Institut für Architektur von Anwendungssysteme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 75

Entwurf und Realisierung von REST- Anwendungen nach Prinzipien der modellgetriebenen Softwareentwicklung

Benjamin Schroth

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Florian Haupt
begonnen am:	18.06.2013
beendet am:	18.12.2013
CR-Nummer:	D.2.2, D.2.6

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Representational State Transfer (REST) definiert einen Architekturstil, der versucht, Latenzzeiten und Netzwerkkommunikation auf ein Minimum zu reduzieren und dabei die Unabhängigkeit und Skalierbarkeit der Komponenten zu maximieren. REST wurde von Roy T. Fielding in seiner Dissertation [Fie00] bereits im Jahr 2000 beschrieben. Er führte dabei den Erfolg des *World Wide Webs* auf bestimmte Eigenschaften der verwendeten Mechanismen und Protokolle (wie z. B. HTTP) zurück. Seit der Entstehung des *Cloud Computing-Paradigmas* gewinnt REST wieder mehr an Bedeutung. Die Verwendung einer generischen Schnittstelle wird notwendig, um einen möglichst hohen Grad an Interoperabilität und Ortsunabhängigkeit der verwendeten Dienste zu erreichen.

REST-basierte Anwendungen werden typischer- aber nicht notwendigerweise mit HTTP als Protokoll auf der Anwendungsschicht implementiert. HTTP wird von REST-Regeln in voller Breite ausgenutzt, während traditionelle *Web-Dienste*, die meist auf Simpe Object Access Protocol (SOAP) basieren, HTTP lediglich als Transportprotokoll nutzen. Sie bieten damit eher einen Protokollbaukasten, mit dem Entwickler ein anwendungsspezifisches Protokoll entwerfen können. Dies schränkt die Interoperabilität von SOAP-basierten Diensten ein. Im Gegensatz dazu fordert REST unter anderem eine uniforme Schnittstelle und macht damit die Entwicklung von generischen Diensten und Werkzeugen in der *Cloud* sehr viel einfacher. Daher bietet REST für Anforderungen von stark verteilten Systemen eine geeignete Antwort.

Große Unternehmen wie *Amazon* und *Google* setzen mittlerweile auf REST-Schnittstellen und immer mehr Dienste wie z. B. *Wiki Webs* sind von Hause aus in weiten Teilen REST-konform.

Abbildung 1.1 auf der nächsten Seite gibt einen Anhaltspunkt auf die Entwicklung des Interesses an REST über die letzten Jahre. Dabei wurden 5.100 Web-APIs verglichen, die im Februar 2012 bei der Seite *programmableweb*[1] registriert waren.

REST-basierte Anwendungen besitzen Konstrukte, die häufig in gleicher oder ähnlicher Weise realisiert werden. So werden große Objektlisten häufig mit der Möglichkeit zur Filterung und seitenweiser Navigation implementiert. Es existieren mehrere Rahmenwerke, wie beispielsweise auf Basis von JAX-RS, um die Entwicklung von REST-Anwendungen zu vereinfachen. Auch die Verwendung dieser Rahmenwerke bedeutet – bei Realisierung solcher Konstrukte – den immer gleichen oder ähnlichen Anwendungscode wiederholt zu programmieren. Dies stellt häufig eine unnötige Fehlerquelle dar.

Desweiteren definiert REST eine Menge an Regeln (Constraints), die eingehalten werden müssen, um eine Anwendung als RESTful bezeichnen zu können. In der Praxis werden diese Regeln oft nicht richtig verstanden, was zu ungenügenden Implementierungen führt.

[1] `http://www.programmableweb.com/`

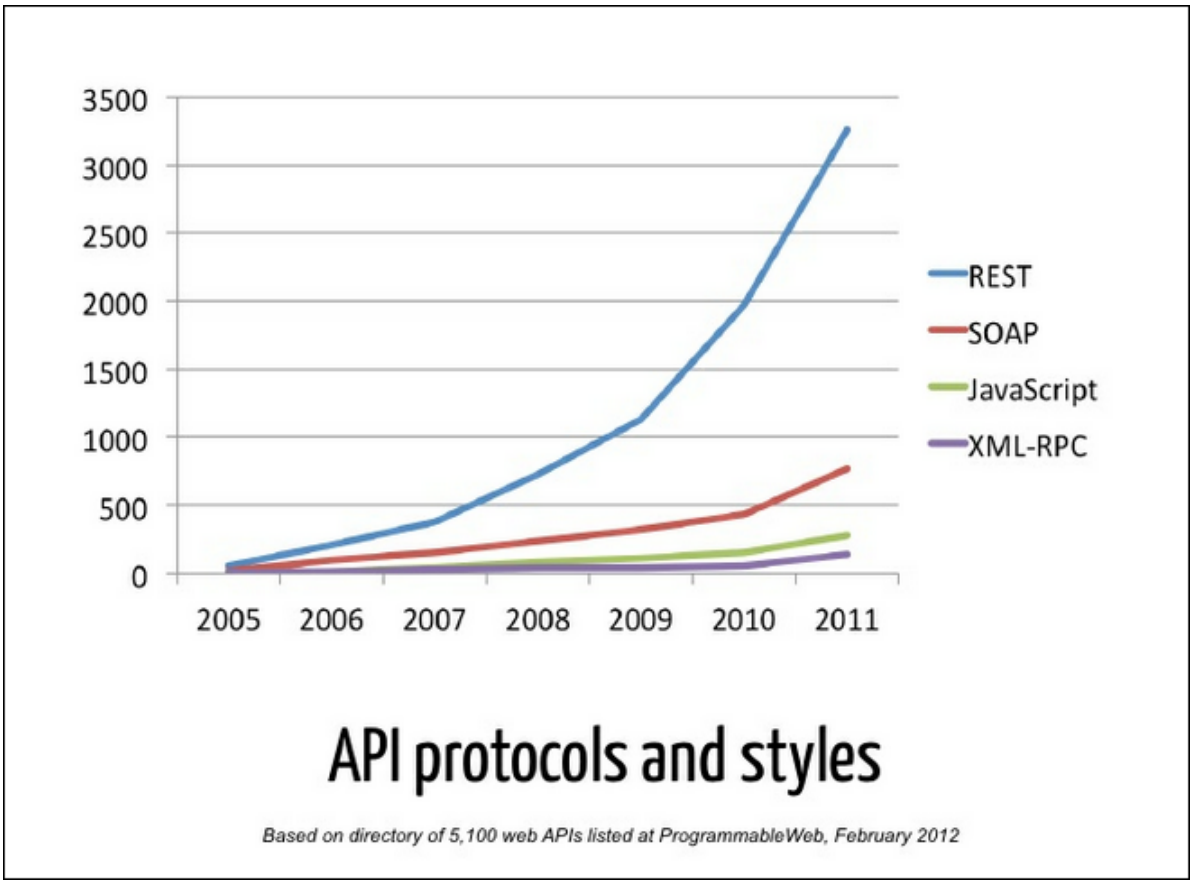

Abbildung 1.1 – Entwicklung des Anteils REST -APIs in den letzten Jahren [Mus12]

Das Konzept der modellgetriebenen Softwareentwicklung erlaubt es, repetitive und damit fehleranfällige Implementierungsarbeit zu vermeiden. Hierbei dienen Modelle als Ausgangspunkt zur Generierung von Anwendungscode. Dabei ist die Generierung nicht auf eine Zielplattform beschränkt, sondern kann, bei richtiger Trennung von plattformunabhängigen und plattformabhängigen Elementen, jegliche Zielplattform bedienen. Die Modelle bilden dabei durch eine höhere Abstraktionsebene gleichzeitig eine verständliche Dokumentation der Anwendung. Außerdem vermeidet der Einsatz von – korrekt implementierten – Generatoren Fehler bei der Umsetzung der von REST definierten Regeln. Der Entwickler muss also nicht unbedingt alle Regeln von REST verstehen, da der Generator den gewünschten Programmcode automatisiert erstellt. Dieser Ansatz eliminiert zuverlässig Fehlerquellen, welche durch unzureichendes Verständnis syntaktischer, semantischer oder inhaltlicher Regeln entstehen können.

Diese Arbeit untersucht, wie die Entwicklung REST-basierter Anwendungen durch den Einsatz der Prinzipien der modellgetriebenen Softwareentwicklung verbessert werden kann. Hierzu wurden die nötigen Modelle und Modell-Transformationen identifiziert und auf der so gewonnen Basis ein Prototyp in Form einer *Eclipse-Erweiterung* implementiert. Um die Plattformunabhängigkeit der Modelle der oberen Schichten zu zeigen, wurden Generatoren für zwei Zielarchitekturen entwickelt.

1.1 Motivation

Um die repetitiven Arbeiten bei der Implementierung von REST-Anwendungen zu vermeiden, soll eine Modellstruktur entwickelt werden, aus der eine REST-Applikation generiert werden kann. Dadurch sollen möglichst viele Fehlerquellen bei der Entwicklung solcher Applikationen vermieden werden.

Die angestrebte Modellstruktur sollte flexibel genug sein, um für verschiedene Zielplattformen benötigte Artefakte generieren zu können.

Der Fokus dieser Arbeit liegt allerdings nicht darauf, vollständige Modelle zu erarbeiten und somit alle Aspekte von REST modellieren zu können, sondern vielmehr darauf, eine konsistente Methodik zu entwickeln, mit der dieses Ziel langfristig zu erreichen ist.

1.2 Aufbau der Arbeit

Die Arbeit gliedert sich in 7 Kapitel:

Kapitel 2 – Grundlagen führt im folgenden in die Grundlagen von REST und die Prinzipien der modellgetriebenen Softwareentwicklung ein.

Kapitel 4 – Modelle zeigt, welche Modelle notwendig sind und welche Transformationen entworfen wurden.

Kapitel 6 – Implementierung stellt den entwickelten Prototypen vor.

Kapitel 7 – Zusammenfassung und Ausblick fasst die vorliegende Arbeit zusammen und gibt einen Ausblick auf anstehende Arbeiten und mögliche Entwicklungen.

2 Grundlagen

2.1 REST

REST ist ein Architekturstil für verteilte (Hypermedia) Systeme in der Softwareentwicklung im *World Wide Web*, der als Abstraktion aus dem *HTTP Object Model* von Roy Thomas Fielding im Rahmen seiner Disseration *Architectural Styles and the Design of Network-based Software Architectures* aus dem Jahr 2000 entwickelt wurde.

Unter einem Architekturstil versteht man eine benannte Menge von Bestimmungen auf architektonische Elemente, deren Einhaltung eine Menge von gewünschten Eigenschaften der Architektur zur Folge haben ([Fie00]).

Wird REST auf ein Minimum reduziert, dann ergeben sich nach [Til11] fünf Grundprinzipien:

- Resourcen mit eindeutiger Identifikation
- Verknüpfungen
- Standardmethoden
- Unterschiedliche Repräsentation
- Statuslose Kommunikation

Die Schlüsselabstraktion in REST bilden Resourcen. Eine Resource ist eine konzeptionelle Zuordnung einer Menge von Entitäten. Dabei ist lediglich die Semantik der definierten Resource statisch, während die zugeordneten Daten dynamisch sind. Die eindeutige Identifikation geschieht über sogenannte URI-Verknüpfungen, sie funktionieren dabei wie im HTTP beschrieben, anwendungs- und serverübergreifend.

