark domänenspezifisch ist.

Die vierte und abstrakteste Ebene, bei der Betrachtung der Vier-Ebenen-Architektur, ist die Ebene des Meta-Metamodells, Metamodellierungsebene M3. Sie bildet die domänenunabhängige Grundlage aller (Meta-)Modelle, und ist somit eine abstrakte Beschreibung von Elementen und Eigenschaften des Metamodells. Alle Aussagen des allgemeinen Meta-Metamodells müssen auf das Metamodell zutreffen. Wegen dieser Mächtigkeit ist es sehr wichtig, dass das Meta-Metamodell sehr gut durchdacht ist, und alle möglichen domänenspezifischen Modelle aufgrund dessen modelliert werden können.

Ein gutes Meta-Metamodell als Grundlage der Modellierung ist so wichtig, dass schon einige namhafte Unternehmen offene Meta-Metamodelle erstellt haben. Die wichtigsten unter ihnen sind MOF der Object Management Group (OMG), sowie Ecore der Eclipse Foundation.

Auch ein Meta-Metamodell benötigt wiederum eine formale Beschreibung. Steinberg et al. (2008, S. 122) erläutert, dass sich ein gutes Meta-Metamodell selber beschreibe. Diese Eigenschaft der Selbstbeschreibung nennt man *reflexiv*. Laut ihm gebe es noch zwei weitere Möglichkeiten zur Beschreibung von Meta-Metamodellen, welche beide jedoch weitaus weniger schön seien als die Selbstbeschreibung. Man könne das Meta-Metamodell mit einem Modell auf einer höheren Abstraktionsebene, dem Meta-Meta-Metamodell, beschreiben. Allerdings hätte man das Problem damit nicht gelöst, sondern nur um eine Ebene verschoben, unter Umständen käme man hiermit in unendliche Rekursionen. Eine weitere Möglichkeit sei es, das Meta-Metamodell nicht formal zu beschreiben, sondern einfach „durchzuwinken“, dies zeuge allerdings auch von einem unschönen und problembehafteten Meta-Metamodell.

2.2 Meta Object Facility

MOF, das von der OMG entwickelt wurde, beschreibt eine Architektur für Metadaten, wobei der Hauptaugenmerk auf der Definition eines Meta-Metamodells liegt. MOF befindet sich momentan in Version 2, was eine komplette Überarbeitung und Verbesserung der ersten Version MOF1 ist. Als Austauschformat nutzt es XML Metadata Interchange (XMI), was eine Erweiterung von XML zum Austausch von Metadaten ist.

Im Folgenden wird tiefer auf den Aufbau und Zweck von MOF eingegangen. Die

getroffenen Aussagen beziehen sich auf die offizielle MOF Core Specification, siehe Object Management Group (2014).

2.2.1 Ebenen bei MOF

Die Architektur von MOF wird häufig als „Four layered metamodel architecture“ (Object Management Group, 2014, S. 6) beschrieben, da in den meisten Anwendungsfällen vier Ebenen verwendet werden. Allerdings ist laut Spezifikation eine beliebige Anzahl an Ebenen gleich oder größer zwei erlaubt.

Es seien mindestens zwei Ebenen notwendig, damit zwischen Klasse und Objekt navigiert werden kann. Mit drei Ebenen könne bspw. ein relationales Datenbanksystem mit SysTable, Table und Row abgebildet werden. Der Aufbau mit vier Ebenen werde am häufigsten verwendet. UML bspw. nutze die vier Ebenen MOF, UML, User Model und User Object, wie sie auch in Abschnitt 2.1 beschrieben sind. In bestimmten Anwendungsfällen seien auch mehr als vier Ebenen sinnvoll (vgl. Object Management Group, 2014, S. 6f).

2.2.2 Aufbau des MOF-Modells

MOF nutzt viele Teile der UML Spezifikation, erweitert diese und leitet von ihr ab. Dies hat den Vorteil, dass nur eine Teilmenge von UML verstanden werden muss, um MOF-konforme Metamodelle erstellen zu können. Des Weiteren gibt es schon Methoden um die so erstellten Metamodelle eindeutig zu serialisieren, zum Beispiel mit XMI oder Java Metadata Interface (JMI). Außerdem gibt es bereits viele Programme zum Modellieren von UML, mit welchen allesamt auch MOF-konforme Metamodelle erstellt werden können.

MOF bietet keine fertige Implementierung eines Meta-Metamodells, sondern ist nur eine Implementierungsvorlage. Es definiert Klassen, Methoden und Attribute des Meta-Metamodells, was plattformunabhängig in verschiedenen Sprachen umgesetzt werden kann. Die Spezifikationen der Schnittstellen sind objektorientiert beschrieben. Zur Veranschaulichung der möglichen Umsetzung werden häufig konkrete Java-Klassen genannt, allerdings ist die Umsetzung in anderen Sprachen auch möglich. Es wird genau festgelegt, welche Klassen es geben soll, welche Methoden und Attribute diese haben sollen, wie die Methoden sich verhalten sollen und welche Exceptions in welchen Fällen geworfen werden sollen.

Eigentlich gibt es nicht ein großes MOF-Meta-Metamodell, sondern die beiden Unterteilungen Complete Meta Object Facility (CMOF) und Essential Meta Object Facility (EMOF). CMOF ist die sehr komplexe Grundlage für viele große Metamodelle der OMG. Aufgrund des Umfangs und der Komplexität

gelte CMOF allerdings als schwer zu implementieren und als überladen. Viele in CMOF beschriebene Konstrukte seien in der Praxis nicht notwendig und bringen nur eine unnötige Komplexität in die Implementierung, vgl. Steinberg et al. (2008, S. 56).
Deshalb wurde neben CMOF noch EMOF entwickelt, ein einfacheres und kleineres Modell. EMOF ist eine Teilmenge von CMOF, welche mit Vorgaben und Einschränkungen weiter verkleinert wurde. Es enthält nur einen Teil der Elemente und Eigenschaften des CMOF und genügt den Anforderungen von objektorientierten Sprachen und von XML. Die mit EMOF als Meta-Metamodell erstellten Metamodelle können nicht so komplex sein wie die mit CMOF erstellten, in einem Großteil der Anwendungsfälle reiche EMOF allerdings aus.

2.3 Ecore

Das Ecore Meta-Metamodell wurde von der Eclipse Foundation entwickelt und bildet die Grundlage des Eclipse Modeling Framework (EMF). Ecore und EMOF sind sehr stark verwandt und nahezu austauschbar. Sie sind sogar so weit äquivalent, dass sich mit EMF erstellte Modelle sowohl mit der Ecore– als auch mit der EMOF-Serialisierung abspeichern lassen. Während EMOF allerdings komplett sprachunabhängig existiert, bildet Ecore die Grundlage des EMF, und ist deshalb hauptsächlich auf Java-Anwendungen ausgelegt.

Das EMF ist „[...] a powerful framework and code generation facility for building Java applications based on simple model definitions“ (Steinberg et al., 2008, S. 17). Modelle haben hierbei drei Darstellungsweisen: als annotierte Java-Klassen und –Interfaces, als XML-Schema und als UML-Diagramm. Diese Darstellungen können ineinander überführt, und aus ihnen Code generiert werden. Außerdem können daraus Schemas für relationale Datenbanken, sowie Java-Klassen zum Zugriff auf diese generiert werden, was die Persistierung der Modelle erleichtert.

Das Ecore-Modell kann selber mit dem auf Ecore basierenden EMF modelliert werden, und ist somit sein eigenes reflexives Metamodell.

Im Folgenden werden die Ebenen und der Aufbau des Ecore näher erläutert.

2.3.1 Ebenen bei Ecore

Ecore ist das Metamodell, aufgrund dessen im EMF modelliert werden kann. Somit kann es als direktes Metamodell für die Modellierung von Sachverhalten verwendet werden. Wie MOF ist allerdings auch Ecore sehr allgemein formuliert, damit er in allen möglichen Anwendungsfällen eingesetzt werden kann. Deshalb wird er in den meisten Fällen als Meta-Metamodell betrachtet und bildet die Grundlage für die Metamodelle der verschiedenen Domänen. Die Anzahl der

Ebenen ist beim Ecore nicht vorgeschrieben. Es müssen mindestens zwei sein, der Ecore als Metamodell und darauf aufbauend das Modell, es können allerdings beliebig viele Meta-Ebenen eingeschoben werden. Häufig wird der Aufbau mit drei Modellierungsebenen und der Ebene des zu modellierenden Objektes verwendet, wie in Abschnitt 2.1 beschrieben.

2.3.2 Aufbau des Ecore-Modells

Der komplette Aufbau des Ecore Metamodells ist in Abbildung 2.2 zu sehen, eine starke Vereinfachung mit den grundlegenden Elementen in Abbildung 2.1.

Wie in Steinberg et al. (2008, S. 34) beschrieben, stellt `EClass` eine modellierte Klasse dar. Sie besteht aus einem Namen, null oder mehr Referenzen sowie null oder mehr Attributen. Um Vererbungshierarchien darzustellen, besitzt `EClass` im Feld `eSuperTypes` eine Liste von `EClass`-Objekten. Alle hier aufgelisteten Klassen bilden Superklassen der entsprechenden Klasse; die Klasse erbt von all diesen Klassen. `EReference` beschreibt eine Verbindung zwischen zwei Klassen. Sie hat einen Namen, sowie eine Boolean-Variable um zu signalisieren, ob die Referenz ein *containment* ist, also eine *has-a*-Beziehung. Eine Referenz ist bei Ecore immer innerhalb einer Klasse definiert, welche den Anfangspunkt der Referenz darstellt. Für den Endpunkt besitzt die Referenz einen `eReferenceType` vom Typ `EClass`. Somit stellt sie immer eine `1:1`-Verbindung, eine Verbindung zwischen genau zwei Klassen, dar. Eine Klasse hat eine beliebige Anzahl an `EAttributes`, die deren Attribute beschreiben. Attribute bestehen aus einem Namen und einem Datentyp `EDataType`, der ein primitiver Datentyp oder eine komplexe Klasse sein kann.

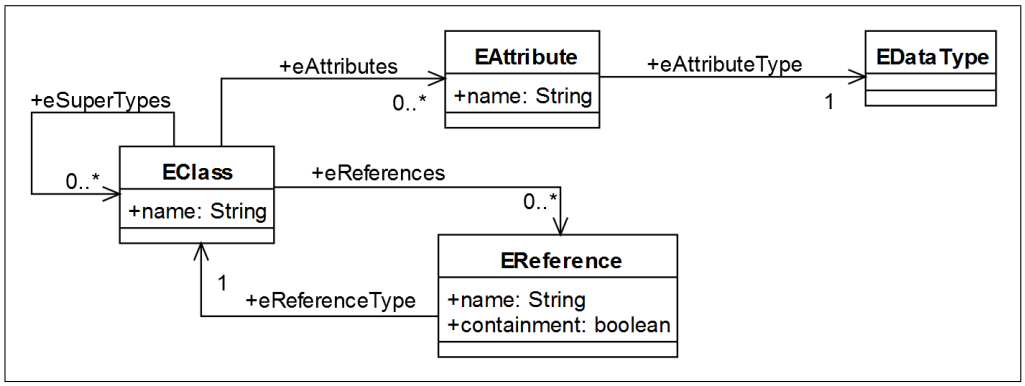

Abbildung 2.1: Vereinfachte Struktur des Ecore Metamodells, eigene Darstellung in Anlehnung an Steinberg et al. (2008, S. 34)

Wie in Abbildung 2.2 zu sehen ist, besteht das Ecore-Modell aus weitaus mehr Klassen als in Abbildung 2.1 dargestellt ist, und diese besitzen noch mehr Attribute und Methoden. Die hier dargestellte vereinfachte Teilmenge genügt

allerdings, um die Grundlagen des Ecore zu verstehen. Außerdem zeigt sie die Parallelen zum in Abschnitt 2.4 vorgestellten MoDiGen-Metamodell gut auf.

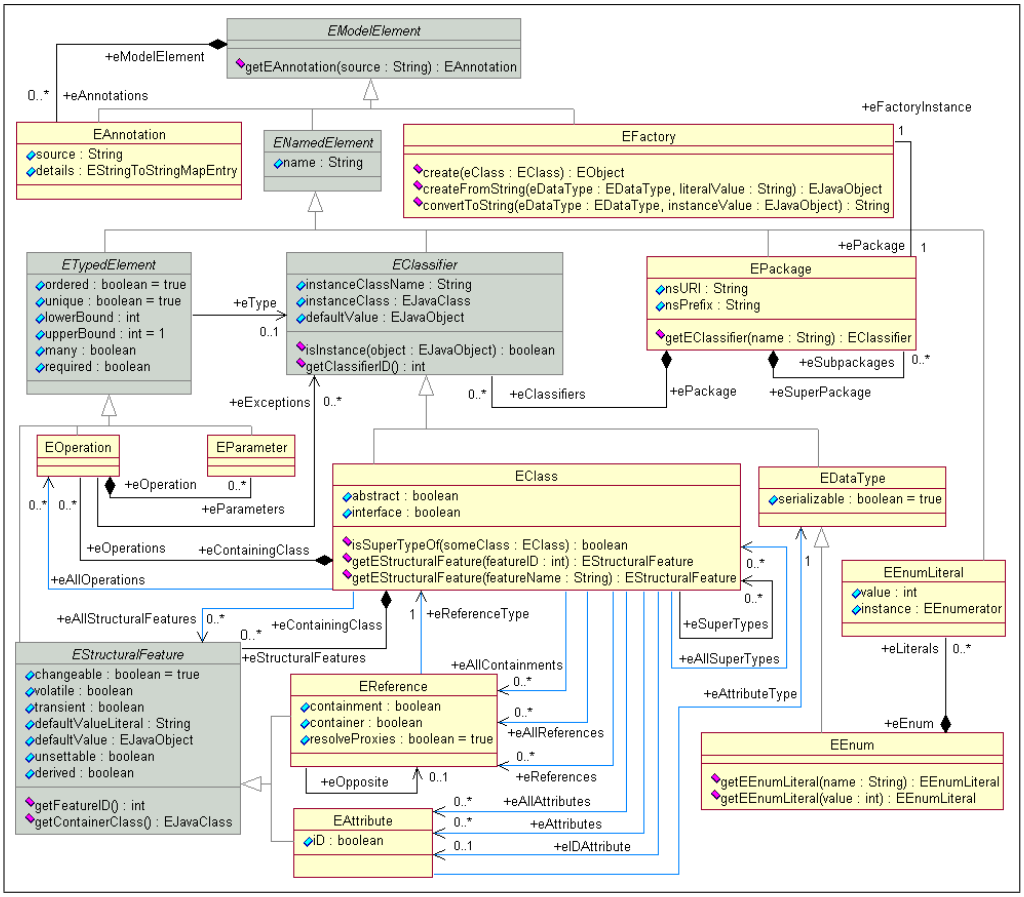

Abbildung 2.2: Vollständige Struktur des Ecore Metamodells,
Quelle: `http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/doc-files/EcoreRelations.gif`,
Abgerufen am 07.05.2015

Ein Modell in EMF kann auf drei Arten definiert werden: Als UML-Diagramm, anhand eines XML-Schemas und mit Hilfe von annotierten Java-Klassen. Diese drei Darstellungen enthalten dieselben Informationen und können aus jeweils einem der beiden anderen generiert werden. Die Serialisierung der Modelle erfolgt mit XMI. Der Vorteil des XML-Datenformates ist, dass eine XML-Struktur automatisiert gegen ein XML-Schema geprüft werden kann. Somit kann überprüft werden, ob ein Modell, in Form seiner XMI-Serialisierung, zum vorgegebenen Metamodell konform ist. Ferner kann daraus ein Schema für eine relationale Datenbank generiert werden, was die Persistierung und Datenhaltung erleichtert.