Hypermedia as the Engine of Application State (HATEOAS) oder *hypertext-driven* ist eine Anwendung, die den Client durch die möglichen Interaktionen führt. Dafür wird in der angeforderten Repräsentation einer Resource zusätzlich ein Element eingefügt, welches die URI der möglichen Folgeinteraktionen enthält. In HTML bietet sich hierfür z. B. ein *link*-Element an. Dies ermöglicht eine lose Kopplung; die Schnittstelle lässt sich somit anpassen und verändern, ohne dass dabei bereits festgelegte Einstellungen am Client adjustiert werden müssen.

Leonard Richardson entwickelte das nach ihm benannte Richardson REST Maturity Modell (REST-Reifegradmodell) als vierstufigen Maßstab, der angibt, wie stark ein Dienst auf REST basiert. Tabelle 2.1 auf der nächsten Seite zeigt einzelnen Reifegradstufen, wie sie nach [Fow10] definiert sind.

Level	Beschreibung
0	• Verwendet HTTP als Transportprotokoll • Verwendet einzelne URI als Dienstendpunkt • Verwendet einzelne HTTP-Methode (POST / GET) • Beispiele SOAP oder XMLRPC
1	• Verwendet HTTP als Transportprotkoll • Verwendet Resourcen mit unterschiedlichen URIs • Verwendet einzelne HTTP-Methode (POST / GET)
2	• Verwendet HTTP • Verwendet Resourcen mit unterschiedlichen URIs • Verwendet mehrere HTTP-Methoden
3	• Verwendet HTTP • Verwendet Resourcen mit unterschiedlichen URIs • Verwendet HTTP Methoden • Verwendet HATEOAS

Tabelle 2.1 – Definition der einzelnen Level des REST Maturity Models nach [Fow10]

Roy Thomas Fielding schreibt in einem Blogeintrag ([Fie08]), dass die Stufe 3 des REST Maturity Modells eine notwendige Bedingung für REST ist. Anwendungen müssen also zwingend HATEOAS verwenden, um vollumfänglich als REST-basierend zu gelten.

Die Bemühungen, einfache Rahmenbedingungen zu schaffen, um REST-basierte Dienste zu klassifizieren, legen nahe, dass die Landschaft der Dienste, die sich RESTful nennen einer genaueren Analyse bedürfen. Oft werden fälschlicherweise einfache RPC-APIs eben als REST-Implementierung bezeichnet. Es bedarf also noch weiterer Anstrengungen, um den Entwurf und die Implementierung von tatsächlich REST-basierenden Anwendungen zu vereinfachen.

Ein Beispiel für eine API, die fälschlicherweise als REST-API bezeichnet wird findet sich in der Web-Anwendung *Twitter* ([Twi13]). Zwar definiert sie durch verschiedene URI unterschiedene Resourcen, allerdings verwendet sie ausschließlich die HTTP-Methoden GET und POST und Methodenaufrufe wie *Löschen* (*destroy*) ebenfalls über die URI ab. Sie liegt also auf Level 1 des RMM.

2.2 Prinzipien der modellgetriebenen Softwarenetzwicklung

Viele betriebswirtschaftliche Prozesse laufen heute mindestens teilweise durch Software automatisiert ab. Dadurch entwickelte sich Software schnell zu einem wichtigen Faktor in Unternehmen. Fortschrittliche und fehlerfreie Software bedeutet daher einen großen Wettbewerbsvorteil. Mit der zunehmenden Automatisierung steigt auch die Komplexität moderner Softwaresysteme, wodurch die Entwicklung dieser Systeme kostenintensiver und fehleranfälliger wird [GPR06].

Eine Antwort auf die steigende Komplexität von Software findet sich im Ansatz der *modellgetriebenen Softwarentwicklung*. Hauptziele dieses Ansatzes sind (vgl. [GPR06]):

- Portierbarkeit
- Erleichterung der Bildung von Schnittstellen
- Reduzierung der Entwicklungszeit
- Abstraktion der Komplexität
- Qualitätssicherung
- Erhöhung der Robustheit gegen häufige Anforderungsänderung

Nach [SVEH07] geht es in der modellgetriebenen Softwareentwicklung unter anderem darum, sich möglichst nicht selbst zu wiederholen. Dies nennt man das 'Don't repeat yourself' (DRY)-Prinzip. Dies verringert das Fehlerpotential sowie die Wiederverwendbarkeit bei der Softwareentwicklung und reduziert dadurch die Entwicklungszeit, ohne das benötigte Budget zu erhöhen.

Es wird im Zusammenhang mit modellgetriebener Softwareentwicklung von folgenden Modellen gesprochen (vgl. [GPR06]):

Das Computational Independend Model (CIM) beschreibt eine von der Implementierung völlig losgelöste Sicht auf das System. Das Modell wird im Vokabular seiner Domäne beschrieben und betont die Anforderungen an das System und seine Umwelt.

Das Plattform Independent Model (PIM) dient hierbei als formale Beschreibung der Struktur und Funktionalität eines Systems. Es ist vollständig unabhängig von Implementierungsdetails und abstrahiert die damit von der zugrundeliegenden Plattform. Plattform beschreibt in diesem Zusammenhang die Ausführungsumgebung der modellierten Anwendung.

Das Plattform Specific Model (PSM) beschreibt eine Anreicherung der Informationen aus dem PIM mit plattform-abhängigen Details. Sofern das PSM alle Informationen zur Konstruktion und zum Betrieb eines Systems enthält, wird es als Implementierung bezeichnet. In diesem Zusammenhang spricht man dann von *ausführbaren Modellen*.

Ein Kernkonzept der modellgetriebenen Softwareentwicklung ist die *Transformation*. Dies bezeichnet die Umwandlung eines, in ein anderes Modell desselben Systems. Dazu zählen auch *Modell-zu-Text-Transformationen*, wie sie zur Codegenerierung verwendet werden. Eine Transformation wird durch Abbildungsregeln (engl. *Mapping Rules*) definiert.

Abbildung 2.1 zeigt den Entwicklungsprozess auf Basis von PIM und PSM.

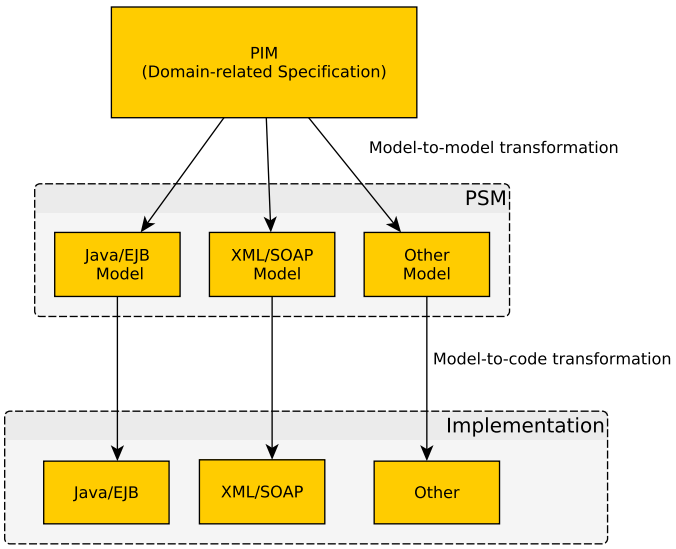

Abbildung 2.1 – Grundlegendes Prinzip der modellgetriebenen Softwareentwicklung

3 Verwandte Arbeiten

In [Sch11] zeigt Sylvia Schreier zwei getrennte Modelle für das Verhalten der Anwendung und deren Struktur auf. Sie stellt damit einen Bezug zwischen Resourcen-Definitionen (Typ, Name, mögliche Operationen) und Metainformationen (URI-Struktur) in einem Modell her und verwaltet alle Verhaltensdefinitionen in einem Weiteren. Die Abbildungen 3.1 und 3.2 auf der nächsten Seite zeigen Beispiele für eine Instanz dieser beiden Modelle.

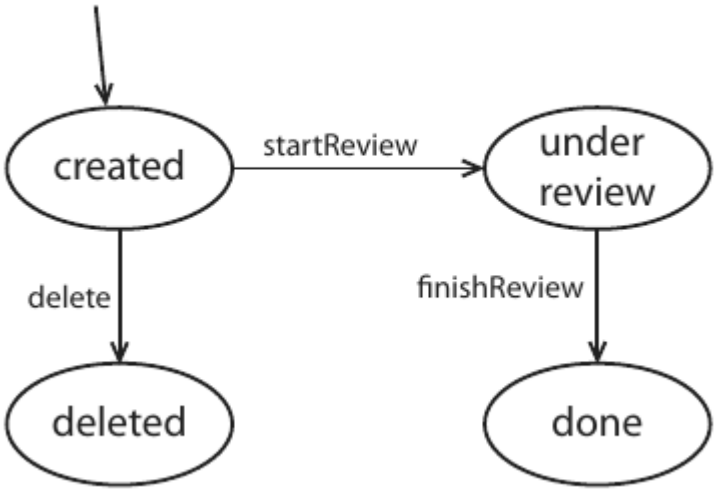

Abbildung 3.1 – Beispiel eines Verhaltensmodells nach [Sch11]

Ivan Žužak, Ivan Budiselić und Goran Delac haben in [ZBD11] auf Basis des Modells eines nichtdeterministisch endlichen Zustandsautomats mit leerem Zustandsübergang (ϵ-Übergang) ein formales Modell für auf REST-basierende Systeme entwickelt. Dabei konzentrierten sie sich auf die Verhaltensmodellierung der Anwendung. Die Abbildung zwischen den Komponenten eines REST-basierten Systems auf die Elemente des NEA und die Verteilung zwischen Client und Server ist in Abbildung 3.3 auf Seite 11 dargestellt.

Abbildung 3.4 auf Seite 11 zeigt die Beispielapplikation, die in [ZBD11] verwendet wurde, um die erarbeiteten Konzepte zu illustrieren. Die Anwendung soll auf einer Detailseite, die über die Einstiegsseite erreichbar ist, Temperatur und Wetter (bewölkt oder sonnig) ausgeben. Den resultierenden NEA zeigt Abbildung 3.5 auf Seite 12.

Nicolas Karrer und Marco Sonderegger haben in ihrer Bachelorarbeit [KS12] mit dem Titel *MDSD & REST* zeitglich zum Entstehen dieser Arbeit eine ähnliche Aufgabenstellung bearbeitet. Sie erstellten ein einzelnes Meta-Modell, um damit eine REST-Schnittstelle mehrheit-

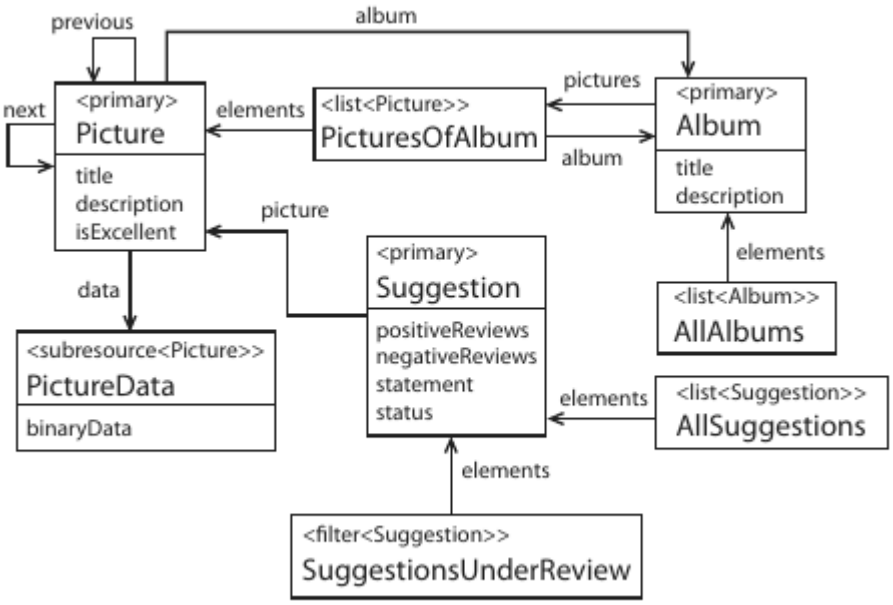

Abbildung 3.2 – Beispiel eines Strukturmodells nach [Sch11] illustriert UML-ähnlicher Notation

lich graphisch zu entwerfen. Sie verwendeten bei der Entwicklung die Eclipse-Erweiterung *Actifsource*[1], das auf *Model Driven Design* ausgelegt ist. Ein zentraler Punkt ihrer Arbeit war die Untersuchung von REST-Anwendungen, die auf Level 3 des REST *Maturity Models* und somit auf HATEOAS aufbauen. Dies zeigt sich in dem erstellten Modell. Es ist möglich, für eine Resource mehrere URI zu erstellen und diese dann für die Verknüpfungen zwischen Resourcen zu verwenden. Dabei besteht ein URI aus mehreren *URIParts*. Die Ansätze in der Arbeit von Nicolas Karrer und Marco Sonderegger unterscheiden sich deutlich von denen, die in dieser Arbeit verfolgt wurden. So gibt es keine Trennung der modellierten Teile der Anwendung. Resourcen werden zusammen mit ihren Daten modelliert.

[1] `http://www.actifsource.com/`

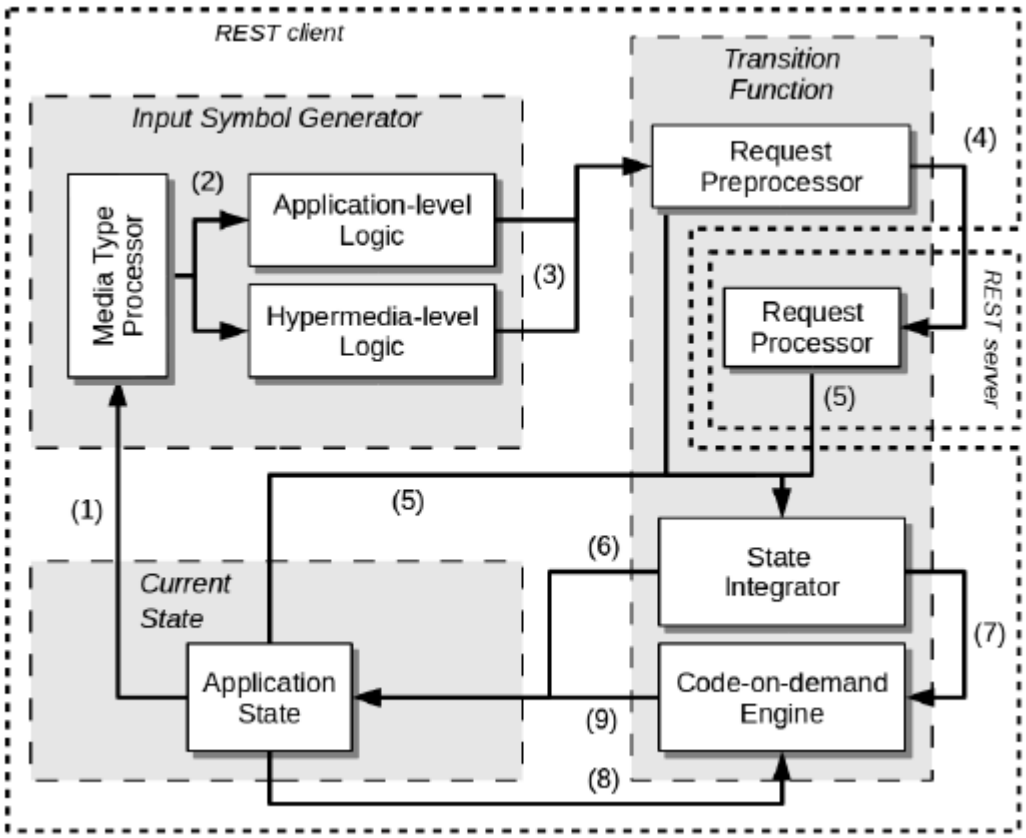

Abbildung 3.3 – Abbildung der Komponenten eines REST-basierten Systems auf das NEA-Modell [ZBD11]

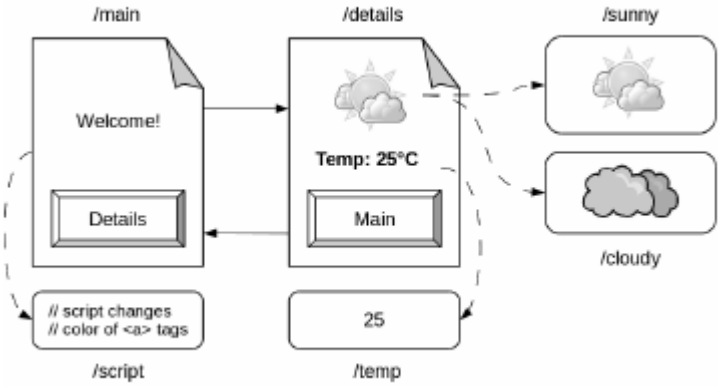

Abbildung 3.4 – Beispiel Web-Anwendung, die in [ZBD11] modelliert wurde

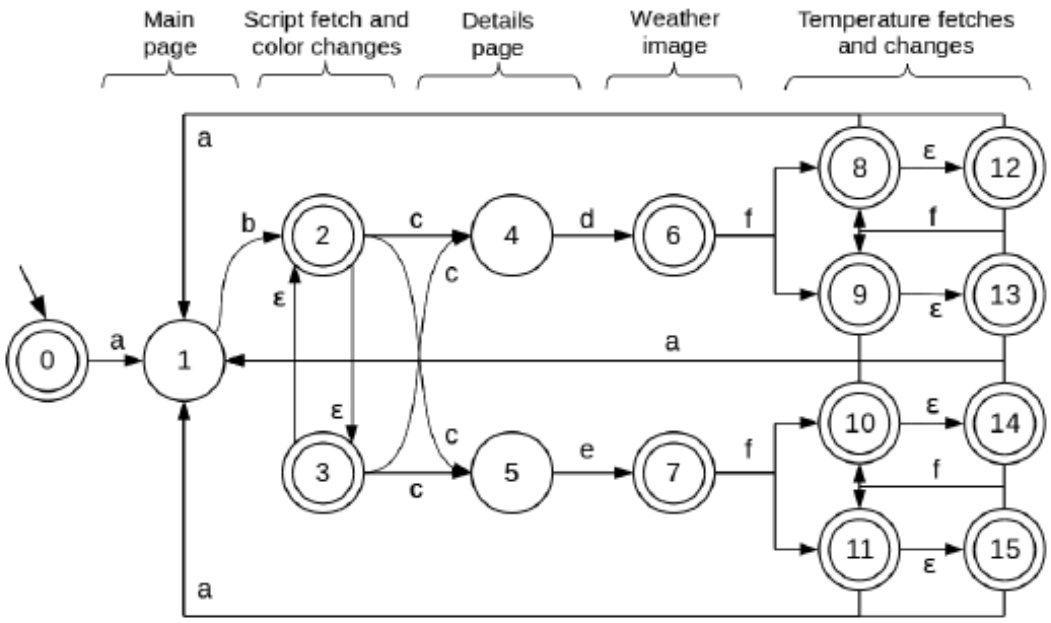

Abbildung 3.5 – Resultierender Zustandsautomat für die Beispielapplikation aus [ZBD11]

4 Modelle

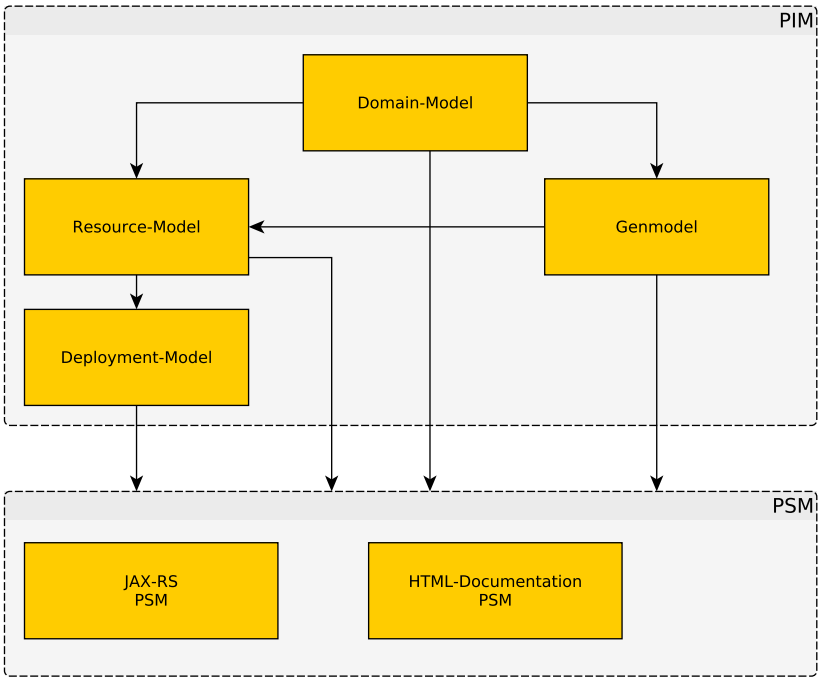

Abbildung 4.1 – Verwendete Modellstruktur

Abbildung 4.1 zeigt die verwendeten Modelle. Die Applikation wird von drei plattformunabhängigen Modellen definiert.

Im Einzelnen sind diese:

- Domänen-Modell
- Resourcen-Modell
- Deployment-Modell

des Weiteren existiert ein Zwischenmodell, das zur Steuerung der Transformation zwischen Domänen-Modell und Resourcen-Modell dient.

Zur Demonstration der Möglichkeiten dieser Modellstruktur wurden zwei mögliche Zielplattformen definiert:

- JAX-RS-Applikation

- HTML-Dokumentation der modellierten REST-Applikation

Für diese beiden Zielplattformen wurden entsprechende PSM entworfen. Im folgenden Kapitel werden die einzelnen Metamodelle genauer beschrieben.

4.1 Domänen-Metamodell

Dieses Modell beschreibt den Zuständigkeitsbereich, der der Applikation zugrundeliegenden Domäne. Abbildung 4.2 zeigt das Diagramm des dazugehörigen Metamodells. Im Fall des Prototypen handelt es sich hierbei um ein stark vereinfachtes objektorientiertes Metamodell, denkbar wäre allerdings auch eine andere Art, die Domäne ausreichend zu beschreiben. Ein Beispiel hierfür wäre ein Entity-Relationship-Diagramm (ER-Diagramm).