2.4 MoDiGen-Metamodell

Das MoDiGen-Metamodell, welches im Rahmen des Projektes *Progress in Graphical Modeling Frameworks* an der HTWG Konstanz entwickelt wurde, ist ein einfaches Meta-Metamodell, welches als Grundlage für domänenspezifische Metamodelle genutzt werden kann. Der grafische Editor, der im Rahmen dieser Arbeit programmiert wurde, nutzt das MoDiGen-Metamodell als Basis der Modellierung. Dieses Meta-Metamodell wurde in Anlehnung an den Ecore der Eclipse Foundation entwickelt, ist jedoch weitaus weniger komplex und implementiert einige neue Ansätze und Ideen.

2.4.1 Ebenen bei MoDiGen

Die Modellierung mit dem MoDiGen-Metamodell ist auf vier Ebenen festgelegt. Drei Modellierungsebenen – das Meta-Metamodell (MoDiGen-Metamodell), das domänenspezifische Metamodell und das daraus resultierende Modell –, sowie die Ebene des tatsächlichen Sachverhaltes, welcher modelliert werden soll. Diese Ebenen sind klar voneinander getrennt und erfüllen ihre eigenen Aufgaben:

1. Das MoDiGen-Metamodell ist das sehr allgemein gehaltene Meta-Metamodell und genügt als Grundlage für die Metamodelle. Es beschreibt die grundlegenden Bausteine der Metamodelle: Klassen, Referenzen, Attribute etc.

2. Das Metamodell wird mit Hilfe des im Rahmen dieser Arbeit entwickelten grafischen Metamodell-Editors auf Grundlage des Meta-Metamodells modelliert. Es nutzt die Bausteine, die im Meta-Metamodell festgelegt sind, und bietet die domänenspezifische Grundlage für konkrete Modelle.

3. Das Modell stellt die direkte Ableitung des zu modellierenden Objektes dar. Es nutzt die im Metamodell definierten Elemente und modelliert somit einen konkreten Sachverhalt.

4. Das Objekt, die Instanz des Modells, ist der eigentliche komplexe Sachverhalt, der vereinfacht durch das Modell dargestellt wird.

Im Gegensatz zu MOF, welche aus zwei oder mehr Ebenen bestehen kann, ist die Anzahl der Ebenen bei MoDiGen auf vier festgelegt und unveränderbar. Dies stellt jedoch kaum eine Einschränkung dar, da alle relevanten Anwendungsfälle mit diesem Aufbau abgedeckt sind. Auch bei MOF und Ecore ist der Aufbau aus vier Ebenen der geläufigste; mehr oder weniger Ebenen sind als Sonderfälle zu betrachten.

2.4.2 Aufbau des MoDiGen-Metamodells

Im Folgenden wird der Aufbau des MoDiGen-Metamodells näher erläutert. Grafisch kann der Aufbau des Meta-Metamodells als eine UML-ähnliche Struktur dargestellt werden, wie in Abbildung 2.3 ersichtlich ist.

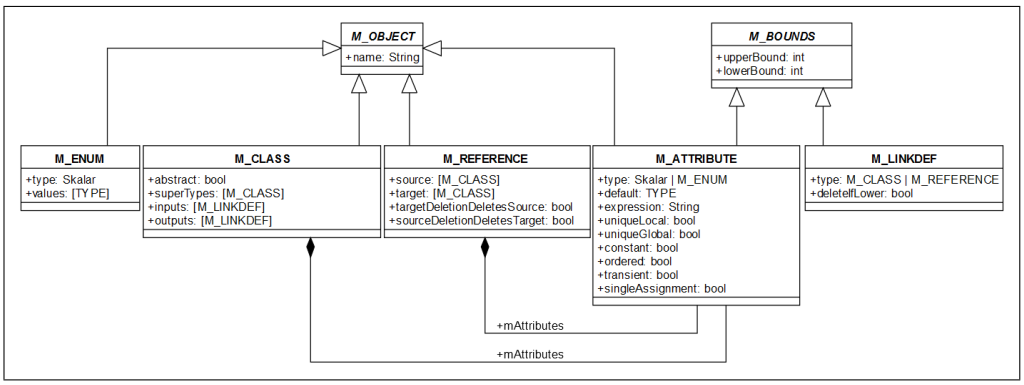

Abbildung 2.3: Vollständige Struktur des MoDiGen-Metamodells, eigene Darstellung

Die Darstellung der Verbindungen im Meta-Metamodell ist wie in einem UML-Klassendiagramm zu verstehen. Die Verbindungen mit dem weißen nicht-ausgefüllten Pfeil stellen Vererbungen dar. Die Verbindungen mit der schwarzen Raute zeigen Kompositionen: die Klasse am Verbindungsende ohne Raute kann eigenständig, also ohne die Klasse am Ende mit Raute, nicht existieren.

Im Folgenden werden die einzelnen Klassen des MoDiGen-Metamodells und deren Attribute näher erläutert.

`M_OBJECT` ist die abstrakte Wurzelklasse aller benannten Klassen, und enthält deshalb das Attribut `name` vom Typ String.

`M_CLASS` stellt eine Klasse dar. Sie erbt von `M_OBJECT` und enthält die folgenden eigenen Attribute:

- `abstract` ist ein Boolean-Wert, der aussagt, ob die Klasse abstrakt ist oder nicht. Abstrakte Klassen können nicht direkt instanziiert werden, sondern stellen nur die Verallgemeinerung für eine oder mehrere Subklassen dar, die diese abstrakten Klassen erweitern.

- `superTypes` ist eine Liste von Referenzen zu `M_CLASS`-Objekten. Es ermöglicht die Modellierung von Vererbungsstrukturen, und erlaubt, durch die Implementierung als Liste, explizit auch Mehrfachvererbungen.

- `inputs` ist eine Liste von Referenzen zu `M_LINKDEF`-Objekten. Es beschreibt alle Referenzen, die für diese Klasse als eingehende Referenz erlaubt sind.

- `outputs` ist eine Liste von Referenzen zu `M_LINKDEF`-Objekten, und beschreibt die Referenzen, die als ausgehende Referenzen der Klasse erlaubt sind.

Des Weiteren hat eine `M_CLASS` eine beliebige Anzahl an `M_ATTRIBUTE`-Objekten, die die möglichen Attribute und deren Eigenschaften darstellen.

`M_REFERENCE` beschreibt Referenzen. Referenzen sind Verbindungen zwischen Klassen, mit welchen deren Beziehungen modelliert werden können. Da es von `M_OBJECT` erbt, hat es einen Namen und außerdem die folgenden Attribute:

- `source` ist eine Liste von Referenzen zu `M_LINKDEF`-Objekten, die die möglichen Quell-Klassen der Referenz beschreiben.

- `target` ist eine Liste von Referenzen zu `M_LINKDEF`-Objekten, die die Ziel-Klassen der Referenz beschreibt. Dadurch, dass `source` und `target` keine einfachen Werte sondern Listen sind, sind sowohl `1:1`–, als auch `1:n`–, `n:1`– und `n:m`-Beziehungen explizit erlaubt.

- `targetDeletionDeletesSource` ist ein Boolean-Wert, der beschreibt, ob beim Löschen der Ziel-Klasse die Quell-Klasse mit gelöscht werden soll.

- `sourceDeletionDeletesTarget` ist ein Boolean-Wert, der beschreibt, ob beim Löschen der Quell-Klasse die Ziel-Klasse gelöscht werden soll. Mit den beiden Werten `targetDeletionDeletesSource` und `sourceDeletionDeletesTarget` kann die Stärke der Verbindung modelliert werden. So kann bspw. mit dem Setzen von `sourceDeletionDeletesTarget` auf *true* eine *is-a*– bzw. *has-a*-Beziehung modelliert werden, was einem `containment` im Ecore entspricht.

Außerdem haben Referenzen eine beliebige Anzahl von `M_ATTRIBUTE`s, womit Attribute dieser Referenz modelliert werden können.

`M_ENUM` kommt vom Wort *enumeration*, was so viel wie *Aufzählung* bedeutet. Mit `M_ENUM`s können also Aufzählungen definiert werden, welche danach bei allen Attributen als Datentyp verwendet werden können. Es erbt von `M_OBJECT` damit es einen Namen hat, und besitzt außerdem die folgenden Attribute:

- `type` beschreibt den Datentyp, den alle Werte des Enums besitzen. Der Wert ist mit Skalar angegeben, und kann somit ein primitiver Datentyp, String, Integer oder Float, sein.

- `values` ist eine Liste von Werten, die das Enum annehmen kann. Der Datentyp dieser Werte entspricht dem in `type` angegebenen Datentyp.

Alle angelegten `M_ENUM`s können beim `type`-Attribut der `M_ATTRIBUTES` als Datentyp ausgewählt werden. Enums finden auch in vielen Programmiersprachen

Anwendung, und dienen dort auch zur Auswahl von einzelnen konstanten Werten aus einer Aufzählung. Im Ecore ist der Enum-Datentyp als Abbildung von String-Literalen auf jeweils einen Integer-Wert implementiert.

`M_BOUNDS` ist eine abstrakte Klasse. Sie beschreibt upper und lower Bounds, also obere und untere Grenzen. Verwendet werden die `M_BOUNDS` bspw. für die Angabe von Multiplizitäten von Attributen. Somit hat `M_BOUNDS` die folgenden Attribute:

- `upperBound` ist ein Integer-Wert, der eine obere Grenze beschreibt. Hat der `upperBound` den Wert –1, so wird er als nicht existent bzw. als unendlich angesehen.

- `lowerBound` ist ein Integer-Wert, der eine untere Grenze beschreibt.

`M_ATTRIBUTE` beschreibt Attribute, von `M_CLASS`– oder `M_REFERENCE`-Typen. Es erbt sowohl von `M_OBJECT` als auch von `M_BOUNDS`, hat somit einen Namen und eine obere und eine untere Grenze. Die obere und untere Grenze beschreiben die Anzahl, wie viele Werte eine Klasse oder Referenz von diesem Attribut haben darf. Eine Person hat bspw. genau ein Geburtsdatum, also müssen sowohl `upper`– als auch `lowerBound` beim Attribut mit dem Namen *Geburtsdatum* auf 1 stehen. Dieselbe Person kann allerdings eine oder mehrere Steuernummern besitzen, weshalb beim Attribut *Steuernummer* die untere Grenze auf 1, die obere auf –1 gesetzt werden muss. Außerdem haben `M_ATTRIBUTE`s die folgenden Felder:

- `type` beschreibt den Datentyp der Attribut-Werte. Hierbei gibt es die Datentypen Skalar, also String, Integer, Float oder Boolean, sowie `M_ENUM`, also eines der in `M_ENUM` festgelegten Enum-Datentypen.

- `default` beschreibt einen Standard-Wert, mit dem das Attribut belegt ist, wenn kein eigener Wert gewählt wird. Der Datentyp dieses Standard-Wertes muss dem in `type` definierten Datentyp entsprechen.

- `expression` kann einen arithmetischen Ausdruck festlegen, nach welchem der Wert des Attributes berechnet wird. Dies kann dann sinnvoll sein, wenn der Wert des Attributes von anderen Attributen abhängt. Sie ist vom Typ String.

- `uniqueLocal` ist ein Boolean-Wert. Es sagt aus, dass in der Liste der Werte innerhalb des Attributes kein Wert mehrmals vorkommen darf. Alle Attributwerte müssen somit lokal, innerhalb der Klasse oder Referenz, eindeutig sein.

- `uniqueGlobal` ist ein Boolean-Wert, der aussagt, dass der Wert des Attributes modellweit eindeutig sein muss. Keine Klasse oder Referenz, die dieses Attribut nutzt, darf somit einen Wert besitzen, der in einer anderen Klasse oder Referenz bereits vorkommt.

- `constant` sagt aus, dass der Wert konstant ist. Er ist auf den `default`-Wert festgelegt und darf nicht verändert werden. Der `default`-Wert muss dafür gesetzt sein.

- `ordered` sagt aus, dass die Werte des Attributes einer Sortierung unterliegen. Die Reihenfolge der Werte ist wichtig und muss deshalb beachtet und beibehalten werden.

- `transient`, flüchtig, ist ein Hinweis zur Persistierung des Attributes. Ist dieser Wert *true*, so darf das Attribut nicht in der Datenbank abgespeichert werden. Dies kann dann sinnvoll sein, wenn der Wert anhand einer `expression` errechnet wird.

- `singleAssignment` ist ein Boolen-Wert. Er ist ähnlich dem Wert `constant`, nur dass der Attribut-Wert hierbei nicht auf den `default`-Wert festgesetzt ist, sondern genau einmal zugewiesen werden darf. Nach dieser Zuweisung ist der Wert konstant und darf nicht mehr verändert werden.

`M_LINKDEF` spezifiziert `input`– und `output`-Referenzen von Klassen, bzw. `source`– und `target`-Klassen von Referenzen genauer. Die Felder `input`, `output`, `source` und `target` innerhalb der Klassen bzw. Referenzen referenzieren nicht direkt die jeweilige Referenz oder Klasse, sondern `M_LINKDEF`-Typen. Es erbt von `M_BOUNDS`, hat somit upper und lower Bounds. Die Bounds geben an, wie viele Referenzen oder Klassen dieses Typs die Klasse bzw. Referenz als `input`, `output`, `source` bzw. `target` besitzen darf. Hat bspw. *KlasseA* als `output` einen `M_LINKDEF` vom Typ *ReferenzA*, bei welchem der `lowerBound` auf 1 gesetzt ist, muss *KlasseA* mindestens eine ausgehende Referenz vom Typ *ReferenzA* besitzen.

Außer den Bounds hat `M_LINKDEF` die folgenden Attribute:

- `type` beschreibt die Klasse oder Referenz, auf welche sich das `M_LINKDEF`-Objekt bezieht. Es ist vom Typ `M_CLASS` oder `M_REFERENCE`.

- `deleteIfLower` gibt an, ob das `M_LINKDEF` und die dazugehörige Klasse oder Referenz gelöscht werden soll, wenn die Anzahl der verbundenen Klassen oder Referenzen bei der Modellierung auf den festgelegten `lowerBound` fällt. Somit können bspw. *is-part-of*-Beziehungen modelliert werden.

2.4.3 Serialisierung und Skalierbarkeit

Zur weiteren Verwendung von Metamodell und Modell, sowie zur Persistierung der Modelle, müssen diese serialisiert werden. Ecore nutzt hierfür das XML-Format, bzw. XMI. Wie allerdings in Kolovos et al. (2013, S. 6) beschrieben wird, ist dies kein effizientes Format zur Darstellung von Modellen. Es ist nicht möglich, Modelle nur teilweise in den Speicher zu laden, was bei sehr

großen Modellen erforderlich ist. Außerdem haben XML-basierte Formate viel Overhead, und benötigen somit viel mehr Speicherplatz als für die eigentlichen Informationen notwendig wäre.