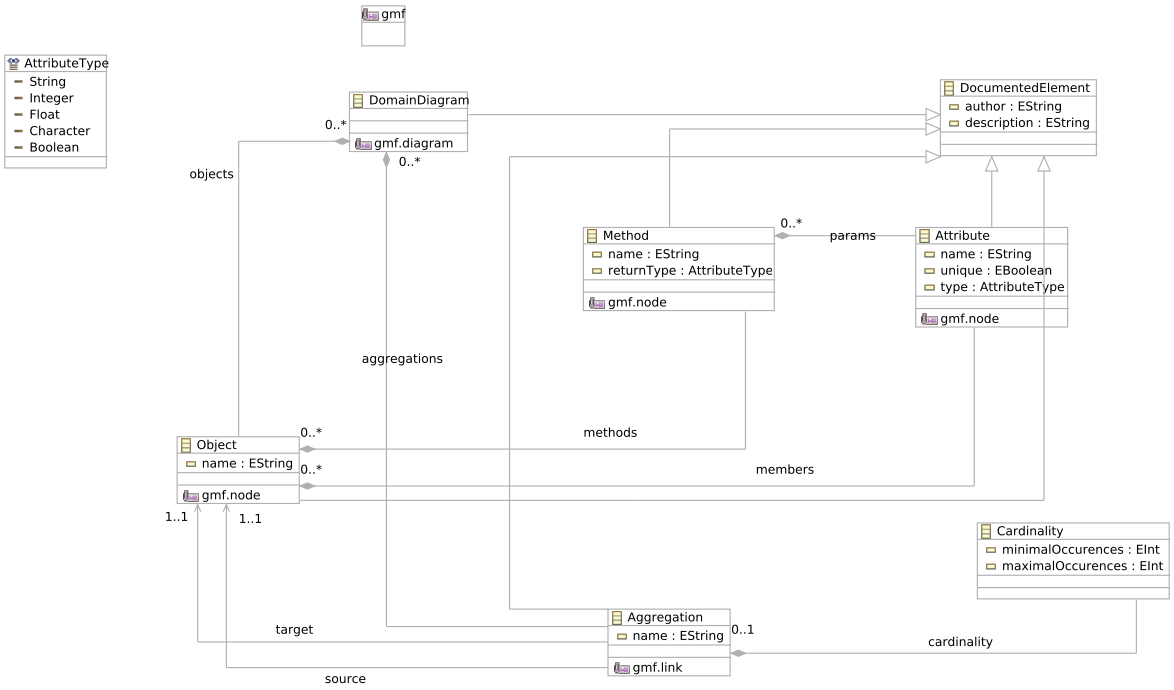

Abbildung 4.2 – Ecore-Diagramm des Domänen-Metamodells

Das verwendete Metamodell beschreibt lediglich *Objekte* mit ihren Attributen und Methoden. Zusätzlich stellt das Modell eine Art der Beziehung zwischen diesen Klassen bereit, die Aggregation.

Attribute wurden mit einem Datentyp (Standarddatentypen aus Java) und einem Namen definiert. Zusätzlich kann einem *Attribut* mit der Eigenschaft *unique* versehen werden um ein das Domänen-Objekt eindeutig identifizierendes Attribut zu kennzeichen.

Methoden wurden mit einem Rückgabetyp (Standarddatentyp aus Java), einem Namen und Parametern modelliert. Die Definition dieser Parameter entspricht der Definition der Attribute.

Zur Demonstration des Konzepts reicht dieses einfache Modell aus. Eine Erweiterung dieses Metamodells oder der Wechsel auf ein anderes Modell zieht Anpassungen in den ETL-Skripten für die Transformation sowie in der Erzeugung des Zwischenmodells nach sich.

4.1.1 Zwischenmetamodell

Dieses Metamodell dient als externe Modellmarkierung (vgl. [SVEH07]) des Domänen-Modells zur Transformation in das Resourcen-Modell. Eine externe Modellmarkierung bietet Vorteile gegenüber der direkten Kennzeichnung im Quellmodell, da das Modell nicht durch Informationen, die nicht zu den eigentlich modellierten Objekten gehören, *verschmutzt* wird. Diese Modellmarkierungen sind die einzigen, die in der gesamten Modellstruktur verwendet werden. Alle weiteren Umwandlungen (Transformationen) benötigen keine zusätzlichen Parameter.

Das Zwischenmodell legt fest, welche Objekte des Domänenmodells durch welche Resourcen in der späteren REST-Anwendung abgebildet wird. Darüber hinaus wird die Art der Resource sowie einige zusätzliche Parameter für die Transformation beschrieben. In diesem Modell ist es möglich, eine Strategie für die Generierung der CRUD (Create, Read, Update, Delete)-Methoden einer Resource festzulegen.

Dabei ermöglicht es das Modell, eine Resource per POST-Anfrage gefüllt oder unausgefüllt zu erstellen. Da POST eine HTTP-Request-Methode mit Seiteneffekten ist, wäre die bessere Strategie, unausgefüllte (leere) Resourcen zu erzeugen, um somit einen mehrfachen Aufruf der Methode zu ermöglichen, ohne dabei größere Aufräumarbeiten auf Seiten des Servers leisten zu müssen.

Dabei muss der Server lediglich die erstellten leeren Resourcen entfernen. Das Zwischenmodell bietet dennoch die Strategie an, Resourcen gefüllt oder gänzlich ohne die Verwendung von CRUD-Methoden zu erstellen. Hierbei besteht die Möglichkeit eine geeignete Strategie für jede der vorhandenen Resourcen festzulegen. Die Abbildungen 4.3 auf der nächsten Seite und 4.4 auf Seite 17 stellen die Abläufe der beiden POST-Anfragen als Sequenzdiagramme dar.

Für die Aggregationen ist es üblich, zwischen der aggregierten und der aggregierenden Resource eine Listen-Resource zwischenzuschalten. Abbildung 4.5 auf Seite 17 illustriert diese Abbildung zwischen Domänen- und Resourcen-Modell. Dieser Vorgang kann im Zwischenmodell für einzelne oder alle Aggregationen abgeschaltet werden. Wird die Abbildung abgeschalten, wird das aggregierende Domänen-Objekt als Listenresource abgebildet.

Abbildung 4.6 auf Seite 18 zeigt ein Diagramm des zugehörigen *Ecore-Metamodells*.

4.2 Resourcen-Metamodell

Das Resourcen-Modell bildet die zur Verfügung gestellten Resourcen und ihre Verbindungen untereinander ab. Für den Prototyp unterscheidet das Modell lediglich Wurzelresourcen

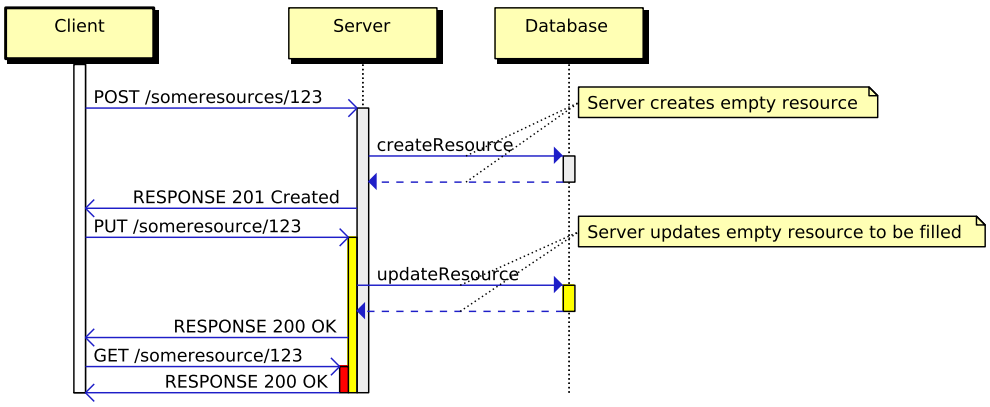

Abbildung 4.3 – Darstellung einer POST-Anfrage, die leere Resourcen erstellt

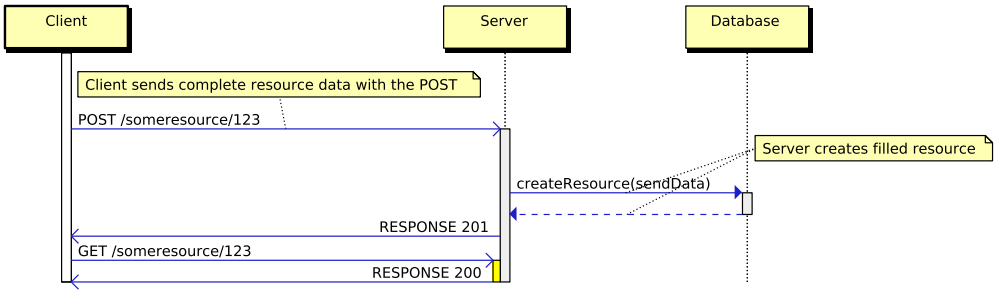

Abbildung 4.4 – Darstellung einer POST-Anfrage, die gefüllte Resourcen erstellt

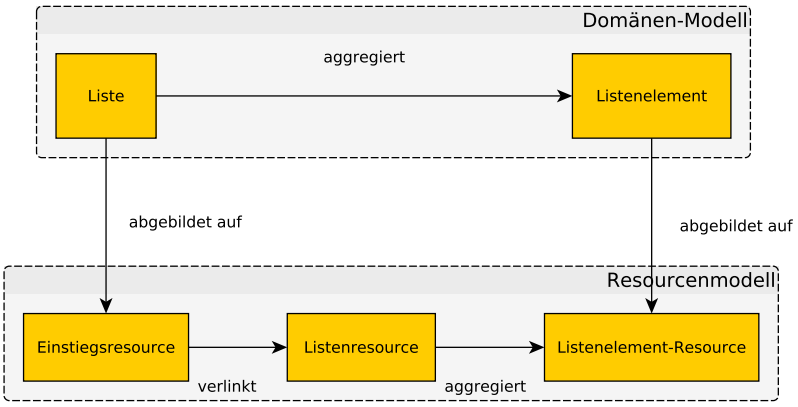

Abbildung 4.5 – Abbildung einer Aggregation vom Domänen- zum Resourcen-Modell

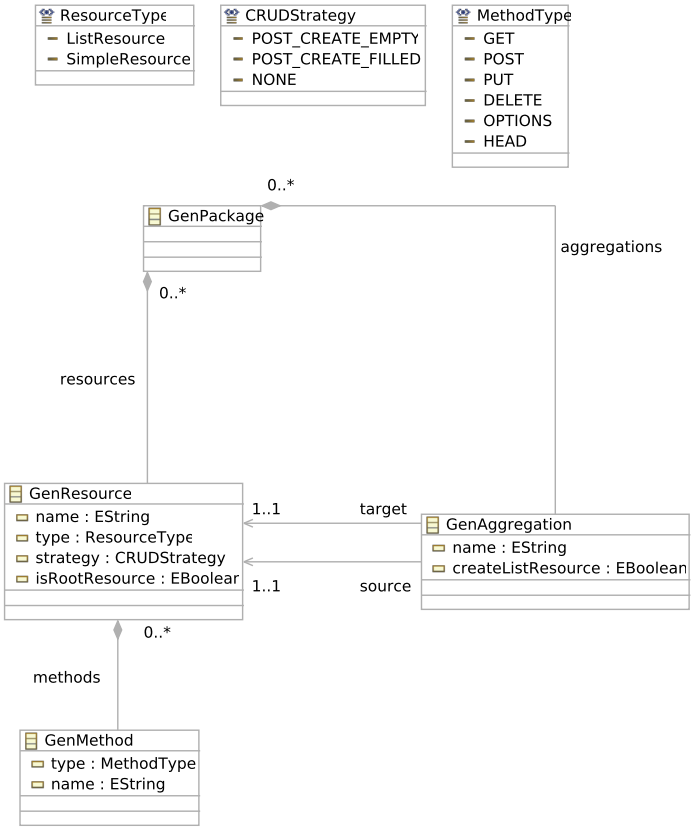

Abbildung 4.6 – Ecore-Diagramm des Zwischenmodells

– die als Einstiegspunkte in die Applikation dienen – einfache Resourcen und Listenresourcen. REST selbst unterscheidet keine Resourcentypen, rein logisch ist es möglich durchaus noch weitere Resourcentypen zu unterscheiden, wie beispielsweise Filterresourcen (vgl. dazu [Til11]).

Abbildung 4.7 zeigt das Diagramm des zugehörigen *Ecore-Metamodells*.

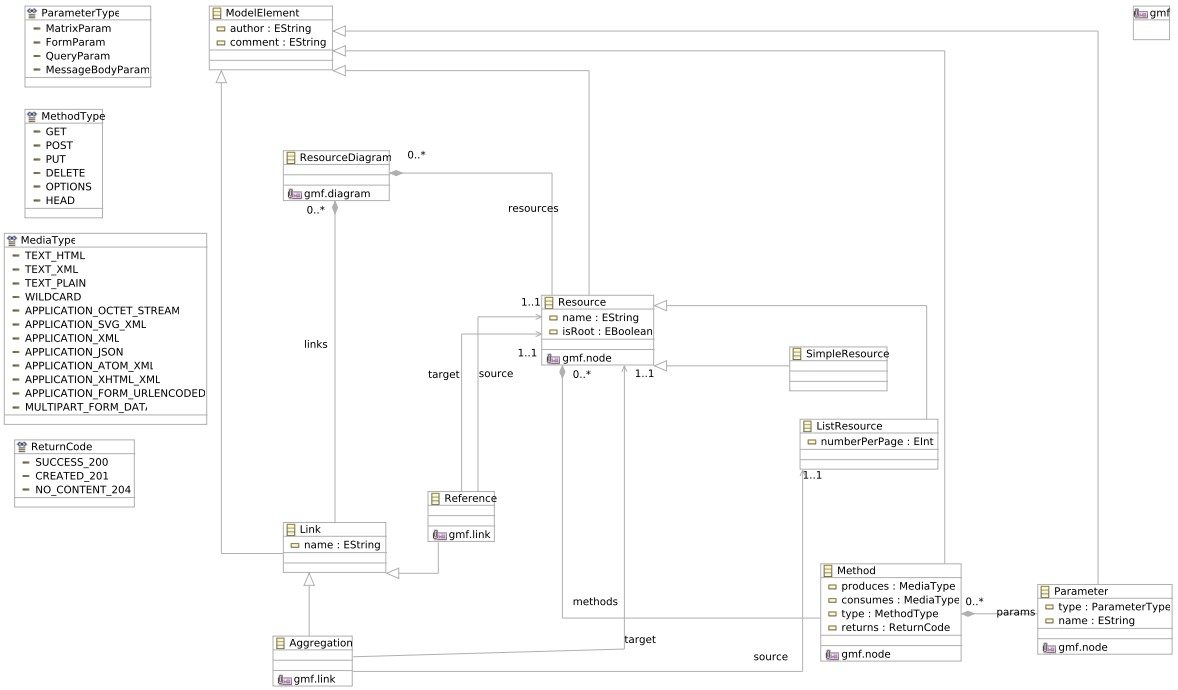

Abbildung 4.7 – Ecore-Diagramm des Resourcen-Metamodells

ResourceDiagram dient im Modell als Wurzelelement, das alle anderen Elemente enthält. Eine *Resource* ist als Objekt definiert, dass Methoden aggregiert und zusätzlich zu einem Namen die Dokumentationsattribute *author* und *comment* besitzt. Listenresourcen definieren zusätzlich das Attribut *numberPerPage*, das angibt, wieviele Elemente eine Seite der Liste zurückgeben soll.

Methoden besitzen das Attribut *Type*, das das HTTP-Verb angibt, das diese Methode verwendet. Außerdem ist für jede Methode die Medientypen definiert, die die Methode produzieren beziehungsweise konsumieren kann. Die möglichen Medientypen sind Standardmedientypen wie HTML JSON oder XML.

Zu jeder Methode lassen sich Parameter definieren. Diese sind durch ihren Namen und einen Typ modelliert. Der Typ eines Parameters beschreibt dessen Art der Übergabe. Im Prototypen reichen die Parameter-Typen *QueryParam*, für einen Parameter, der als Teil des URL übergeben wird, und *MessageBodyParam*, der im Nachrichtenkörper übergeben wird.

Jede Methode besitzt einen Rückgabewert (HTTP-Return Codes). Diese wurden ebenfalls als Teil einer Methode modelliert. Mögliche Rückgabewerte, die im Prototypen definiert wurden, sind:

Code	Beschreibung
`200 - OK`	Anfrage erfolgreich. Antwort wird im Nachrichtenkörper zurückgegeben
`201 - Created`	Resource wurde erstellt.
`204 - No Content`	Anfrage erfolgreich. Kein weiterer Inhalt im Nachrichtenkörper

Tabelle 4.1 – Definierte HTTP-Rückgabewerte

4.3 Deployment-Metamodell

Das Deployment-Modell definiert eine Abbildung der Resourcen und Links auf Pfade in der Applikation. Hier wird also die URI-Struktur der Applikation festgelegt.

Es ist möglich Platzhalter für die Attribute des zugehörigen Domänen-Objekts zu verwenden. Die Syntax der Platzhalter lehnt sich dabei an die Syntax der in JAX-RS verwendeten *PathParam*-Annotationen an. Es wird also ein Name in geschweiften Klammern eingefasst. Der Name beschreibt dabei das Attribut des entsprechenden Domänen-Objekts, das für diesen Platzhalter eingessetzt werden soll.

Zum Beispiel verweist der Platzhalter {*title*} auf das Attribut *title* im korrespondierenden Domänenobjekt. Zusätzlich verweist der allgemeine Platzhalter {*id*} auf das Attribut im Domänenobjekt, für das die *unique*-Eigenschaft auf *true* gesetzt ist.

4.4 Plattformspezifische Modelle

Nachdem alle nötigen plattformunabhängigen Modelle erfasst wurden, sind damit alle nötigen abstrakten Informationen über die REST-Applikation festgelegt. Somit endet der plattformunabhängige Teil des Modellierungsprozesses und ein auf die Zielplattform abgestimmtes PSM (Plattform Specific Model) muss ausgewählt werden. Um die Unabhängigkeit des Plattform Independent Model zum PSM zu zeigen, wurden für den Prototypen zwei verschiedene Zielplattformen ausgewählt:

- JAX-RS als Rahmenwerk zur Erstellung von REST-Anwendungen
- Eine HTML-Dokumentation der Anwendung

Im Folgenden werden die beiden Zielplattformen und die korrespondierenden Modelle genauer beschrieben.

4.4.1 JAX-RS als Ziel der Generierung

JAX-RS (Java API for RESTful Web Services) bezeichnet die Spezifikation einer Programmierschnittstelle zur Implementierung von REST-Anwendungen im Rahmen von Webservices. Hierfür gibt es mehrere Implementierungen:

- Jersey
- RESTlet
- JBoss RESTEasy
- Apache Wink
- Apache CXF (JAX-RS-Erweiterung)

Für den Prototypen wird die Referenzimplementierung *Jersey* verwendet, allerdings ist das PSM nicht direkt auf *Jersey* angepasst. *Jersey* umfasst mehr als den reinen JAX-RS-Standard, wie beispielsweise eine API zur programmatischen Erstellung von Resourcen zur Laufzeit. Diese Möglichkeiten werden von dem verwendeten PSM allerdings nicht unterstützt.

Im Wesentlichen aggregiert das Modell die Informationen der anderen Modelle und reichert sie mit verschiedenen Einstellungen für die spätere Code-Generierung an. Die zusätzlichen Informationen, die in diesem Modell definiert werden sind:

- Basis-URL der Anwendung
- Namen der Java-Pakete für die Resourcen- und Domänenklassen
- Verwendete Java Version
- Gewünschte Serialisierung der Domänenobjekte (für den Prototypen die Wahl zwischen JAXB oder benutzerdefiniert)

Abbildung 4.8 auf der nächsten Seite zeigt das Metamodell des PSMs als Ecore-Diagramm.

4.4.2 HTML-Dokumentation als Ziel der Generierung

Als zweites Ziel der Generierung soll eine HTML-Dokumentation dienen. Sie beschreibt die vorhandenen Domänenklassen und Resourcen – ihre Verknüpfungen und Abhängigkeiten – sowie die Struktur der Anwendung. Das PIM zu diesem Generierungsziel ist einfach gehalten. Es bietet dem Nutzer die Möglichkeit einzelne Domänenklassen oder Resourcen aus der Dokumentation zu entfernen oder hinzuzufügen und/oder die Dokumentation für die gesamte Domäne oder alle Resourcen global ab- bzw. anzuschalten.

Die Modellierung im Domänen- und Resourcenmodell mithilfe eines abstrakten, übergeordneten Elements – von dem alle anderen Modellelemente erben – lässt hierbei Platz für weitere Dokumentationselemente.

Abbildung 4.9 auf der nächsten Seite zeigt das entstandene Metamodell als Ecore-Diagramm.