Aus diesem Grund nutzt MoDiGen kein XML-basiertes Format, sondern JSON. Wie in Severance (2012, S. 7) beschrieben, gab es, und gibt es immer noch, eine Debatte, welches das „bessere“ Format zur Serialisierung von Daten und zum Datenaustausch sei: XML oder JSON. XML wird auch heute noch als Quasi-Standard in vielen Business-Lösungen verwendet. Der Vorteil von XML ist, dass ein Schema definiert werden kann, gegen das die XML-Dateien validiert werden können. In vielen Fällen überwiegen allerdings die Vorteile von JSON, weshalb es heute, insbesondere im Web-Bereich, immer mehr Verwendung findet. Die Datenmenge für die Darstellung der gleichen Informationen ist im Allgemeinen bei JSON kleiner als bei XML. Zudem kann JavaScript nativ mit dem JSON-Format umgehen, und für die meisten gängigen Programmiersprachen gibt es gute und einfache Bibliotheken für den Umgang mit JSON-Daten. Des Weiteren ist die Persistierung von JSON-Daten trivial. Einige No-SQL-Datenbanken wie CouchDB oder MongoDB arbeiten mit JSON bzw. dessen binärem Pendant Binary JSON (BSON), weshalb man die Daten ohne Vorarbeit in die Datenbank geben kann. Diese erlaubt einen einfachen und schnellen Zugriff auf die Daten. Für die Persistierung von XML kann aus dessen Schema eine Datenbankstruktur für relationale Datenbanken erstellt werden, die erlaubt, die XML-Daten zu speichern, was allerdings mit mehr Aufwand verbunden ist.
Ein weiterer Vorteil der nicht-relationalen Datenbanken ist die einfache horizontale Skalierung. Wo eine horizontale Skalierung mit SQL-Datenbanken wie bspw. MySQL nur mit großem Aufwand erreicht werden kann, unterstützen viele nicht-relationale Datenbanken das Hinzufügen von zusätzlichen Datenbankservern sehr gut und ohne großen Konfigurationsaufwand.
Mit JSON ist es ohne Weiteres möglich, nur Teile der Modell-Daten zu laden und verändert abzuspeichern, was den Umgang mit sehr großen Modellen erleichtert. Wo die Daten bei XML als Ganzes im Hauptspeicher gehalten werden müssen, können die Daten mit JSON objektweise aus der Datenbank geholt werden. Bei großen Modellen im XML-Format kann es somit passieren, dass der Hauptspeicher vollläuft, was die Arbeit mit dem Modell unmöglich macht.

Da JSON im Gegensatz zu XML keine Unterstützung von Schemas hat, mit welchem die Datenstruktur formal beschrieben werden kann, ist es schwierig, den Aufbau der serialisierten JSON-Strukturen von Metamodell und Modell allgemeingültig zu formalisieren. Deshalb wird der Aufbau der serialisierten JSON-Strukturen bei MoDiGen im Folgenden beispielhaft beschrieben. Wegen der Einfachheit von JSON, Objekte als Sammlung von Schlüssel-Wert-Paaren, primitive Datentypen und Listen, ist die beispielhafte Beschreibung allerdings sehr leicht auf weitere komplexe Modelle übertragbar.

Die Serialisierung der Metamodelle, welche das MoDiGen-Metamodell als Grundlage nutzen, bestehen aus einem großen JSON-Objekt. Dieses enthält eine beliebige Anzahl von Objekten, welche als Schlüssel den vom Benutzer definierten

Namen besitzen. Diese Objekte müssen von einem der drei Typen `M_CLASS`, `M_REFERENCE` oder `M_ENUM` sein, was in deren `mType`-Feld als *mClass*, *mRef* oder *mEnum* festgehalten ist. Weiter enthalten diese Objekte genau die Felder, die im Meta-Metamodell direkt oder durch Vererbung festgelegt sind, siehe Abbildung 2.3.

Die grafische Darstellung eines vereinfachten Metamodells für einen Familienbaum, welches auf Grund des MoDiGen-Metamodells entwickelt wurde, ist in Abbildung 2.4 zu sehen. Dort gibt es den allgemeinen Typ `Person`, welcher die Attribute `Vorname`, `Nachname`, `Steuernummer` sowie `Geburtsdatum` enthält. Sowohl die Klasse `Mann` als auch `Frau` erben von der abstrakten Klasse `Person`. Ein Mann kann Vater einer Person sein, eine Frau Mutter einer Person, außerdem kann ein Mann Ehemann einer Frau sein und eine Frau Ehefrau eines Mannes.

Die Pfeile mit der nicht ausgefüllten weißen Spitze stellen, wie auch bei UML-Modellen, Vererbungen dar. Die Pfeile mit der einfachen Pfeilspitze sind Kompositionen, wobei bei MoDiGen auch Aggregationen und Assoziationen definiert sind, welche sich jeweils an den Werten von `targetDeletionDeletesSource` und `sourceDeletionDeletesTarget` unterscheiden.

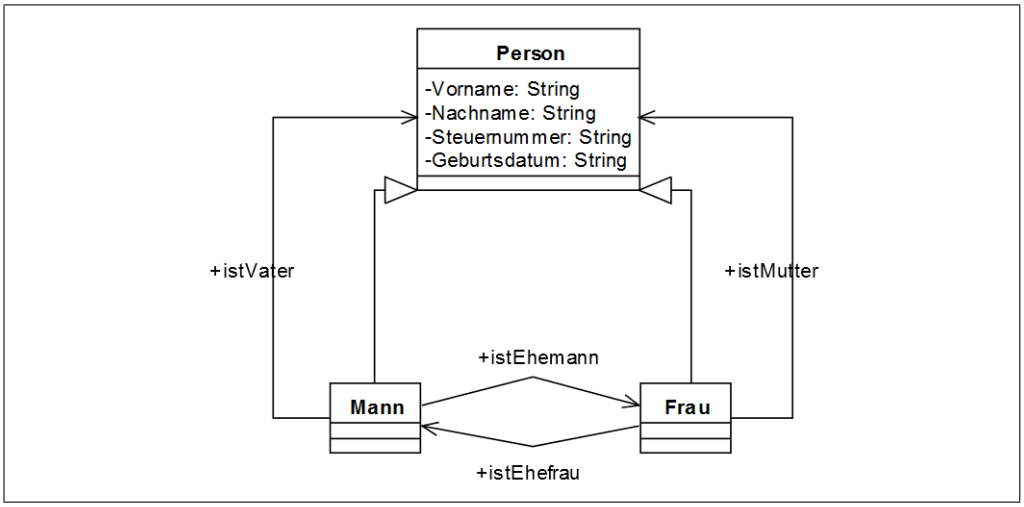

Abbildung 2.4: Grafische Darstellung des Metamodells für einen Familienbaum, eigene Darstellung, Stand: 22.05.2015

Die serialisierte JSON-Darstellung der `Mann`-Klasse des Metamodells ist in Abbildung 2.5 zu sehen. Er hat keine eigenen Attribute und erbt von `Person`, weshalb das `mAttributes`-Objekt leer ist. Wie man in der JSON-Darstellung auch sieht, haben die Felder `inputs` und `outputs` keine direkten Referenzen auf ein `M_REFERENCE`-Objekt, sondern auf `M_LINKDEF`. Darin ist nicht nur der Typ der Referenz festgelegt, sondern wie im Meta-Metamodell definiert auch Informationen über die oberen und unteren Grenzen, sowie über das Löschverhalten.

```
"Mann": {
    "mType": "mClass",
    "name": "Mann",
    "abstract": false,
    "superTypes": [
      "Person"
    ],
    "mAttributes": {},
    "inputs": [
      {
        "type": "istEhefrau",
        "upperBound": 1,
        "lowerBound": 0,
        "deleteIfLower": false
      }
    ],
    "outputs": [
      {
        "type": "istEhemann",
        "upperBound": 1,
        "lowerBound": 0,
        "deleteIfLower": false
      },
      {
        "type": "istVater",
        "upperBound": -1,
        "lowerBound": 0,
        "deleteIfLower": false
      }
    ]
  }
```

Abbildung 2.5: JSON-Darstellung eines Objektes im Metamodell bei MoDiGen, eigene Darstellung, Stand: 22.05.2015

Die Darstellung von Referenzen und Enums in der JSON-Serialisierung des Metamodells ist entsprechend; sie enthält die Felder, die im Meta-Metamodell festgelegt sind. Da der vom Benutzer definierte Name eines Objektes als Schlüssel im Metamodell-Objekt fungiert, muss dieser innerhalb des Modells eindeutig sein.

Nachdem das Metamodell definiert ist, kann daraus eine beliebige Anzahl von Modellen erstellt werden. Diese haben, wie auch das Metamodell, zwei Darstellungen: eine grafische Darstellung, die vom Benutzer in der grafischen Oberfläche modelliert wird, sowie eine Darstellung im JSON-Format, welche aus

dem grafischen Modell extrahiert wird. Im Modell dürfen nur die im Metamodell festgelegten Klassen und Referenzen vorkommen, diese dürfen nur die definierten Attribute in den festgelegten Bounds enthalten.

In Abbildung 2.6 ist ein sehr einfaches Modell zu sehen, das zu dem in Abbildung 2.4 definierten Metamodell konform ist. Hierbei ist die familiäre Beziehung der drei Personen Hans, Julia und Kevin Maier modelliert. Hans und Julia sind Ehepartner, Kevin deren Kind.

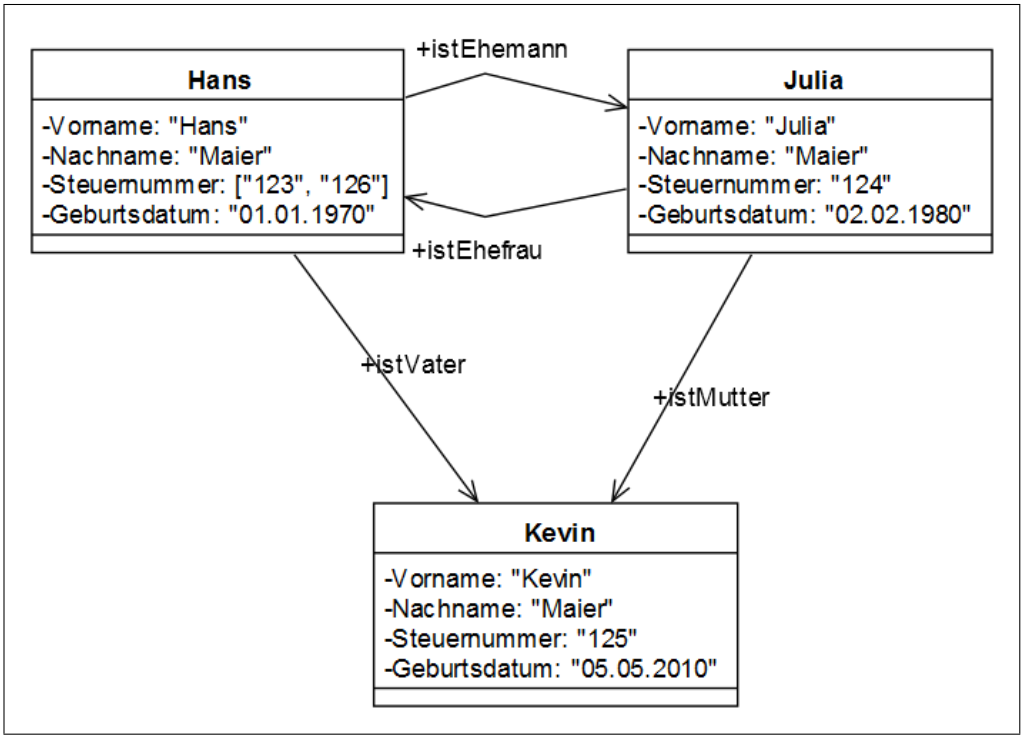

Abbildung 2.6: Grafische Darstellung eines einfachen Modells basierend auf Abbildung 2.4, eigene Darstellung, Stand: 22.05.2015

Die JSON-Darstellung der Klasse `Hans` ist in Abbildung 2.7 zu sehen. Das Modell besteht, wie auch das Metamodell, aus einem großen JSON-Objekt. Dieses enthält für jede Klasse und jede Referenz ein Unterobjekt. Die Schlüssel dieser Objekte sind typischerweise Universally Unique Identifier (UUID), können aber auch andere eindeutige Bezeichner sein. Stellt das Objekt eine Klasse dar, ist im Feld `mClass` die entsprechende Klasse des Metamodells, die Metaklasse, gelistet. Bei Referenzen steht die entsprechende Referenz des Metamodells im Feld `mRef`. Das Feld `outputs` innerhalb der Klasse enthält die nach Typ sortierten ausgehenden Referenzen der Klasse. Dies wird als Listen der IDs dargestellt, die die Referenzen bezeichnen. `inputs` beschreibt die eingehenden Referenzen der Klasse. Die entsprechenden Felder bei Referenzen sind `source` und `target`. Unter `mAttributes` sind die im Metamodell definierten Attribute der Klasse aufgelistet. Da Attribute eine untere und obere Grenze besitzen, können unter

Umständen auch mehrere Attribute der gleichen Art existieren, wie bspw. beim Feld `Steuernummer` zu sehen ist. Deshalb werden die Attributwerte immer als Liste dargestellt.

```
"846bc8a2-401f-0b0252516aee": {
    "mClass": "Mann",
    "outputs": {
      "istVater": ["8e9b1093-4ae4-1b3d63a3f842"],
      "istEhemann": ["ee204744-49d4-1442e8bc70c4"]
    },
    "inputs": {
      "istEhefrau": ["666d4de7-4620-d5469b40be1f"]
    },
    "mAttributes": {
      "Vorname": ["Hans"],
      "Nachname": ["Maier"],
      "Steuernummer": ["123", "126"],
      "Geburtstag": ["01.01.1970"]
    }
  }
```

Abbildung 2.7: JSON-Darstellung eines Objektes im Modell bei MoDiGen, eigene Darstellung, Stand: 22.05.2015

Das JSON-Modell enthält die inhaltlichen Informationen des grafisch modellierten Modells. Grafische Informationen, wie Größe und Positionierung der einzelnen Klassen und Referenzen spielen hierbei keine Rolle. Somit ist die Rekonstruktion des grafischen Modells aus der JSON-Darstellung zwar inhaltlich, nicht aber optisch möglich.

2.4.4 Umgang mit Referenzen

Wie in Scheidgen (2013, S. 2) beschrieben, werden in der XMI-Darstellung der Metamodelle in EMF Referenzen als Kind-Knoten innerhalb von Klassen dargestellt. Im Beispiel aus Abbildung 2.4 wären somit die Referenzen `istEhemann` und `istVater` innerhalb von `Mann` definiert. Im Feld `eType` stünde hierbei die Zielklasse der Referenz, also bspw. die Klasse `Frau` bei der Referenz `istEhemann`.

Diese Form der Darstellung von Referenzen wie sie im Ecore angewandt wird hat jedoch einige Nachteile. Objekte können nur als Ganzes in den Speicher geladen werden. Will man eine Referenz bearbeiten, muss die gesamte Klasse inklusive aller Referenzen geladen werden. Bei sehr großen Modellen kann dies dazu führen, dass der Heap-Speicher schnell komplett gefüllt ist. Will man über

alle Referenzen einer Klasse iterieren, könne in konstanter Zeit auf die „nächste“ Referenz zugegriffen werden. Bei der Suche nach einer bestimmten Referenz müsse allerdings auf jede Referenz ein Predicate angewandt werden; dies sei also nur in linearer Zeit möglich, vgl. Scheidgen (2013, S. 2).

Ein weiterer von Scheidgen (2013, S. 2f) genannter Ansatz zur Speicherung von Referenzen ist als relationale Struktur, wie in einer SQL-Datenbank. Referenzen sind somit nicht mehr direkt Teil der Klassen, und können einzeln abgefragt werden. Die Abfrage einer einzelnen Referenz sei, abhängig von der verwendeten Datenbank und den Indexen, langsamer als in konstanter, aber schneller als in linearer Zeit durchführbar. Das Iterieren über alle Referenzen sei hingegen vergleichbar langsam.