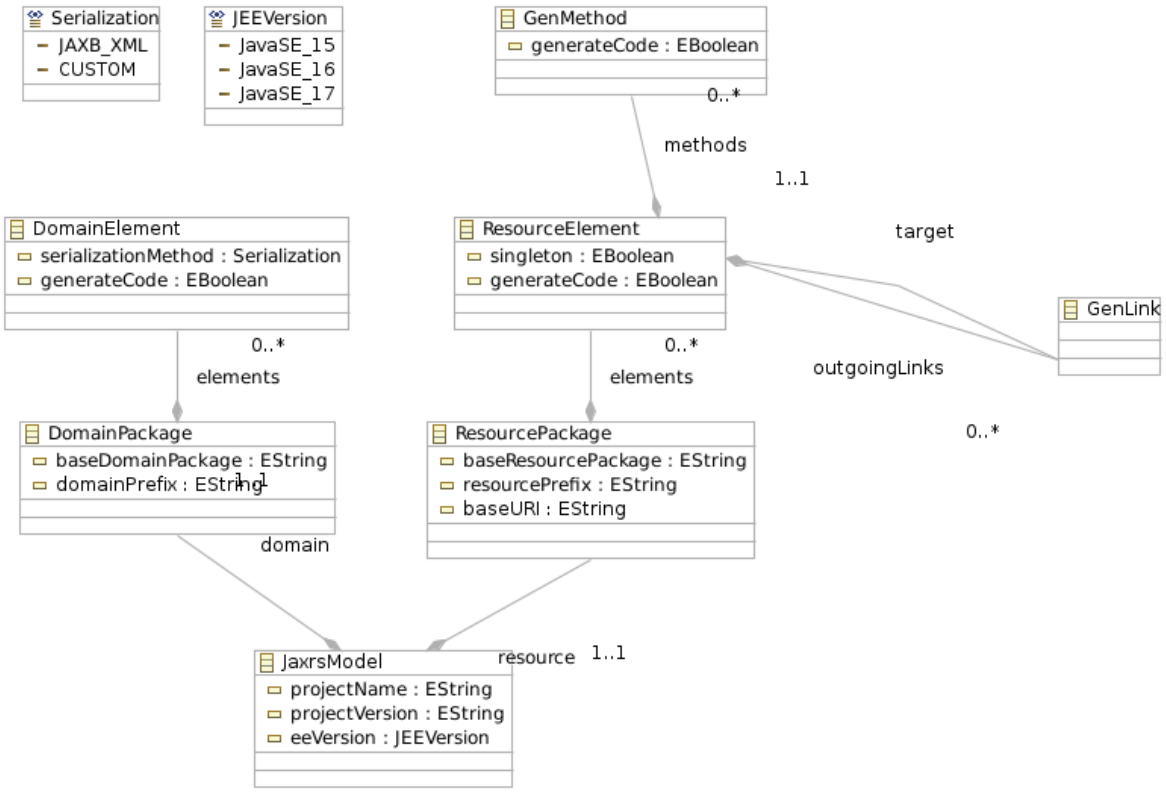

Abbildung 4.8 – Ecore-Diagramm des JAX-RS-Modells

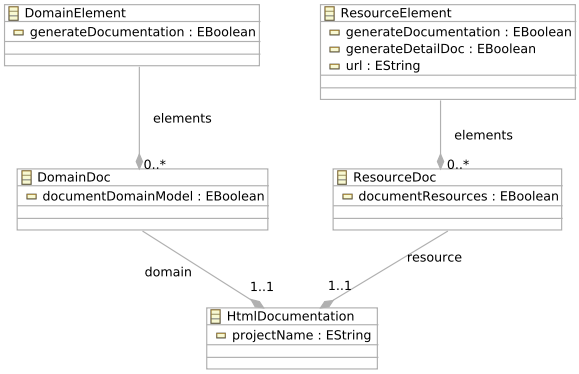

Abbildung 4.9 – Ecore-Diagramm des PSMs zur HTML-Dokumentation

5 Verwendete Technologien

5.1 Epsilon

Epsilon bezeichnet eine Menge an Sprachen und Werkzeugen um Codegenerierung, Modell-zu-Modell-Transformation, Modellvalidierung, Modellvergleich, Migration und Refactoring mit EMF und anderen Modelltypen zu vereinfachen. Den Kern von *Epsilon* bildet die Epsilon Object Language (EOL), eine imperative, modellorientierte Sprache, die Modellabfragen mit prozeduraler Ausführung kombiniert. Auf dieser Basis wurden einige spezifische Sprachen definiert um die einzelnen Aufgaben innerhalb der modellgetriebenen Softwareentwicklung zu unterstützen.

Abbildung 5.1 zeigt die Architektur von Epsilon, wie in der Dokumentation des auf der Webseite des Projekts dargestellt [Eps13].

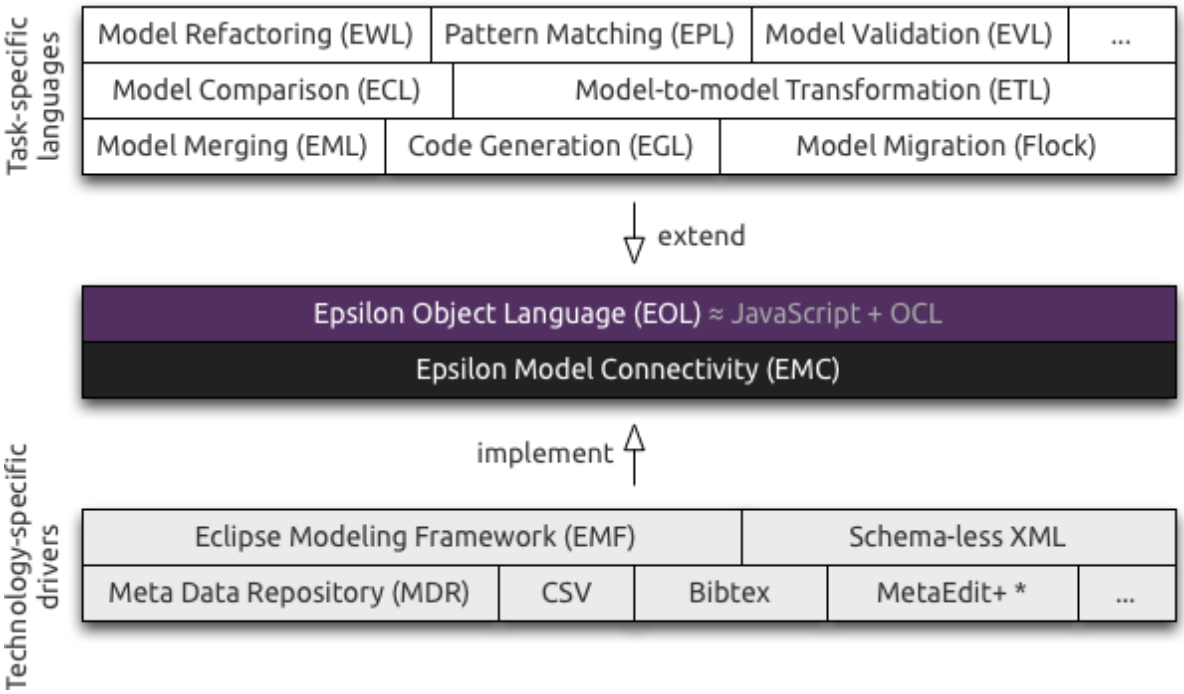

Abbildung 5.1 – Epsilon-Architektur [Eps13]

Nachstehend eine kurze Beschreibung der Komponenten von *Epsilon*, die in dieser Arbeit verwendet wurden.

5.1.1 EuGENia

EuGENia generiert automatisch die benötigten GMF-Modelle auf Basis eines entsprechend annotierten *Ecore-Metamodells*. Dadurch bleibt dem Benutzer die Komplexität von GMF verborgen, eine etwaige Einstiegshürde wird dadurch vermieden. Änderungen am ursprünglichen EMF-Modell sowie an den generierten GMF-Modellen können dabei über Menüeinträge mit den Generierungsmodellen von EMF und GMF (genmodel und gmfgen) synchronisiert werden. Dadurch gehen manuelle Änderungen an den Einstellungen der Generierungsmodelle nicht verloren.

EuGENia bietet zudem Mechanismen, um größere Anpassungen am generierten Editor zu spezifizieren und die erzeugten GMF-Modelle automatisch bei jeder Generierung anzupassen. Dies geschieht mithilfe von zwei EOL-Transformationen. Diese werden in Dateien mit speziellen Namen (*ECore2GMF.eol* und *FixGMFGen.eol*) gespeichert und von EuGENia automatisch nach der eigentlichen Generierung der GMF-Modelle ausgeführt. Die Transformation *FixGMFGen.eol* wird hierbei für eine Anpassung des Synchronisationsprozesses des GMF-Generierungsmodells verwendet und wird auch hier als letztes ausgeführt.

Dies macht EuGENia nicht nur für schnelles *prototyping* einsetzbar, sondern ermöglicht eine Verwendung bis zur Produktreife.

5.1.2 Emfatic

Emfatic bietet die Möglichkeit, *Ecore-Modelle* in einer Java-ähnlichen Syntax textuell zu beschreiben. Darüberhinaus ist es möglich durch spezielle Annotationen die Generierung eines zugehörigen GMF-Editors zu steuern und so schon hier die benötigten Knoten und Verknüpfungen für den grafischen Editor festzulegen. Aus der *Emfatic-Quelldatei* kann per Kontextmenüeintrag ein entsprechendes *Ecore-Modell* erzeugt werden. EuGENia kann auch direkt auf *Emfatic-Quelldateien* angewandt werden, um ohne Zwischenschritte den GMF- Editor zu generieren.

Listing 5.1 zeigt als Beispiel die Deklaration eines einfachen Dateisystemmodells. Es enthält ebenfalls die Annotationen, die EuGENia zur Generierung der GMF-Modelle verwendet. Abbildung 5.2 auf Seite 26 stellt den Arbeitsablauf bei der Arbeit mit der Kombination von *Emfatic* und *EuGENia* da.

```
1  namespace(uri="filesystem", prefix="filesystem")
2  package filesystem;
3
4  @gmf.diagram
5  class Filesystem {
6      val Drive[*] drives;
7      val Sync[*] syncs;
8  }
9
10 class Drive extends Folder {
11 }
12
13 class Folder extends File {
14     @gmf.compartment
15     val File[*] contents;
```

```
16 }
17
18 class Shortcut extends File {
19     @gmf.link(target.decoration="arrow", style="dash")
20     ref File target;
21 }
22
23 @gmf.link(source="source", target="target", style="dot", width="2")
24 class Sync {
25     ref File source;
26     ref File target;
27 }
28
29 @gmf.node(label = "name")
30 class File {
31     attr String name;
32 }
```

Listing 5.1 – Emfatic-Beschreibung eines einfachen Dateisystem-Modells

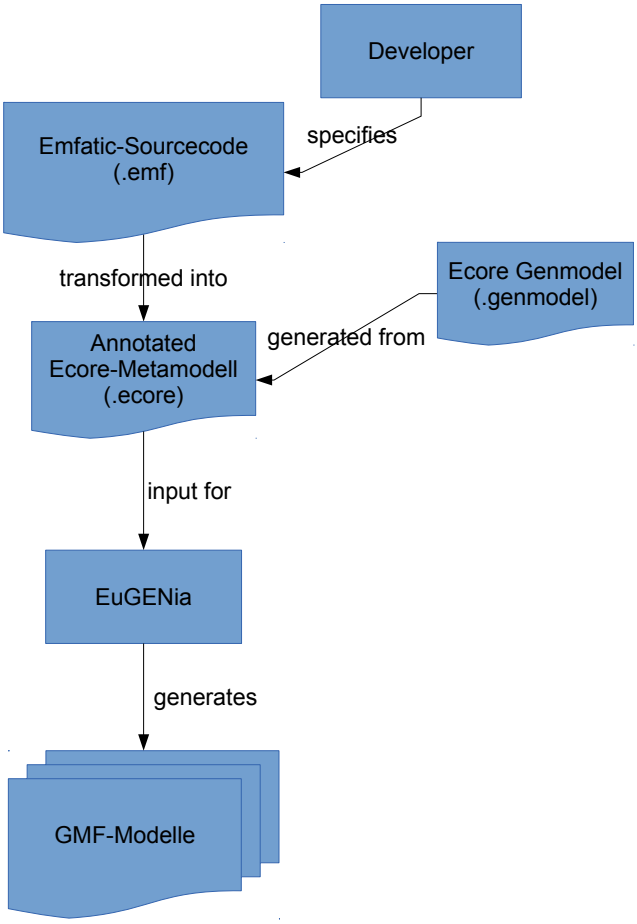

Abbildung 5.2 – Arbeitsablauf zur Generierung von GMF-Editoren mithilfe von *Emfatic*

5.1.3 EVL – Epsilon Validation Language

Die Epsilon Validation Language bietet eine Möglichkeit Metamodellinstanzen sowohl im generierten *Ecore-Modelleditor*, als auch in einem zugehörigen GMF-basierten Editor zu validieren. Hierfür werden Regeln in einer EOL-basierten Syntax beschrieben und mittels *Eclipse Extension Points* in die bestehenden Editoren eingebunden. Dabei wird der *Eclipse Extension Point org.eclipse.epsilon.evl.emf.validation* verwendet, um Metamodell und EVL-Skript zu verknüpfen, während per *org.eclipse.ui.ide.markerResolution* verwendet wird, um die Eintragungen in den *Marker-View* von Eclipse abzubilden.

Listing 5.2 zeigt als Beispiel eine Validierung des in Abschnitt 5.1.2 auf Seite 24 als Beispiel verwendeten Dateisystem-Modells.

```
1 context File {
```

```
2     constraint HasName {
3         check : self.name.isDefined()
4         message : 'Unnamed ' + self.eClass().name + ' not allowed'
5     }
6   }
7   context Folder {
8     critique NameStartsWithCapital {
9         guard : self.satisfies('HasName')
10        check : self.name.firstToUpperCase() = self.name
11        message : 'Folder ' + self.name +
12            ' should start with an upper-case letter'
13        fix {
14            title : 'Rename to ' + self.name.firstToUpperCase()
15            do {
16                self.name := self.name.firstToUpperCase();
17            }
18        }
19    }
20  }
21  context Sync {
22    constraint MustLinkSame {
23        check : self.source.eClass() = self.target.eClass()
24        message : 'Cannot synchronize a ' + self.source.eClass().name
25            + ' with a ' + self.target.eClass().name
26        fix {
27            title : 'Synchronize with another ' +
28                self.source.eClass().name
29            do {
30                var target := UserInput.choose('Select target',
31                    Model.getAllOfType(self.source.eClass().name));
32                if (target.isDefined()) self.target := target;
33            }
34        }
35    }
36  }
```

Listing 5.2 – EVL-Skript zur Validierung eines einfachen Dateisystem-Modells

Es wird für das Objekt *File* die Existenz eines Namens überprüft, für *Folder*-Objekte muss dieser Name zusätzlich mit einem Großbuchstaben beginnen. Außerdem dürfen Verknüpfungen nur Objekte der selben Klasse (also z. B. nur *File*-Objekte mit *File*-Objekten) verbinden.

5.1.4 ETL – Epsilon Transformation Language

Die Epsilon Transform Language bietet die Möglichkeit, Modell-zu-Modell-Transformationen in einer EOL-basierten Syntax, anzugeben. Dabei werden Regeln für die Transformation zwischen einem Element des Quellmodells in ein Element des Zielmodells beschrieben. Dabei können beliebig viele Modelle als Eingabe und Ausgabe verwendet werden. Grundsätzlich wird unter drei verschiedene Arten von Modelltransformationssprachen unterschieden:

- Deklarative Sprachen
- Imperative Sprachen

- Hybride Sprachen

Alle Ansätze haben verschiedene Vor- und Nachteile. So ist die Abstraktionsebene von regelbasierten, deklarativen Sprachen hoch und sie eignet sich nur für Szenarien, in denen Quell- und Zielmodell sich in ihrer Struktur ähneln, wodurch nur minderkomplexe Abbildungen unterstützt werden.

Imperative Sprachen haben den Nachteil die Abstraktionsebene sehr niedrig zu halten. Die Entwicklung mit solchen Sprachen ist weitaus aufwendiger, bieten jedoch auf der anderen Seite die Möglichkeit, mit komplexeren Abbildungen umgehen zu können. Um die Nachteile bei der Softwareentwicklung damit zu beheben, wurden hybride Sprachen (wie ATL und QVT) entwickelt. Hybride Sprachen vereinen die Vorteile der regelbasierten Ausführung mit den Möglichkeiten der imperativen Sprachen und bieten so einen Kompromiss aus hoher Abstraktion und komplexen Möglichkeiten.

ETL gehört hierbei zu der Gruppe der hybriden Sprachen und bietet somit auch die Möglichkeit, komplexe Transformationen mit regelbasierter Ausführung umzusetzen.

Zu diesem Zweck werden ETL-Skripte entweder über entsprechende Epsilon-API-Aufrufe oder über mitgelieferte *Ant-Targets* ausgeführt. Vor dem Aufruf des Scripts wird Quell- und Zielmodell übergeben. Der Prototyp verwendet die erste Möglichkeit und enthält zu diesem Zweck eine Java-Klasse zur Ausführung und Initialisierung der Skripte.

5.2 JET – Java Emitter Templates

Mithilfe von Java Emitter Templates lassen sich in JSP-ähnlicher *Syntax Templates* zur Textgenerierung definieren. Dies macht JET sehr gut zur Codegenerierung nutzbar. Der Prototyp verwendet JET zur Generierung aller textuellen Komponenten der beiden Zielplattformen.

5.3 GMF – Graphical Modeling Framework

Mit GMF wurde eine Abstraktionsebene über dem regulär in Eclipse verwendeten GEF (Graphical Editing Framework) erstellt. Anstelle der manuellen Implementierung auf Basis der von GEF zur Verfügung gestellten Schnittstellen und abstrakten Klassen, kann von GMF mithilfe von mehreren Modellen der gesamte Editor definiert und anschließend generiert werden. Dies beschleunigt die Entwicklung von grafischen Editoren. GMF definiert zu diesem Zweck vier verschiedene Modelle:

Graphical Definition beschreibt das Erscheinungsbild der verwendeten Knoten und Verknüpfungen

Tooling Definition definiert Einträge in der Werkzeugleiste des Editors

Mapping Model erklärt die Verbindung zwischen dem Modell, den Einträgen in der Werkzeugleiste und den definierten Knoten und Verknüpfungen

Generation Model führt andere Modelle zur Generierung zusammen

Eine Alternative zu GMF wird in *Graphiti* gesehen, einem weiteren Rahmenwerk, das als Abstraktionsebene über GEF und grundlegenden Java-APIs (Draw2D zur Zeichnung von 2D-Objekten auf dem Bildschirm) entwickelt wurde. Anders als bei GMF wird bei *Graphiti* direkt mit der von *Graphiti* zur Verfügung gestellten Java-API gearbeitet, was einen höheren Implementierungsaufwand bedeutet. *Graphiti*-Diagrammeditoren werden plattformunabhängig entwickelt, so dass auf eine andere Rendering-Engine gewechselt werden kann. Allerdings unterstützt *Graphiti* derzeit ausschließlich Eclipse.

Im Gegensatz zur Alternative *Graphiti* muss bei der Verwendung von GMF keine Komponente selbst implementiert werden. Durch die Unterstützung von *EuGENia* besteht die Möglichkeit die erzielten Ergebnisses nachträglich anzupassen und zu verändern. In dieser Arbeit wurden die generierten Editoren nur wenig modifiziert, deshalb war die Entwicklung mit GMF deutlich schneller und einfacher, ohne hierbei Flexibilität für zukünftige Arbeiten einzubüßen. Mit GMF ist es darüberhinaus möglich, die Daten der Modellinstanz von den Daten, die zur grafischen Repräsentation genutzt werden, auf zwei verschiedene Dateien zu verteilen und so auch mit externen Werkzeugen sehr viel besser zu verarbeiten, als es in *Graphiti* möglich ist.

5.4 EMF – Eclipse Modeling Framework

EMF bietet dem Nutzer die Möglichkeit ein Metamodell im XMI-Format zu definieren oder aus externen Quellen (wie beispielsweise einer XSD-Datei) generieren zu lassen. Zu diesem Zweck wird vor der Generierung das Quellmodell (also entweder die erstellte Ecore-Datei oder die externe Quelle) auf ein Generierungs-Modell von EMF abgebildet. Dieses bietet Einstellungsmöglichkeiten zur Struktur des generierten Codes wie Paket- und Projektnamen oder die Dateiendung der Modellinstanzen.

Auf dieser Basis kann EMF den nötigen Code zur Implementierung des Metamodells generieren. Dazu zählt eine einfache API zur Verwaltung von Modellinstanzen, wie der Speicherung als XML-Datei, das Laden aus einer bestehenden Datei und die Manipulation einer geladenen Instanz.

Innerhalb dieser Arbeit wurde EMF zur Generierung sämtlicher Modelle der Modellstruktur genutzt.

6 Implementierung

6.1 Architektur

Der Prototyp gliedert sich im Wesentlichen in vier Komponenten:

Modelle alle durch EMF/GMF-generierten Modell- und Editorquelltexte

Transformator zuständig für alle Modell-zu-Modell-Transformationen

Validator verantwortlich für alle Modell-Validierungen

Generator leitend für die Generierung der Zielartefakte

Der Zusammenhang dieser Komponenten wird in Abbildung 6.1 gezeigt.

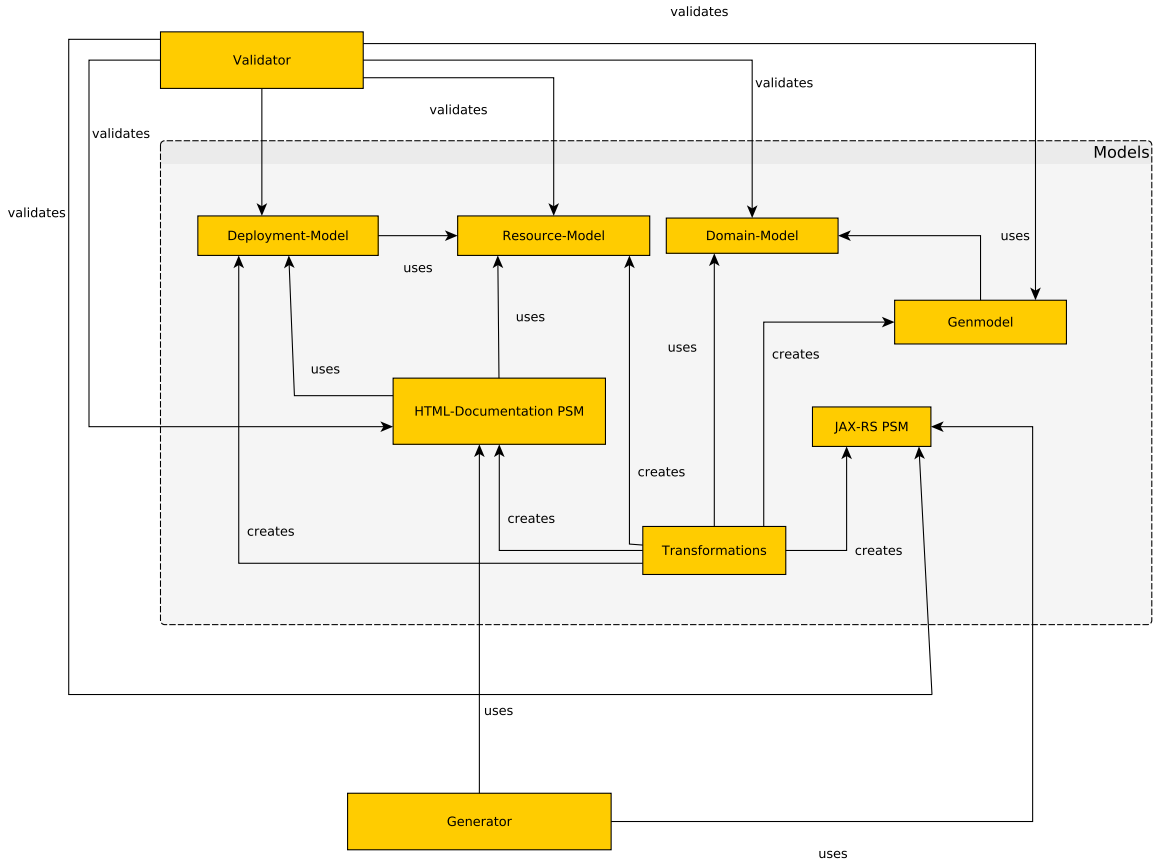

Abbildung 6.1 – Architektur des Prototypen

Nachfolgend schildert eine ausführliche Beschreibung dieser Komponenten die Funktionsweise.