Ein ähnlicher Ansatz wurde, wie in Gerhart et al. (2015, S. 2) beschrieben, bei MoDiGen gewählt. Auch hier sind Referenzen unabhängig zu den dazugehörigen Klassen abrufbar. Dies geschieht allerdings nicht anhand einer relationalen Struktur, sondern als gleichwertige Darstellung von Klassen und Referenzen als *first-level*-Objekte innerhalb der JSON-Struktur des Metamodells. Dokumentorientierte Datenbanken die auf JSON-Daten arbeiten, wie MongoDB oder CouchDB, erlauben somit einen einfachen und gleichermaßen schnellen Zugriff auf einzelne Klassen und Referenzen in logarithmischer Zeit.

Die Einführung der Bounds-Attribute innerhalb der `M_LINKDEF`-Objekte bei MoDiGen erlaubt außerdem die Modellierung von `1:1`–, `1:n`–, `n:1`– und `n:m`-Beziehungen ohne zusätzlichen Aufwand. Bei Ecore hingegen besitzt eine Referenz genau einen Anfangs– und einen Endpunkt, weshalb die Modellierung von Referenzen mit mehreren Anfangs– oder Endpunkten nicht ohne Weiteres möglich ist. Diese einfache Darstellung der Mehrfachverbindungen ist ein großer Vorteil des MoDiGen-Metamodells gegenüber anderen Metamodellen.

3 Umsetzung

Die Umsetzung besteht aus drei Teilen, auf welche im Folgenden eingegangen wird.

Der erste Teil, beschäftigt sich mit dem Entwicklungsvorgang und der Struktur des grafischen Editors zur Metamodellierung. Hierbei werden die genutzten Features der Bibliothek JointJS, sowie die eigenen Erweiterungen am Editor erläutert.

In Teil zwei, Export des Metamodells, wird der Vorgang der Konvertierung des grafisch modellierten Metamodells zur festgelegten JSON-Struktur betrachtet.

Teil drei, Validierung von Modell-Instanzen, erläutert den Vorgang der Entwicklung des Werkzeugs zur Validierung von Modellen gegen ein Metamodell. Besonders wird die Eigenschaft und deren Umsetzung hervorgehoben, dass dasselbe Programm zur Validierung sowohl browser– als auch serverseitig genutzt werden kann.

3.1 Grafischer Editor zur Metamodellierung

Für die Modellierung von Metamodellen wurde ein grafischer Editor entwickelt.

Der Editor wird in modernen Web-Browsern ausgeführt. Somit ist er vollkommen plattformunabhängig und kann auf allen Systemen gleichermaßen verwendet werden. Die grafische Oberfläche ist ansprechend und einfach zu bedienen, sodass Metamodelle auch ohne große Einarbeitungszeit modelliert werden können. Der Export des Metamodells als JSON kann mit einem Klick ausgeführt werden.

Im Folgenden werden die verschiedenen Eigenschaften des Editors und deren Umsetzung näher betrachtet.

3.1.1 Aufbau des Metamodell-Editors

JointJS ist eine Bibliothek, mit welcher grafische Editoren erstellt werden können. Sie steht unter der Mozilla Public License Version 2.0 (MPL) und ist somit

Open Source und frei verfügbar. Man ist frei, Programmteile zu ändern und zu veröffentlichen, solange Programmteile, welche vorher unter der MPL standen, weiterhin unter der MPL stehen, vgl. Mozilla Foundation (2012).

Das Rappid Diagramming Toolkit nutzt JointJS als Grundlage und erweitert es nach eigenen Angaben um mehr als 20 Plugins, Komponenten für die Benutzeroberfläche, Diagramme, Formen und Funktionalitäten (vgl. client IO, 2015b). Es steht unter einer kommerziellen Lizenz, welche es, einmal gekauft, erlaubt, eine beliebige Anzahl an Applikationen mit dem Toolkit zu entwickeln.

Der Metamodell-Editor nutzt das Rappid Toolkit als Grundlage. Für die Oberfläche wurde eine mit Rappid mitgelieferte übersichtliche und gut bedienbare Beispieloberfläche angepasst und erweitert, siehe Abbildung 3.1.

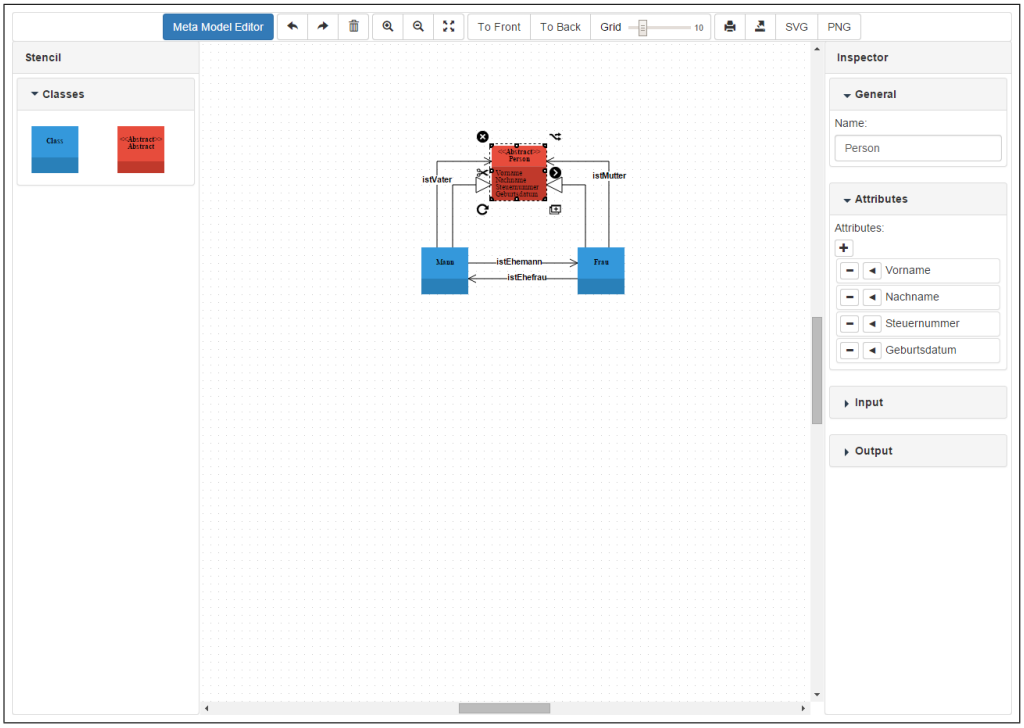

Abbildung 3.1: Grafische Oberfläche des Metamodell-Editors, Screenshot, Stand: 10.06.2015

Die Oberfläche besteht aus vier Bereichen.

Die Spalte im linken Bereich nennt sich *Stencil*, Schablone. Hier sieht man alle möglichen Elemente, die für die Modellierung verwendet werden können. Zur besseren Übersicht können die Elemente gruppiert werden. Da im MoDiGen-Metamodell allerdings nur zwei Arten von Elementen vorgesehen sind, Klassen und Abstrakte Klassen, gibt es nur eine Gruppe *Classes*, unterhalb welcher die beiden Elemente zu sehen sind. Per Drag-and-Drop können einzelne Elemente aus dem Stencil-Bereich auf der Zeichenfläche abgelegt und dort weiter bearbeitet

werden.

Der obere Bereich, die *Toolbar*, bietet Funktionen für die Modellierung. Von links nach rechts führen die Icons die folgenden Aktionen aus:

- **Undo** macht die letzte Aktion rückgängig. Als Aktion gilt bspw. das Erstellen oder Löschen von Elementen, das Verschieben von Elementen, das Zeichnen von Referenzen oder das Ändern von Attributen.

- **Redo** macht die letzte Undo-Operation rückgängig. Führt man Redo aus, ohne vorher eine Undo-Operation getätigt zu haben, geschieht nichts.

- **Clear Paper** löscht alle Elemente und Referenzen von der Zeichenfläche.

- **Zoom In** vergrößert die Ansicht des Modells.

- **Zoom Out** verkleinert die Ansicht des Modells.

- **Toggle Fullscreen Mode** öffnet den Editor im Vollbild-Modus.

  To Front ändert die Ebene eines markierten Objektes so, dass es über anderen Objekten oder Referenzen liegt.

  To Back schiebt ein markiertes Objekt in eine Ebene hinter andere Objekte oder Referenzen.

  Grid ändert mithilfe des Schiebereglers die Größe des Gitters im Hintergrund der Zeichenfläche. Beim Verschieben von Elementen schnappen diese am Gitter ein, was es möglich macht, einzelne Elemente pixelgenau zueinander auszurichten.

- **Open Print Dialog** öffnet ein Fenster, über welches das aktuelle Modell gedruckt werden kann.

- **Export as JSON** exportiert das aktuelle Modell in seine JSON-Darstellung, und öffnet diese als Text in einem neuen Browser-Tab.

  SVG öffnet das aktuelle Modell als SVG-Grafik in einem neuen Browser-Tab.

  PNG öffnet das aktuelle Modell als PNG-Grafik in einem neuen Browser-Tab, von wo aus es als Grafik gespeichert werden kann.

Die Spalte im rechten Bereich, der *Inspector*, erlaubt die Änderung von Attributen einzelner Objekte. Ist ein Objekt markiert, öffnet sich in diesem Bereich der Inspector für das Element. Welche Attribute es dort zur Auswahl gibt, hängt vom

Typ des Elements ab. Klassen haben bspw. die vier Gruppen *General*, *Attributes*, *Input* und *Output*, während der Inspector für Referenzen aus den Gruppen *General*, *Deletion*, *Attributes*, *Source* und *Target* besteht. Diese Gruppen enthalten Textfelder, Listen, Checkboxen etc., deren Werte allesamt bearbeitet werden können. Die Werte dieser Felder werden direkt im markierten Element gespeichert. Somit können über den Inspector alle Arten von Namen, Attributen etc. der Elemente bearbeitet werden. Ist kein Element markiert, wird der Inspector zum Editieren der *mEnum*s angezeigt (siehe Unterabschnitt 3.1.4).

Der große Hauptbereich in der Mitte wird *Paper* oder Zeichenfläche genannt. In ihm geschieht die grafische Modellierung. Per Drag-and-Drop eines Elements aus dem Stencil-Bereich wird ein Element auf dem Paper erstellt. Per Klick auf ein Element auf dem Paper öffnet sich der entsprechende Inspector. Per Drag-and-Drop eines Elements auf der Zeichenfläche kann dieses verschoben werden.

Ist ein Element auf dem Paper markiert, wird um das Element herum das sogenannte *Halo* angezeigt. Dies ist in Abbildung 3.1 um das rote Element herum zu sehen. Das Halo besteht aus fünf Schaltflächen, welche von oben links im Uhrzeigersinn die folgenden Funktionen durchführen:

- **Remove the object** löscht das Objekt und alle ein- und ausgehenden Referenzen.
- **Clone and connect the object in one go** kopiert ein Objekt mitsamt allen Attributen und veränderten Darstellungen, und verbindet dieses zur gleichen Zeit mithilfe einer Association mit dem Ursprungsobjekt.
- **Connect the object** verbindet das Objekt per Drag-and-Drop mit einem anderen Objekt.
- **Clone the object** kopiert das Objekt mit allen Attributen und Änderungen.
- **Rotate the object** rotiert das Objekt um einen beliebigen Winkel.

3.1.2 Auswahl der Referenztypen

Verbindet man in einem JointJS-Editor ein Element mit einem Anderen, so hat die entstehende Referenz eine festgelegte Standard-Darstellung. Das Aussehen dieser Referenz, Farbe, Dicke, Darstellung von Anfangs– und Endpunkt, könnte unter Umständen im Inspector-Fenster angepasst werden. Allerdings ist es im Metamodell-Editor wichtig, dass es einen genau vorgegebenen Satz an Referenztypen gibt, da diese eine unterschiedliche Semantik besitzen. Somit ist die vorgesehene Verwendung der Referenzen, bei denen man Darstellung

frei verändern kann, für den Metamodell-Editor nicht sinnvoll einsetzbar. Aus diesem Grund musste die Logik zum Erstellen von Referenzen angepasst und erweitert werden.

Das Erzeugen einer Referenz funktioniert im Metamodell-Editor folgendermaßen: Führt man eine Mousedown-Aktion auf die `Connect the object`–Schaltfläche der Halo eines Elements aus und befindet sich anschließend im Mouseover-Zustand über einem anderen Element, öffnet sich ein Auswahlfenster mit allen erlaubten Referenztypen. Nun kann man die Mouseup-Aktion auf dem gewünschten Referenztyp ausführen. Die Referenz mit der entsprechenden Darstellung und den korrekten Attributen wird somit erzeugt, siehe Abbildung 3.2.

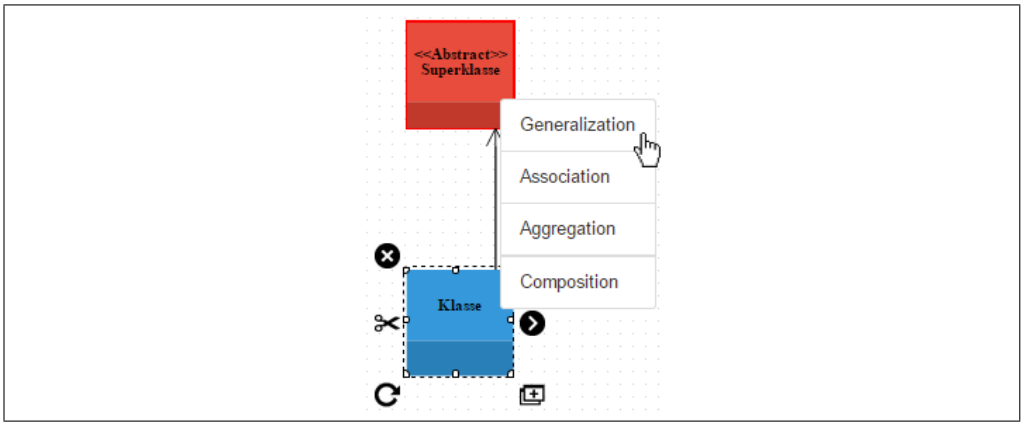

Abbildung 3.2: Auswahlfenster für den Referenztyp, Screenshot, Stand: 10.06.2015

Wie oben erwähnt, werden nur diejenigen Referenztypen vorgeschlagen, welche zwischen den beiden Elementen erlaubt sind. Nach aktuellem Stand sind die erlaubten Referenztypen, unabhängig vom Typ des Quell– oder Ziel-Elements, immer *Generalization*, *Association*, *Aggregation* und *Composition*. Am Anfang der Entwicklung sah das Meta-Metamodell allerdings noch den Element-Typ *Interface* vor, welcher wieder verworfen wurde. Dieser Typ verlangte nach einem weiteren Referenztyp, der *Implementation*, und diese konnte ausschließlich dann angewandt werden, wenn das Ziel der zu zeichnenden Referenz vom Typ *Interface* war. In allen anderen Fällen sollte dieser Referenztyp nicht vorgeschlagen werden.

Da zukünftige Änderungen am Meta-Metamodell, welche Element– oder Referenztypen mit eigenen Verbindungseigenschaften hinzufügen oder verändern, nicht ausgeschlossen sind, wurde für den Fall der *Implementation* kein Sonderfall im Code verankert, sondern eine Struktur erdacht, mit der solche Fälle beschrieben werden können.

So gibt es zwei Matrizen, die *Source*– und die *Target-Matrix*. Diese sagen aus, bei welchen Element-Typen welche Referenztypen ein– bzw. ausgehen dürfen.

Abbildung 3.3 zeigt einen Eintrag aus der Source-Matrix, welcher aussagt, dass der Referenztyp `uml.Generalization` den Element-Typ `uml.Class` als Quelle haben darf. Das Feld `name`, welches hier mit *Generalization* belegt ist, beschreibt den Namen der Referenz, wie er im Auswahlfenster in Abbildung 3.2 zu sehen ist. Somit ist es möglich, die Referenzen intern mit anderen Namen zu behandeln, als sie dem Benutzer auf der Benutzeroberfläche angezeigt werden.

```
1  "uml.Class": {
2     "uml.Generalization": {
3        "name": "Generalization",
4        "connectable": true
5     }, [...]
6  }
```

Abbildung 3.3: Eintrag in der Source-Matrix, eigene Abbildung, Stand: 11.05.2015

Für Interface und Implementation wären die folgenden Werte in Source– und Target-Matrix eingetragen:
Der Referenztyp Implementation darf als Quelle jeden Element-Typ haben außer Interface, denn nur Klassen und abstrake Klassen können Interfaces implementieren. Deshalb wäre in der Source-Matrix der Wert von `connectable` beim Referenztyp Implementation innerhalb des Element-Typs Interface *false*, bei allen anderen Element-Typen *true*.
Das Ziel des Referentyps Implementation muss immer ein Interface sein. Deshalb wäre in der Target-Matrix der Wert von `connectable` beim Referenztyp Implementation nur beim Element-Typ Interface *true*, bei allen anderen Element-Typen *false*.

Beim Modellieren einer Referenz werden also stets die möglichen Referenztypen für die Quell– und Zielklassen aus der Matrix ausgelesen und angezeigt. Mit diesen Datenstrukturen kann einfach ausgedrückt werden, welche Referenzen in welche Richtung mit welchen Elementen verbunden werden können.

3.1.3 Inspector als Attribut-Editor

Elemente und Referenzen haben Attribute und Eigenschaften, welche im Meta-Metamodell festgelegt sind (vgl. Abschnitt 2.4). Einige dieser Eigenschaften werden durch den Kontext bestimmt. Ob der Wert von `abstract` bei der `M_CLASS` *true* oder *false* ist, hängt davon ab, ob als Typ des Elements im Metamodell-Editor *Class* oder *Abstract* gewählt wurde; die `superTypes` werden durch die modellierte Vererbungshierarchie festgelegt; die Werte von `targetDeletionDeletesSource` und `sourceDeletionDeletesTarget` hängen vom gewählten

Referenztyp ab. Alle anderen Attribute und Eigenschaften können vom Benutzer händisch festgelegt und verändert werden. Die Bearbeitung all dieser Eigenschaften geschieht im Metamodell-Editor über den Inspector.

JointJS beschreibt das im Rappid-Toolkit enthaltene Plugin Inspector als „[a]n extremelly [sic] flexible and completely configurable properties editor and viewer“ (client IO, 2015b). Er erlaubt es, Attribute eines Elements übersichtlich darzustellen und bietet die Möglichkeit, diese an gleicher Stelle auch zu bearbeiten. Der Inspector ist somit immer mit genau einem Element verknüpft. Technisch enthält das JavaScript-Objekt, welches ein Element auf der Zeichenfläche darstellt, das Objekt `attributes`. Wird beim Klick auf das Element der entsprechende Inspector generiert, werden im Inspector alle Attribute angezeigt deren Werte unter dem `attributes`-Objekt festgelegt sind und deren Schlüssel eine festgelegte Darstellung in der Inspector-Konfiguration besitzen. Innerhalb des Objektes sind die Attribute einfache Schlüssel-Wert-Paare ohne Informationen über deren grafische Darstellung. Der Inspector muss somit schon vorher wissen, welche Schlüssel die Attribute haben können, und von welchem Typ die entsprechenden Werte sind, damit sie korrekt dargestellt werden können.

Welche Attribute welchen Typ haben, wird in einer einfachen JSON-Struktur zur Konfiguration festgelegt, siehe Abbildung 3.4. Wenn das Element auf der Zeichenfläche den Schlüssel `sourceDeletionDeletesTarget` besitzt, weiß der Inspector anhand dieser Konfiguration, dass dieses Feld als *Toggle*, also als Checkbox, mit dem Bezeichner „Source deletion deletes target“ innerhalb der Gruppe *deletion* angezeigt werden soll.

```
1  "sourceDeletionDeletesTarget" : {
2     "type" : "toggle",
3     "label" : "Source deletion deletes target",
4     "group" : "deletion",
5     "readonly" : true
6  }
```

Abbildung 3.4: Konfiguration des Inspectors, eigene Darstellung, Stand: 13.05.2015

Mit `group` können mehrere Attribute optisch in einer Gruppe vereinigt werden. In Abbildung 3.5 ist bspw. die Gruppe *Attributes* zu sehen. Diese können zur besseren Übersicht per Klick auf das Label ein– und ausgeblendet werden. Welchen Titel eine Gruppe im Inspector hat, und in welcher Reihenfolge die einzelnen Gruppen angezeigt werden, wird in einem eigenen Objekt zur Konfiguration der Gruppen festgelegt.

`readonly` sagt aus, ob das Feld vom Benutzer nur gelesen, oder auch geschrieben werden darf. Der Inspector bringt diese Funktionalität nicht mit, weshalb er unter

Anderem deshalb um eigene Funktionen erweitert werden musste. Diese Funktionalität kommt bei den Referenz-Attributen `sourceDeletionDeletesTarget` und `targetDeletionDeletesSource` zum Einsatz, da die Werte dieser Attribute zwar für den Benutzer sichtbar sein sollen, sie jedoch durch den Typ der Referenz bereits festgelegt und nicht änderbar sind. So ist bspw. eine *Composition* eine „has a"-Beziehung, somit muss das Ziel-Element gelöscht werden, wenn das Quell-Element entfernt wird; der Wert von `sourceDeletionDeletesTarget` muss unveränderbar auf dem Wert *true* stehen.

Unter `type` wird der Typ des Attribut-Wertes festgelegt. Beim Beispiel in Abbildung 3.4 ist dies *Toggle*, der Inspector unterstützt jedoch viel mehr Datentypen. Diese werden in ihrer HTML-Darstellung auf der Benutzeroberfläche in verschiedene `input`-Felder übersetzt. Die Darstellung dieser `input`-Felder hängt zu großen Teilen vom verwendeten Web-Browser ab:
Die Typen **number**, **text**, **range** und **color** erzeugen im HTML-Code `input`-Felder, bei denen das entsprechende `type`-Attribut gesetzt ist. Der Web-Browser Google Chrome fügt z.B. einem `number`-Feld zwei kleine Buttons hinzu, mit welchen die eingegebene Nummer erhöht oder verringert werden kann. Legt man als Typen **toggle** fest, wird dies im Web-Browser als Checkbox dargestellt, **select** erzeugt ein Dropdown-Menü mit welchem eine von mehreren festgelegten Optionen ausgewählt werden kann. Weitere wichtige Typen, die im Editor Anwendung finden, sind **list** und **object**.

Die Darstellung einiger der genannten Typen der Attribut-Werte kann in Abbildung 3.5 betrachtet werden. `Attributes` ist eine Liste von Objekten. Mit dem Plus-Zeichen unter `Attributes` kann ein Objekt dieser Liste hinzugefügt, mit dem Minus-Zeichen dieses aus der Liste entfernt werden. In der Abbildung besteht die Liste aus vier Objekten, Vorname, Nachname, Steuernummer und Geburtsdatum, wobei drei der vier Objekte zur besseren Übersicht minimiert sind. Die Objekte der Liste bestehen aus mehreren Feldern verschiedener Typen. *Name* ist bspw. vom Typ `text`, *Upper Bound* vom Typ `number`. Das Feld *Type* ist vom Typ `select` und bietet eine Auswahl zwischen String, Integer, Float und Boolean an. Die Felder *Unique local*, *Unique global* etc. sind vom Typ `toggle`, und können entweder abgehakt oder nicht abgehakt sein.

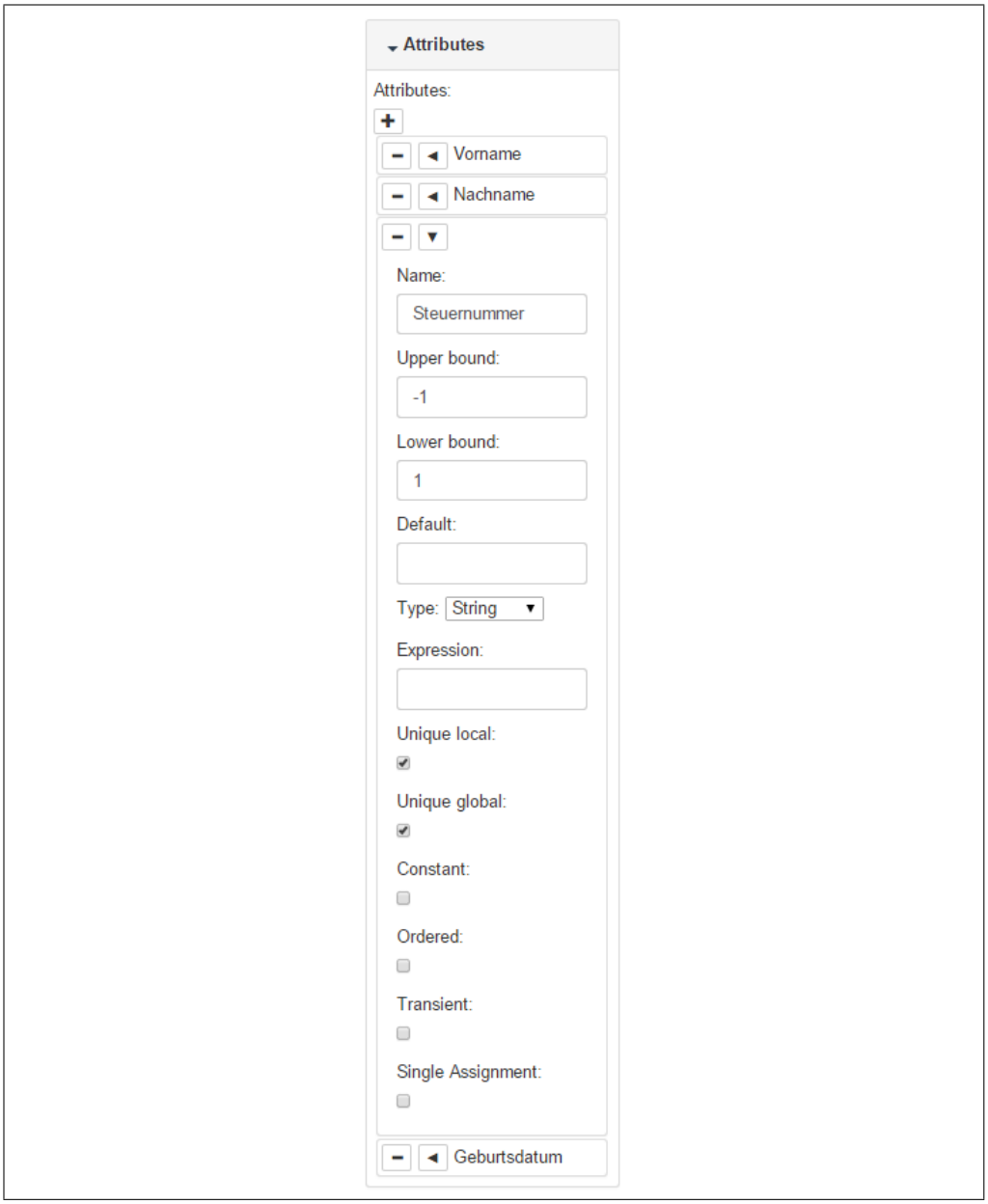

Abbildung 3.5: Darstellung der Attribute im Inspector, Screenshot, Stand: 10.06.2015

Die Struktur der Attribute spiegelt genau die im Meta-Metamodell festgelegte Struktur wider. So hat ein `M_ATTRIBUTE` bspw. die eigenen Felder `default`, `type` etc., erbt außerdem von `M_OBJECT` und `M_BOUNDS`, hat deshalb auch die Felder `name`, `upperBound` und `lowerBound` (vgl. Abbildung 2.3). Für den Aufbau der Konfiguration des Inspectors gab es somit nun zwei Möglichkeiten. Man hätte die Hierarchien auflösen und jede Gruppe unabhängig von den anderen Gruppen definieren können. Dies hätte allerdings zu Wiederholungen im Code geführt, da bspw. `M_LINKDEF` und `M_ATTRIBUTE` beide die Attribute `upperBound` und

`lowerBound` eigenständig implementieren müssten. Deshalb wurde die Konfiguration des Inspectors hierarchisch aufgebaut. Dafür wird die `extend`-Funktion der JavaScript-Bibliothek Underscore.js genutzt. Underscore.js wird intern von JointJS an vielen Stellen genutzt, weshalb die Abhängigkeit zu dieser Bibliothek bereits besteht. Die `extend`-Funktion nimmt eine beliebige Anzahl von Objekten, und erweitert das erste Objekt um die Werte aller folgenden Objekte. Somit ist der Aufbau einer Hierarchie in der Konfiguration des Inspectors möglich, wie in Abbildung 3.6 dargestellt.

```
1 M_ATTRIBUTE = _.extend({
2     "default" : {
3         [...]
4     },
5     "type" : {
6         [...]
7     },
8     [...]
9 }, M_OBJECT, M_BOUNDS);
```

Abbildung 3.6: Vererbungshierarchie in der Inspectorkonfiguration, eigene Darstellung, Stand: 13.05.2015

In Abbildung 3.6 sieht man gekürzt den Aufbau der Konfiguration des Inspectors für den `M_ATTRIBUTE`-Typ. Wie im MoDiGen-Metamodell festgelegt, erbt `M_ATTRIBUTE` von `M_OBJECT` und von `M_BOUNDS`, deshalb wird das `M_ATTRIBUTE`-Objekt mit Hilfe der `extend`-Funktion um die Objekte `M_OBJECT` und `M_BOUNDS` erweitert.

3.1.4 mEnum-Datentyp

Neben `M_CLASS` und `M_REFERENCE` ist `M_ENUM` ein weiteres Element des Meta-Metamodells. Er besitzt Werte, die vom Benutzer eigenständig angelegt und bearbeitet werden können. Dem Benutzer muss die Möglichkeit gegeben werden, beliebig viele `M_ENUM`s anzulegen und diesen einen Namen, einen Datentyp und eine Liste von Werten zuweisen zu können. Da alle benutzerdefinierten Werte von Klassen und Referenzen über den Inspector gesteuert werden können, werden die `M_ENUM`s auch mithilfe des Inspectors angelegt und bearbeitet.

Der Inspector in JointJS stellt eine direkte Beziehung zwischen einem Element auf der Zeichenfläche und der Ansicht seiner Attribute in der Seitenleiste dar. Allerdings sind `M_ENUM`s keine Objekte, welche zum Modellieren auf der Zeichenfläche zu sehen sein sollen, sondern beschreiben nur Datentypen, welche wiederum in den Attributen anderer Objekte als Datentyp genutzt werden können. Deshalb wurde `M_ENUM` so umgesetzt, dass es zwar im Editor auf der Zeichenfläche ein eindeutiges Element gibt, welches als `M_ENUM`-Container fungiert, dieses aber nicht sichtbar ist. Dieses Element wird beim Start des Editors generiert und unsichtbar auf der Zeichenfläche platziert. Im Gegensatz zu Klassen und Referenzen kann dessen Inspector nicht beim Klick auf das Element

generiert werden. Deshalb wird der Inspector des `M_ENUM`-Containers immer genau dann angezeigt, wenn gerade kein anderes Element auf der Zeichenfläche markiert ist. Dies wird durch Klicken auf eine freie Stelle der Zeichenfläche erreicht.

Mit dieser Umsetzung kann dieses Container-Objekt intern genauso behandelt werden wie Klassen, und stellt somit keinen Sonderfall dar.

Da beliebig viele Enums definiert werden können, besteht der Inspector des Containers aus einer Liste von Objekten. Diese Objekte enthalten, wie im Meta-Metamodell definiert, ein Textfeld für den Namen, ein Dropdown-Menü mit den erlaubten Datentypen String, Integer und Float, sowie eine Liste von Textfeldern für die Werte, siehe Abbildung 3.7.

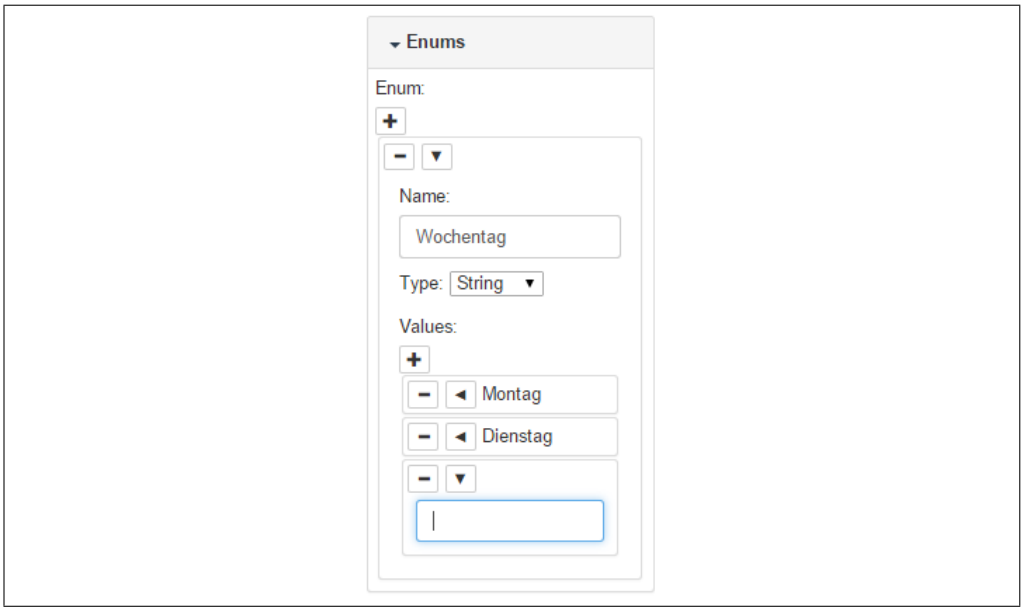

Abbildung 3.7: Darstellung der mEnum-Datentypen im Inspector, Screenshot, Stand: 10.06.2015

Alle hier definierten `M_ENUM`s werden direkt nach dem Anlegen bei den Attributen aller Klassen oder Referenzen im Dropdown-Menü zur Auswahl des `type` angezeigt, und können dort als Datentyp verwendet werden.

3.1.5 Design des grafischen Editors

Eine weitere Anforderung an den Metamodell-Editor war, dass er nicht nur gut funktionieren und die gewünschten Ergebnisse liefern, sondern auch optisch ansprechend und benutzerfreundlich sein sollte. Die Grundlage für die Benutzeroberfläche des Editors ist eine mit JointJS mitgelieferte Beispieloberfläche. Das

Layout ist übersichtlich und nutzt den vorhandenen Platz gut aus. Allerdings wirkt das Design nicht ansprechend, weshalb es in zwei Stufen ansprechender gestaltet und modernisiert wurde, wie in Abbildung 3.8 zu sehen ist:

1. Das ursprüngliche Design der Beispieloberfläche, in Abbildung 3.8 oben links zu sehen, besteht hauptsächlich aus verschiedenen Grautönen. Überschriften sind mit zwei dunklen Grautönen hinterlegt, und der Hintergrund des Stencil-Bereiches ist in einem helleren Grau gehalten. Elemente sind mit dünnen hellgrauen Linien voneinander abgetrennt und die Buttons in der Toolbar haben einen schwachen Farbverlauf von Weiß zu hellem Grau. Diese Farben wirken nicht modern und ergeben kein ansprechendes Bild.

2. Um die Oberfläche moderner zu gestalten, wurde der Editor, wie in Abbildung 3.8 oben rechts zu sehen ist, mit den folgenden Farben aus der modernen Farbpalette von Flat UI Colors neu gestaltet, siehe Sülek (2015):

   Wet Asphalt (#34495e) wird als Hintergrundfarbe hinter dem Titel *Meta Model Editor*, sowie als Farbe der Plus– und Minus-Buttons im Überschrift-Bereich von Stencil und Inspector verwendet.

   Midnight Blue (#2c3e50) ist etwas dunkler als Wet Asphalt, und ist die Hintergrundfarbe der Stencil– und Inspector-Spalten.

   Nephritis (#27ae60) ist die Hintergrundfarbe der Toolbar, sowie der Titel-Bereiche der Gruppen bei Stencil und Inspector.

   Emerald (#2ecc71) ist etwas heller als Nephritis und wird als Hintergrundfarbe der Inhalte bei Stencil und Inspector verwendet.

   Außerdem befinden sich zwischen verschiedenfarbigen Bereichen nun keine dünnen Linien mehr; die Abgrenzung erfolgt alleine durch die Farbunterschiede. Des Weiteren wurde auch das Aussehen der Elemente zur Modellierung, Klassen und Abstrakte Klassen, angepasst. Sie haben keine dicken schwarzen Rahmenlinien mehr, was ihnen ein leichter wirkendes Aussehen verleiht.

3. Für das finale Design, in Abbildung 3.8 unten, wurde die Oberfläche stark überarbeitet und mit Hilfe des „[...] most popular HTML, CSS and JS framework for developing responsive, mobile first projects on the web“ (Otto et al., 2015) Bootstrap angepasst.
   Bootstrap bietet viele Möglichkeiten zur einfachen und schnellen Gestaltung einheitlicher Benutzeroberflächen von Webseiten. Das Framework besteht aus einer Cascading Style Sheets (CSS)– und einer JavaScript-Datei, welche man in die Webseite einbindet. Ist Bootstrap eingebunden,

bietet es viele verschiedene CSS-Klassen mit verschiedenen Styles. Große Bereiche des Editors, wie bspw. der Stencil– und der Inspector-Bereich, haben nun die CSS-Klassen `panel` und `panel-default`, wodurch sie ihr typisches Aussehen mit abgerundeten Ecken bekommen. Die Toolbar besteht aus mehreren `btn-group`s, mit welchen die Buttons thematisch, optisch ansprechend und einheitlich gruppiert werden.
Bootstrap beinhaltet außerdem eine große Sammlung an Icons für verschiedene Anwendungsfälle, die sogenannten *Glyphicons*. Diese werden in der Toolbar des Editors als Icons verwendet und ersetzen die alten Icons von JointJS.
Ein weiterer Vorteil von Bootstrap ist die Möglichkeit, Themes einzubinden. Mit Themes kann das Aussehen von Elementen angepasst, und Farben hinzugefügt oder verändert werden. Sie bestehen meist aus nur einer CSS-Datei, welche zusätzlich zu Bootstrap in die Webseite eingebunden werden kann. Die meisten Elemente des Metamodell-Editors haben keine oder nur sehr wenige von Hand festgelegte Style-Informationen mehr, und nutzen Bootstrap gut aus. Durch das Einbinden von Themes kann das Aussehen des gesamten Editors deshalb einfach und einheitlich verändert werden.

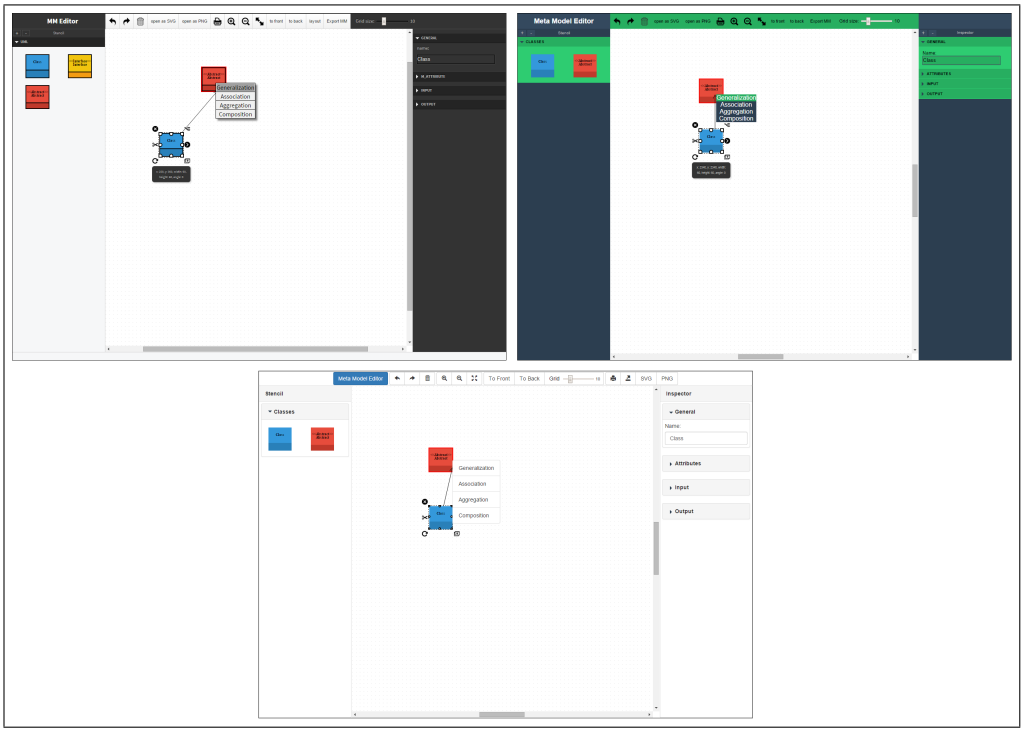

Abbildung 3.8: Designs des Metamodell-Editors, Screenshots, Stand: 16.06.2015

Durch den Einsatz von Bootstrap konnte dem Editor ein einheitliches, leichtes und helles Design gegeben werden. Dieses macht es dem Benutzer leicht, sich zurecht zu finden und den Editor zu bedienen.

3.2 Export des Metamodells

Nachdem ein Metamodell grafisch modelliert wurde, kann es zur weiteren Nutzung in seine JSON-Darstellung exportiert werden. Die JSON-Darstellung eignet sich gut zur Speicherung in einer nicht-relationalen Datenbank, und soll in Zukunft als Eingabe für einen Generator dienen, der daraus und aus zusätzlichen Style-Informationen einen grafischen Editor für zu diesem Metamodell konforme Modelle generiert.

3.2.1 CoffeeScript statt JavaScript

Die Exportierung der Metamodelle wurde nicht in JavaScript, sondern in CoffeeScript entwickelt. CoffeeScript ist „[...] a little language that compiles into JavaScript“ (Ashkenas et al., 2015) und hat die goldene Regel: „It's just JavaScript“(ebd.). CoffeeScript wird ohne weitere Optimierungen nach JavaScript übersetzt, somit kann CoffeeScript problemlos in JavaScript-Umgebungen, und bestehende JavaScript-Bibliotheken in CoffeeScript verwendet werden. Nach eigenen Angaben sei die Sprache der Versuch, die guten Seiten von JavaScript auf eine einfache Weise hervorzuheben.

Der Quellcode von CoffeeScript-Dateien ist im Allgemeinen kürzer und besser lesbar als das entsprechende JavaScript-Pendant. So vereinfacht CoffeeScript viele Aktionen, die in reinem JavaScript eher umständlich umzusetzen sind und verzichtet an vielen Stellen auf Klammern und Semikolons.

Ein großer Vorteil von CoffeeScript ist der reine objektorientierte Ansatz. So implementiert CoffeeScript auf einfache Weise Klassen, Konstruktoren und Vererbungen. Während in JavaScript die Implementierung einer Vererbung aus mehreren Schritten besteht – Übernahme der Attribute aus der Elternklasse in die Kindklasse, Überschreiben des `prototype` der Kindklasse, Überschreiben des `prototype.constructor` der Kindklasse – besitzt CoffeeScript die Keywords `class` und `extends` welche in die genannten Schritte übersetzt werden.

Außerdem bietet CoffeeScript einfache Operatoren um Prüfungen durchzuführen. So wird bspw. der unäre Fragezeichen-Operator „?“ in eine Prüfung übersetzt, ob die Variable existiert; also „`a?`“ nach „`a != null`“.

Weitere syntaktische Feinheiten erleichtern den Umgang und die Implementierung stark, machen den Code eindeutiger und besser lesbar und verkürzen viele Dinge, die in reinem JavaScript umständlich implementiert werden müssten. Für eine vollständige Dokumentation siehe Ashkenas et al. (2015).

Die Installation von CoffeeScript erfolgt über den Node Package Manager (NPM), welcher Teil der serverseitigen JavaScript-Implementierung Node.js

ist. Ist der Compiler installiert, können CoffeeScript-Dateien mit der Endung *.coffee* über die Kommandozeile zu JavaScript-Dateien übersetzt werden. Viele Integrated Development Environments (IDEs), wie bspw. IntelliJ IDEA (siehe JetBrains, 2015), unterstützen Syntax Highlighting und Codevervollständigung bei CoffeeScript-Dateien, und können diese mit Hilfe von sog. *File Watchern* bei Änderungen selbstständig nach JavaScript übersetzen. In die Website bzw. den Editor werden nur die übersetzten reinen JavaScript-Dateien eingebunden, und können somit nahtlos von anderen JavaScript-Dateien genutzt werden oder diese nutzen.

Mit Hilfe von CoffeeScript konnte der Code zum Exportieren des Metamodells einfach und übersichtlich gestaltet werden. Aufgerufen wird er aus dem normalen JavaScript-Kontext des Editors heraus.

3.2.2 Aufruf der Exportierung

Der Export des Metamodells ist objektorientiert gestaltet. Um den Export des grafisch modellierten Metamodells durchzuführen, muss eine neue Instanz der Klasse *Exporter* instanziiert werden. Der Konstruktor der Klasse erwartet das `Graph`-Objekt, wie es intern von JointJS verwaltet wird. Das exportierte Modell erhält man durch den Aufruf der `export`-Methode.

Das von der `export`-Methode zurückgegebene Objekt stellt die Methoden `isValid`, `getMetaModel`, `getMessages` und `toString` bereit.

`isValid` gibt einen Boolean-Wert zurück, der aussagt, ob das modellierte Metamodell valide ist und exportiert werden konnte. Für die Gründe, warum ein Metamodell nicht valide sein könnte, siehe Unterabschnitt 3.2.3.
`getMetaModel` gibt, wenn es exportiert werden konnte, das Metamodell als JSON-Objekt zurück, ansonsten `null`. Um die String-Darstellung des Metamodells zu erhalten, kann die `toString`-Methode des zurückgegebenen Objektes genutzt werden.
`getMessages` gibt ein Array von Strings zurück. War der Export nicht erfolgreich, enthält es Fehlermeldungen, die darauf hinweisen, wo der Fehler lag. Ansonsten ist das Array leer.

Im Metamodell-Editor wird der Aufruf per Klick auf den Export-Button () in der Toolbar angestoßen. Ist der Export erfolgreich, wird die formatierte String-Darstellung des exportierten JSON-Objektes in einem neuen Browser-Tab geöffnet. Ist der Export nicht erfolgreich, sieht der Benutzer einen *alert*-Dialog mit den Fehlermeldungen.

3.2.3 Validierung des Metamodells

Bevor das Metamodell exportiert werden kann, muss geprüft werden, ob es sich in einem Zustand befindet, in dem die Exportierung möglich ist. Nicht vieles kann den Export unmöglich machen, doch auf einige Dinge muss der Benutzer achten.

`M_CLASS`, `M_REFERENCE` und `M_ENUM` werden in der JSON-Darstellung als eigenständige, gleichwertige *first-level*-Objekte dargestellt. Der Schlüssel dieser Objekte im Metamodell ist der vom Nutzer festgelegte Name des jeweiligen Objektes. Vergibt der Benutzer den gleichen Namen an zwei dieser Objekte, ist der Schlüssel nicht mehr eindeutig.

So verhält es sich auch mit den Attributen eines Elements. Unterhalb des Objektes `mAttributes` in einer Klasse oder Referenz wird der vom Benutzer vergebene Name als Schlüssel des Attributs verwendet. Damit die Schlüssel des `mAttributes`-Objektes eindeutig bleiben, darf ein Attribut-Name innerhalb eines Elements nicht doppelt vergeben werden.

Der JSON-Standard beschreibt diesen Fall als nicht zulässig; „[t]he names within an object SHOULD [sic] be unique“ (Bray, 2014, S. 5), wobei das Wort „should“ als das empfohlene Verhalten gesehen werden muss. Auch Web-Browser lassen doppeltes Keys in einem Objekt nicht zu und überschreiben die Duplikate oder zeigen undefiniertes Verhalten.

Vor dem Export muss auf die genannten Fälle geprüft werden. Nur wenn keiner dieser Fälle zutrifft ist eine sinnvolle und korrekte Exportierung möglich.

3.2.4 Ablauf der Exportierung

Bei der Initialisierung des *Exporter*-Objektes mit dem von JoinsJS gelieferten Graph-Objekt wird ein neues, eigenes Graph-Objekt initialisiert. Diese eigene Graph-Klasse ist eine Wrapper-Klasse um den JointJS-Graphen, und bietet einige Hilfsmethoden mit denen das JSON-Objekt des Metamodells einfach zusammengebaut werden kann.

Ist das Modell valide, wird im Anschluss an die Validierung der Export durchgeführt. Dieser läuft in drei Schritten ab:

1. Es wird über alle Klassen und abstrakte Klassen iteriert. Dabei werden die Informationen jeder Klasse dem Metamodell hinzugefügt. Diese sind, wie im Meta-Metamodell beschrieben, `name`, `abstract`, `superTypes`, `inputs`, `outputs` und `mAttributes`. Außerdem hat die Klasse das Feld `mType`, was bei Klassen konstant mit dem Wert *mClass* belegt ist. Hier muss

beachtet werden, dass `M_ENUM`, wie in Unterabschnitt 3.1.4 beschrieben, zwar technisch auch als Element auf der Zeichenfläche implementiert ist, dieses allerdings nicht zu den hier zum Metamodell hinzugefügten Klassen gehört.

2. Es wird über alle Referenzen iteriert. Hierbei ist zu beachten, dass nur die Typen *Association*, *Aggregation* und *Composition* als Referenz im Metamodell auftauchen. Der Typ *Generalization* besitzt keine Attribute und wird über das Feld `superTypes` in den Klassen aufgelöst. Referenzen besitzen im Metamodell, wie im Meta-Metamodell festgelegt, die Felder `name`, `sourceDeletionDeletesTarget`, `targetDeletionDeletesSource`, `source`, `target` und `mAttributes`. Das zusätzliche Feld `mType` ist konstant mit dem Wert *mRef* besetzt.

3. Es wird über alle `M_ENUM`s iteriert. Dafür wir das Container-Element aus dem Graph extrahiert, dessen Attribute die vom Benutzer definierten `M_ENUM`s darstellen. Im Metamodell besitzen die `M_ENUM`s die Felder `name`, `type` und `values`, sowie das Feld `mType`, welches konstant den Wert *mEnum* enthält.

Nach diesen drei Schritten ist das Metamodell komplett und kann dem Benutzer zurückgegeben werden.

3.3 Validierung von Modell-Instanzen

Die dritte Ebene, nach Meta-Metamodell und Metamodell ist das Modell. In Zukunft soll das exportierte Metamodell zusammen mit Style-Informationen in einen Generator gegeben werden können, der daraus einen grafischen Editor zum Modellieren der tatsächlichen Modelle generiert. An dieser Stelle soll es nun die Möglichkeit geben die Modelle mit dem ihnen zu Grunde liegenden Metamodell abgleichen zu können. Hierbei ist zu prüfen, dass nur Klassen und Referenzen verwendet werden, die im Metamodell definiert sind, dass alle verwendeten `input`–, `output`–, `source`– und `target`-Klassen und –Referenzen laut Metamodell erlaubt sind, dass alle Bounds eingehalten sind, und dass mit `unique` markierte Attribute auch tatsächlich eindeutig sind. Trifft auch nur eine dieser Anforderungen nicht zu, ist das Modell nicht valide.

3.3.1 Aufruf des Validators

Die Besonderheit dieser Validierung ist, dass sie von der Client– und der Serverseite gleichermaßen genutzt werden können soll.

Clientseitig muss dem Benutzter des grafischen Modellierungs-Tools die Möglichkeit gegeben werden, die Validität seines Modells zu prüfen. Viele der Einschränkungen, die das Metamodell vorgibt, können schon vom grafischen Modell-Editor behandelt werden. Der Editor wird nur die Typen von Klassen und Referenzen zulassen, die im Metamodell festgelegt wurden. Außerdem wird der Editor nur `input`– und `output`-Referenzen von Klassen bzw. `source`– und `target`-Klassen von Referenzen zulassen, die laut Metamodell valide sind. Um andere „Absprachen“ muss sich der Benutzer allerdings selber kümmern. Da man erst die Klassen modellieren muss bevor man diese mit Referenzen verbinden kann, kann im grafischen Editor eine Klasse ohne eine einzige Referenz existieren. Hat diese Klasse nun `input`– oder `output`-Referenzen deren `lowerBound` größer Null ist, ist das Modell zu dem Zeitpunkt nicht valide. Der Benutzer muss sich selber darum kümmern, dass er die benötigten Referenzen anlegt. Gleich verhält es sich mit erforderlichen `source`– oder `target`-Klassen der Referenzen. Somit muss der Benutzer mit einfachen Mitteln sein aktuelles Modell gegen das Metamodell prüfen lassen können, um diese evtl. vergessenen Referenzen oder Klassen und andere Fehler im Modell aufzuspüren.

Serverseitig existiert die Schnittstelle zur Datenbank, in welcher die Modelle abgespeichert werden. In der Datenbank sollen nur valide Modelle gespeichert werden können, weshalb vor der Speicherung die Validität des zu speichernden Modells geprüft werden muss. Die alleinige clientseitige Validierung reicht hierbei nicht aus, denn „[a]lle an eine Webanwendung oder einen Web-Service übergebenen Daten, unabhängig von Kodierung oder Form der Übermittlung, müssen als potenziell gefährlich behandelt und entsprechend gefiltert werden“ (Bundesamt für Sicherheit in der Informationstechnik (BSI), 2014). Ein Angreifer könnte also potenziell gefährliche Daten an den Server senden, oder sendet mit gefälschten Anfragen nicht-valide Modelldaten an den Server. Damit diese falschen Modelldaten nicht persistiert werden, müssen sie validiert werden.

Die Herausforderung lag bei der Implementierung der Validierungslogik darin, diese gleichermaßen sowohl für die Clientseite, also für den Web-Browser, als auch für die Serverseite zugänglich zu machen. Dies wurde auf drei verschiedenen Wegen versucht, wobei der dritte Weg als die finale Lösung betrachtet werden kann.

1. **Schreiben der Logik in Scala und Kompilierung nach JavaScript**

   Der erste Ansatz beschäftigte sich mit dem *source-to-source-compiler* Scala.js (siehe Doeraene, 2015). Der Compiler kann Code, der in Scala geschrieben ist, nach JavaScript übersetzen, und erlaubt somit das Schreiben von Web-Anwendungen in Scala. Hierbei war die Idee, die Validierungslogik in Scala zu schreiben, was auf dem Web-Server laufen kann, und diese nach JavaScript zu übersetzen, um dieselbe Logik auch

clientseitig nutzbar zu machen. Es zeigte sich allerdings bald, dass dieser Ansatz zu keinem zufriedenstellenden Ergebnis führen würde.

Scala.js erfordert einen speziell annotierten Scala-Quellcode, damit korrekt nach JavaScript übersetzt werden kann. Dies schließt die Benutzung von Scala-Bibliotheken, die nicht speziell für Scala.js geschrieben und kompiliert wurden, aus. Da die Sprache Scala selber allerdings keinen ausreichenden Support für den Umgang mit JSON-Strukturen bietet und man hierfür auf externe Bibliotheken angewiesen ist, konnte die Logik nicht sinnvoll für die Kompilierung nach JavaScript in Scala implementiert werden.

2. **Doppelte Implementierung der Logik, in JavaScript und Java**

   Im zweiten Ansatz wurden zwei separate Programme entwickelt, die beide die gleiche Logik implementieren. Ein Programm in JavaScript bzw. CoffeeScript geschrieben für die Clientseite und ein Programm in Java implementiert für die Serverseite. JavaScript kann, wie die Bezeichnung JSON, JavaScript Object Notation, schon ausdrückt, nativ mit JSON-Daten umgehen. Deshalb konnte der Umgang mit den zu validierenden JSON-Daten sehr einfach in JavaScript implementiert werden. Das serverseitige in Java implementierte Programm orientierte sich sehr stark am JavaScript-Programm. Es war mit Hilfe der offiziellen JSON-Bibliothek von json.org (siehe JSON.org, 2002) implementiert und nutzte die gleichen Strukturen, Klassen und Programmabläufe wie das JavaScript-Programm.

   Dieser Ansatz funktioniert zwar tadellos, allerdings verstößt er gegen das *DRY*-Prinzip, „Don't Repeat Yourself“. Die beiden Programme haben sehr viel gemeinsam, erfüllen den gleichen Zweck und sind sehr ähnlich zueinander programmiert. Wenn in Zukunft irgendetwas an der Validierungslogik geändert werden sollte, wären Änderungen an mindestens zwei Stellen notwendig. Vergäße man eine der Änderungen, erfüllten Client– und Serverlogik nicht mehr den gleichen Zweck und könnten unterschiedliche Ergebnisse liefern. Somit ist diese Art der doppelten Implementierung der gleichen Logik sehr fehleranfällig und sollte möglichst vermieden werden.

3. **Aufrufen der JavaScript-Implementierung mit Hilfe der in Java implementierten JavaScript-Engine Rhino**

   Im dritten Ansatz wurde die doppelte Implementierung der gleichen Logik aufgelöst. Die Logik ist nun in CoffeeScript implementiert, was nach JavaScript übersetzt wird. Dieses in JavaScript implementierte Programm kann clientseitig direkt aufgerufen werden, siehe Abbildung 3.9.

```
1    var metaModel = [Metamodell-JSON als String oder JSON-Objekt];
2    var instance = [Modell-JSON als String oder JSON-Objekt];
3
4    var validator = new ModelValidator(metaModel);
5    var result = validator.validate(instance);
6    var isValid = result.isValid();
7    var messages = result.getMessages();
```

Abbildung 3.9: Aufruf des ModelValidator aus dem JavaScript-Umfeld, eigene Darstellung, Stand: 16.05.2015

Serverseitig wird die von der Mozilla Foundation in Java geschrieben JavaScript-Implementierung *Rhino* verwendet, um denselben JavaScript-Code auszuführen.
Rhino ist „[...] eine Open-Source-Implementierung von JavaScript, die vollständig in Java entwickelt wurde“ (Mozilla Foundation, 2015), und wird „[...] typischerweise in Java-Anwendungen eingebettet, um Endanwendern Skripting zu ermöglichen“ (ebd.). Es besteht aus nur einer Abhängigkeit, die im Java-Umfeld eingebettet werden muss, und bietet viele Methoden zum Erstellen von JavaScript-Objekten, Aufruf von JavaScript-Funktionen und –Methoden mit oder ohne Parameterübergabe, und dem Auswerten der Rückgabeparameter.

Der Aufruf der JavaScript-Dateien aus dem Java-Umfeld besteht somit aus vier einfachen Schritten:

1. Einlesen der JavaScript-Dateien
2. Erstellen des `ModelValidator`-Objektes
3. Aufruf der `validate`-Methode
4. Übertragen der Rückgabewerte auf ein übersichtliches Java-Objekt

Nach außen bietet der Validator eine einfache Schnittstelle ohne direkte Abhängigkeiten zu Rhino, welche so gestaltet wurde, dass der Aufruf der Validierungslogik aus Java genauso einfach durchgeführt werden kann wie aus JavaScript, siehe Abbildung 3.10.

```
1    String metaModel = [Metamodell-JSON als String];
2    String instance = [Modell-JSON als String];
3
4    ModelValidator validator = new ModelValidator(metaModel);
5    ValidationResult result = validator.validate(instance);
6    boolean isValid = result.isValid();
7    List<String> messages = result.getMessages();
```

Abbildung 3.10: Aufruf des ModelValidator aus dem Java-Umfeld, eigene Darstellung, Stand: 16.05.2015

Die Sprachen Java und Scala laufen beide auf der Java Virtual Machine

(JVM), wodurch sie kompatibel zueinander sind. Im Java implementierte Programme oder Klassen können problemlos aus Scala-Programmen heraus instanziiert und aufgerufen werden. Da die Serverseite des MoDiGen-Toolkits größtenteils in Scala implementiert ist, ist der in Java implementierte Modell-Validator problemlos von dort verwendbar.

Die Implementierung der Validierungslogik in JavaScript bietet viele Vorteile gegenüber der Implementierung in Java oder Scala. Da JavaScript nativ mit JSON-Daten umgehen kann, kann der Code viel übersichtlicher und schöner geschrieben werden als mit der Nutzung von JSON-Bibliotheken im Java– oder Scala-Umfeld.
Wo in JavaScript bspw. der Wert innerhalb eines Objektes einfach mit `objekt.key` bzw. `objekt["key"]` ausgelesen werden kann, muss dies mit der *org.json*-Bibliothek in Java über einen Funktionsaufruf geschehen: `objekt.getString(key)` bzw. `getInt`, `getBoolean`, `getJSONObject`, `getJSONArray` etc.

Auch viele andere Aktionen – ändern von Werten, iterieren über die Werte, etc. – können in JavaScript mit einfachen nativen Strukturen durchgeführt werden; in Java benötigt man dafür aufwendigere Funktionsaufrufe.

Dieser direkte Umgang mit JSON-Datenstrukturen führt dazu, dass bei Lösungsansatz 2 der Java-Code nach Entfernung unnötiger Leerzeichen, –zeilen und Kommentaren aus `23.249` ASCII-Zeichen bestand, der CoffeeScript-Code, der genau dieselben Aktionen durchführte, aus nur `8.459`. Außerdem besaß die Java-Implementierung eine Abhängigkeit zur JSON-Bibliothek, während die CoffeeScript-Implementierung eigenständig ausgeführt werden konnte. Trotz der Kürze ist der CoffeeScript-Code sehr übersichtlich und gut lesbar.

Somit ist die Implementierung der Validierungslogik in JavaScript nicht nur notwendig, damit der Web-Browser das Programm ausführen kann, sondern auch für die Serverseite eine sehr schöne und gute Lösung.

Eine vierte Möglichkeit der Implementierung wäre die Nutzung von Node.js gewesen. Node.js bietet die vollständige Software für einen Web-Server, der komplett in JavaScript implementiert werden kann. Somit wäre der JavaScript-Code zur Validierung ohne den Umweg über eine andere Programmiersprache wie Java serverseitig nutzbar. Da allerdings auf dem Server des Toolkits bereits eine JVM läuft, und die Validierung in Zukunft von anderen Programmteilen genutzt werden soll, wäre die Implementierung einer eigenen Server-Instanz mit Node.js unnötig aufwendig und würde die Nutzung der Validierung aus dem Rest des Toolkits heraus erschweren.

3.3.2 Ablauf der Validierung

Im Konstruktor der *ModelValidator*-Klasse wird ein Wrapper um das JSON-Objekt, welches das Metamodell darstellt, mit diesem initialisiert. Beim Aufruf der `validate`-Methode wird ein Wrapper um das `instance`-Objekt initialisiert. Diese Wrapper bieten einige Hilfsmethoden für die Prüfung ob bestimmte Felder in den Modellen existieren und für die Extrahierung dieser. Anschließend wird ein `ValidationResult`-Objekt angelegt, welches das Boolean-Feld `valid` und das Array-Feld `messages` enthält. `valid`, abrufbar über die Methode `isValid` enthält nach der Validierung die Information, ob das Modell gegenüber dem Metamodell valide ist, `messages` enthält Meldungen über alle bei der Validierung aufgetretenen Fehler.

Nun wird über das gesamte `instance`-Objekt iteriert, und für jeden Eintrag die folgenden Prüfungen durchgeführt:

- Ist das Objekt eine Klasse oder eine Referenz? Enthält es weder das Feld `mClass` noch `mRef`, ist die Instanz in sich fehlerhaft.

- Ist der Typ der Klasse bzw. Referenz im Metamodell definiert? Wenn nicht, ist die Instanz nicht valide.

- Sind alle `input`– und `output`-Referenzen einer Klasse bzw. alle `source`– und `target`-Klassen einer Referenz für diese Klasse oder Referenz laut Metamodell zugelassen? Ist etwas davon nicht erlaubt, ist die Instanz an dieser Stelle invalide.

- Sind die `bounds` aller `input`– und `output`-Referenzen bzw. `source`– und `target`-Klassen eingehalten? Dafür müssen die ein– und ausgehenden Verbindungen bzw. Anfangs– und Endklassen pro Typ gezählt und mit den im Metamodell festgelegten Bounds verglichen werden.

- Sind alle Attribute innerhalb eines Objektes der Instanz im Metamodell definiert? Zusätzliche nicht definierte Attribute sind nicht erlaubt.

- Sind die `bounds` der Attribute eingehalten, oder hat ein Attribut zu viele oder zu wenige Werte?

- Sind die mit `uniqueGlobal` markierten Attribute tatsächlich global eindeutig?

- Sind die mit `uniqueLocal` markierten Attribute tatsächlich innerhalb des Objektes eindeutig?

Ist eine der Fragen mit *nein* beantwortet, ist die Instanz nicht zum Metamodell konform.

Besonders ist beim Prüfen der Felder und Attribute darauf zu achten, dass das Metamodell aus Hierarchien mit Einfach– und Mehrfachvererbungen bestehen kann. Wird somit in einem Objekt bspw. ein Attribut nicht direkt gefunden, muss der Vererbungsbaum rekursiv bis zur Wurzel, bzw. den Wurzeln bei Mehrfachvererbung, nach geerbten Attributen durchsucht werden.

Die Fehlermeldungen, welche während der Validierung zum `ValidationResult`-Objekt hinzugefügt werden, sind allesamt so formuliert, dass sie dem Benutzer möglichst viele Informationen liefern. Anhand dieser Meldungen soll ermöglicht werden, den Fehler schnell zu lokalisieren und zu korrigieren. Die Oberfläche zum Testen des Validators ist in Abbildung 3.11 zu sehen. Dort ist beispielhaft die Instanz so manipuliert, dass bei der Validierung zwei Fehler gefunden werden:

1. Das Objekt mit der UUID *846bc8a2-00fc-401f-b626-0b0252516aee*, das vom Typ *Male* ist, hat bei der `input`-Referenz *isWife* zwei Einträge, obwohl im Metamodell der `upperBound` auf eins gesetzt ist.

2. Ein Objekt vom Typ *Male* besitzt ein Attribut *NonExistingAttribute*, welches im Metamodell nicht definiert wurde.

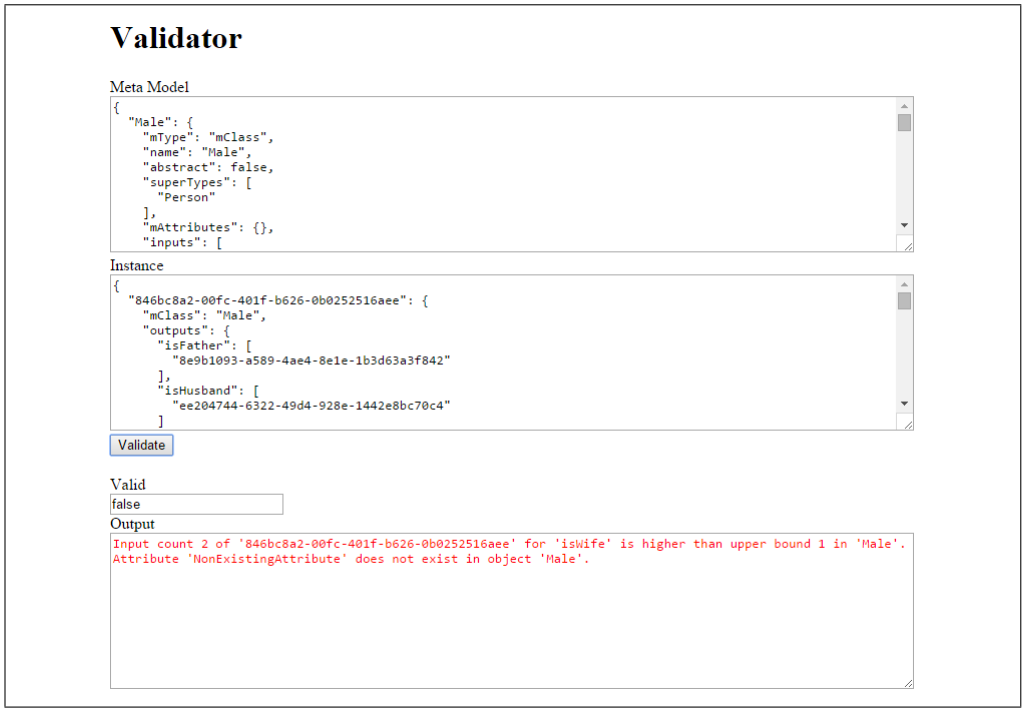

Abbildung 3.11: Test-Oberfläche des Validators im Web-Browser, Screenshot, Stand: 20.05.2015

Die Oberfläche existiert nur zum Test der Validierungslogik und soll aufzeigen, wie der Validator aufzurufen ist. In Zukunft kann der Validator direkt aus dem grafischen Modell-Editor aufgerufen werden.

4 Fazit

Das perfekte Meta-Metamodell für alle Anwendungsfälle existiert nicht. Während Ecore für die Modellierung im EMF gemacht ist (vgl. Abschnitt 2.3), scheinen EMOF und vor allem CMOF allumfassend und für jeden Sonderfall ausgelegt, allerdings auch dementsprechend komplex (vgl. Abschnitt 2.2). Das MoDiGen-Metamodell hingegen besteht aus einer abgewandelten Teilmenge des Ecore, und implementiert neue Ansätze für den Umgang mit Referenzen, welche neben `1:1`– auch die einfache Modellierung von `n:1`–, `1:n`– und `n:m`-Beziehungen erlauben (vgl. Abschnitt 2.4).

Auch die verschiedenen Datenformate zur Serialisierung der Metamodelle und Modelle haben Vor– und Nachteile. Ecore und MOF arbeiten mit XML, MoDiGen mit JSON. Die Struktur von XML kann mit Hilfe von XML-Schemas formal beschrieben werden. Dies erlaubt auch die automatisierte Validierung von XML-Dateien. JSON bietet noch keine offizielle Unterstützung von Schemas, allerdings wurde bereits ein Entwurf für JSON-Schemas vorgelegt, welcher sich zum allgemeingültigen Standard entwickeln könnte, siehe Galiegue und Court (2013). Des Weiteren bietet JSON den großen Vorteil, dass Modelle auch nur teilweise in den Hauptspeicher geladen werden können, wohingegen XML immer als Ganzes vorliegen muss, was gerade bei großen Modellen zu Platzproblemen im Hauptspeicher führen kann (vgl. Unterabschnitt 2.4.3).
Die Datenhaltung ist in beiden Fällen einfach umzusetzen. Während aus den XML-Schemas Datenbank-Schemas für relationale Datenbanken generiert werden können, können JSON-Daten direkt in dokumentenorientierte Datenbanken wie MongoDB oder CouchDB gegeben werden.

Der im Rahmen dieser Arbeit entwickelte grafische Editor zur Modellierung von Metamodellen auf Grundlage des MoDiGen-Metamodells kann nun dafür genutzt werden, um auf einfache Weise Metamodelle zu definieren. Der Export der JSON-Serialisierung erfolgt auf Knopfdruck innerhalb des Editors. Der Editor ist einfach zu bedienen und erlaubt das schnelle Erstellen auch komplexer Metamodelle (vgl. Abschnitt 3.1).

Die Validierung der Modell-Instanzen ist sowohl aus dem Web-Browser heraus, als auch serverseitig gleichermaßen einfach aufzurufen. Sie liefert aussagekräftige Meldungen, sollte das Modell gegenüber dem Metamodell nicht valide sein. Die Implementierung der Validierungslogik in JavaScript bzw. CoffeeScript führte wegen dem nativen Umgang mit JSON-Daten zu einfachem und schnellem Code, ohne die Nutzung von externen Bibliotheken. Da Client und Server auf denselben

Code zugreifen, muss dieser bei Änderungen nur an einer Stelle angepasst werden; Client und Server liefern immer die gleichen Ergebnisse (vgl. Abschnitt 3.3).

4.1 Ausblick

In naher Zukunft soll ein im Metamodell-Editor modelliertes Metamodell zusammen mit Informationen zum Styling der Elemente und Referenzen in einen Generator gegeben werden können. Die Style-Informationen werden mit Hilfe eines eigenen Editors erstellt, der auch Teil des MoDiGen-Projektes ist. Dieser Generator soll daraus einen Editor ähnlich dem Metamodell-Editor generieren, der nur zum dazugehörigen Metamodell konforme Modelle zulässt. Mit diesem Editor unterstützt das MoDiGen-Toolkit den kompletten Weg von Meta-Metamodell über die Definition des Metamodells bis hin zur Modellierung des eigentlichen zu modellierenden Sachverhaltes.

Der Aufbau der JSON-Stuktur des Metamodells ist bislang nur informell anhand der grafischen UML-ähnlichen Darstellung und anhand von von Beispielen beschrieben. Eine formale Beschreibung mit Hilfe von JSON-Schemas wäre zur allgemeingültigen und eindeutigen Dokumentation des Datenformates der Serialisierung sinnvoll.

In einer älteren Version des MoDiGen-Projektes, damals noch *Spray*, bzw. *Spray Online*, wurde eine Kollaborationsfunktionalität für JointJS entwickelt, mit welcher eine beliebige Anzahl an Benutzern zur gleichen Zeit live an einem Modell arbeiten konnten. Diese Kommunikation über WebSockets wird in naher Zukunft so angepasst, dass sie auch für den Metamodell-Editor eingesetzt werden kann, und die Metamodelle somit kollaborativ erstellt und bearbeitet werden können. Außerdem werden die serialisierten Metamodelle bislang nur lokal im Web-Browser ohne die Kommunikation zu einem Server erstellt, angezeigt und bearbeitet. Das Senden der exportierten Metamodelle per Ajax an den Webserver ist im Code des Editors bereits vorgesehen. Allerdings existierte diese serverseitige Schnittstelle zur Zeit der Fertigstellung noch nicht, und ist nicht Teil dieser Arbeit, weshalb die Verbindung noch implementiert werden muss.

Allgemein gibt es im gesamten MoDiGen-Toolkit mit seinen verschiedenen Editoren und deren Kommunikation untereinander noch einige Baustellen. Durch die Implementierung des Metamodell-Editors und des Werkzeuges zur Validierung von Modell-Instanzen wurden allerdings zwei wichtige Teile des Projektes fertiggestellt.

Literatur

Apache Software Foundation (2015). *Apache CouchDB*. https://couchdb.apache.org/. Abgerufen am 15.06.2015.

Ashkenas, Jeremy et al. (2015). *CoffeeScript*. http://coffeescript.org/. Abgerufen am 15.05.2015.

Bray, Tim (2014). *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159.

Bundesamt für Sicherheit in der Informationstechnik (BSI) (2014). *Umfassende Ein– und Ausgabevalidierung bei Webanwendungen und Web-Services*. https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/_content/m/m04/m04393.html. Abgerufen am 15.05.2015.

client IO (2015a). *JointJS – JavaScript Diagramming Library*. http://www.jointjs.com/. Abgerufen am 15.06.2015.

client IO (2015b). *Rappid – A complete Diagramming Toolkit*. http://jointjs.com/about-rappid. Abgerufen am 08.05.2015.

Doeraene, Sébastien (2015). *Scala.js – the Scala to JavaScript compiler*. http://www.scala-js.org/. Abgerufen am 16.05.2015.

Eclipse Foundation (2015). *Eclipse Modeling Framework (EMF)*. https://eclipse.org/modeling/emf/. Abgerufen am 15.06.2015.

Galiegue, Francis und G. Court (2013). *JSON Schema*. http://json-schema.org/. Abgerufen am 25.06.2015.

Gerhart, Markus, J. Bayer, J. M. Höfner und M. Boger (2015). *Approach to Define Highly Scalable Metamodels Based on JSON*. In: *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '15. ACM. Manuskript zur Publikation eingereicht.

JetBrains (2015). *IntelliJ IDEA*. https://www.jetbrains.com/idea/. Abgerufen am 23.06.2015.

JSON.org (2002). *JSON in Java*. `http://www.json.org/java/`. Abgerufen am 16.05.2015.

Kolovos, Dimitrios S., L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi und J. Cabot (2013). *A Research Roadmap Towards Achieving Scalability in Model Driven Engineering*. In: *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, S. 2:1–2:10, New York, NY, USA. ACM.

Kühne, Thomas (2006). *Matters of (Meta-) Modeling*. Software & Systems Modeling, 5(4):369–385.

MongoDB (2015). *MongoDB*. `https://www.mongodb.org/`. Abgerufen am 15.06.2015.

Mozilla Foundation (2012). *Mozilla Public Licence Version 2.0*. `https://www.mozilla.org/MPL/2.0/`. Abgerufen am 08.05.2015.

Mozilla Foundation (2015). *Rhino*. `https://developer.mozilla.org/de/docs/Rhino`. Abgerufen am 16.05.2015.

Object Management Group (2014). *Meta Object Facility 2.4.2 Core Specification*.

Otto, Mark, J. Thornton et al. (2015). *Bootstrap*. `http://getbootstrap.com/`. Abgerufen am 16.06.2015.

Scheidgen, Markus (2013). *Reference Representation Techniques for Large Models*. In: *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, S. 5:1–5:9, New York, NY, USA. ACM.

Seidewitz, Ed (2003). *What models mean*. Software, IEEE, 20(5):26–32.

Severance, Charles (2012). *Discovering JavaScript Object Notation*. Computer, 45(4):6–8.

Steinberg, Dave, F. Budinsky, M. Paternostro und E. Merks (2008). *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional.

Sülek, Ahmet (2015). *Flat UI Colors*. `http://flatuicolors.com/`. Abgerufen am 30.04.2015.