6.1.1 Modelle

Unter Modelle sind alle verwendeten EMF-Modelle zu verstehen, wie sie im Abschnit 4 auf Seite 13 beschrieben wurden. Für alle Modelle wurde zusätzlich EMF Baumeditoren generiert. Für das Domänen- und Resourcenmodell wurde zusätzlich ein GMF-Editor zur einfacheren Analyse und Bearbeitung generiert, da für diese Modelle eine grafische Repräsentation aufgrund ihrer Struktur als sinnvoll erkannt wurde. Abbildung 6.3 auf der nächsten Seite zeigt den generierten GMF-Editor für das Domänen-Modell.

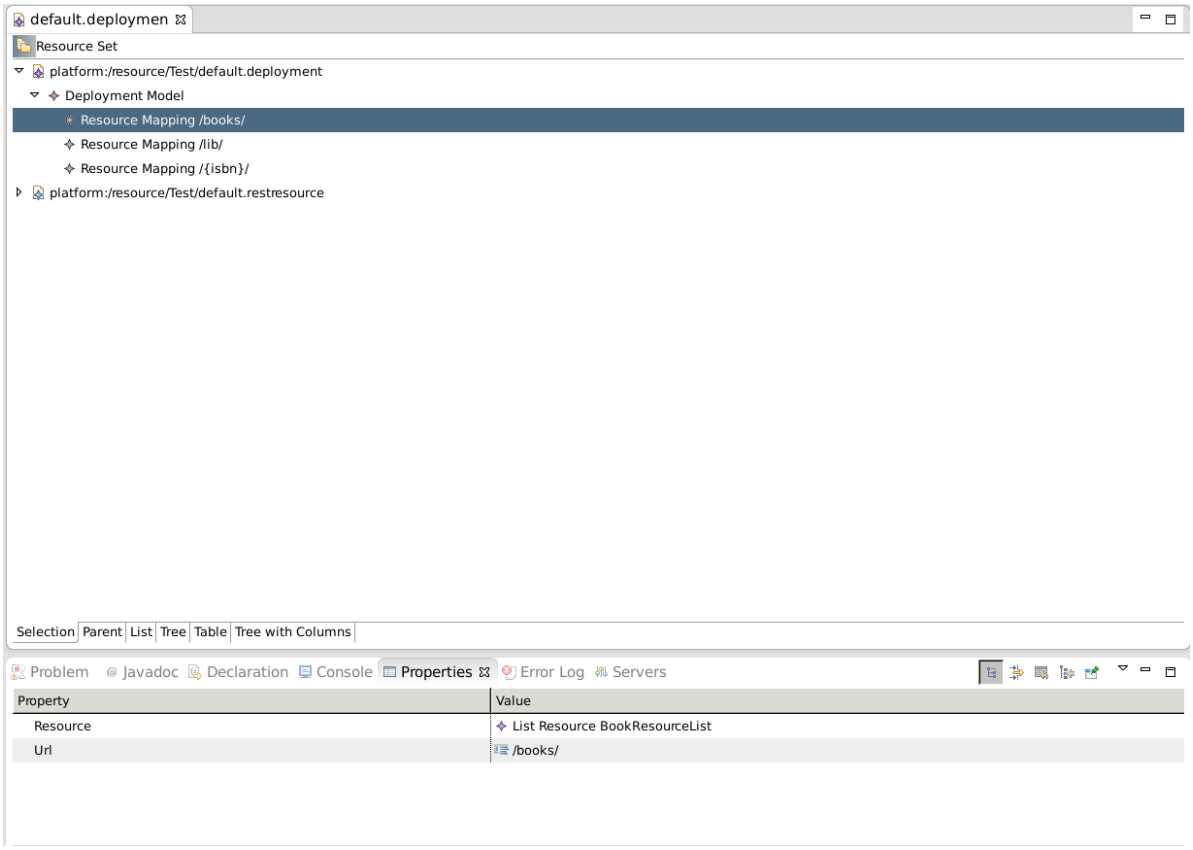

Abbildung 6.2 – EMF Baumeditor des Deployment-Modells

Zur Erzeugung der GMF-Editoren wurden die GMF-spezifischen Annotationen in *Emfatic* verwendet. Abbildung 6.2 zeigt als Beispiel des, mit EMF generierten, Baumeditors, den Editor des Deployment-Modells.

Alle Modelle bestehen aus mindestens 3 Eclipse-Plugins: *Modell*, *Edit*, *Editor*. Die eigentliche Modellimplementierung ist zusammen mit seiner Ecore-Definition in einem einzelnen Plugin untergebracht. Es enthält alle notwendigen Klassen und Funktionen zum Laden und Speichern des Modells im XML-Format, sowie zur Modifikation und Haltung im Speicher.

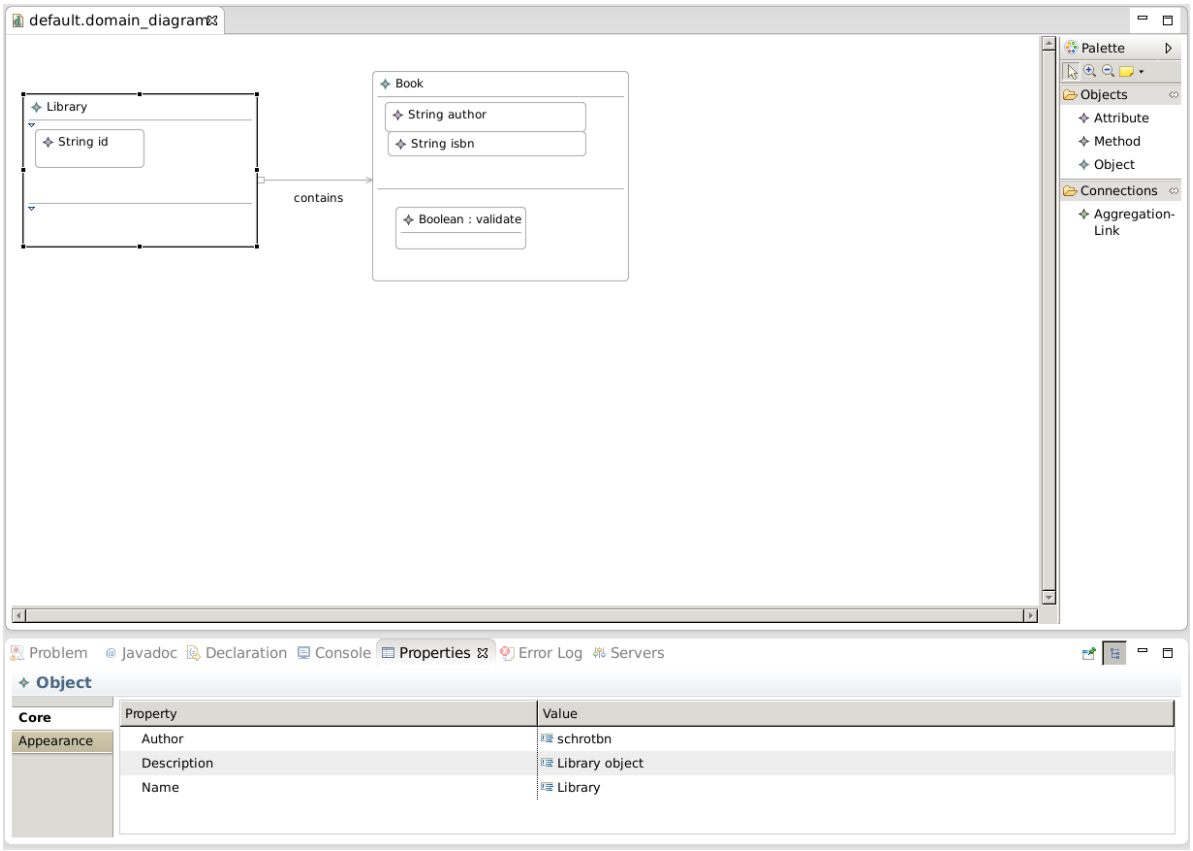

Abbildung 6.3 – GMF-Editor des Domänen-Modells

Das zusätzliche *Edit*-Plugin enthält sogenannte *ItemProvider* für jedes modellierte Objekt. Diese werden von EMF intern verwendet, um Beschriftungen und Inhalt von Element in einer Modellinstanz abrufen zu können. Diese Klassen informieren automatisch die korrespondierenden Anzeigen, wenn sich Beschriftungen oder Inhalt ändert.

Das *Editor*-Plugin macht Gebrauch von dieser Funktionalität, indem es einen Eclipse-Editor bereit stellt, der die korrespondierenden Informationen aus einer geladenen Modellinstanz über die *ItemProvider* einholt und setzt. Ergebnis ist der oben erwähnte Baumeditor.

6.1.2 Transformator

Diese Komponente dient zur Durchführung von Transformationen zwischen den einzelnen Modellen. Sie enthält neben den eigentlichen Transformationsskripten in ETL den Code zur Ausführung derselben. Zusätzlich sind hier über *Eclipse Extension Points* entsprechende Kontextmenüeinträge zum Aufruf der verschiednenen Transformationen definiert und die dazugehörigen Klassen implementiert. Außerdem stellt die Komponente einen Assistenten (Wizard) zur Erstellung des Zwischenmodells zur Steuerung der Transformation zwischen Domänen- und Resourcen-Modell zur Verfügung. Um während der Ausführung der ETL-Transformationen Eingaben des Benutzers entgegennehmen zu können, wurde die von *Epsilon* zur Verfügung gestellte *IUserInput*-Schnittstelle implementiert.

Im Verlauf der Arbeit wurden vier ETL-Skripte definiert:

- Transformation vom Zwischen- zu Resourcen-Modell
- Erstellung von Depoyment-Modell aus Resourcen-Modell
- Erstellung des HTML-Dokumentation-PSM aus verschiedenen Modellen
- Erstellung des JAX-RS-PSM aus verschiedenen Modellen

Eine Java-Klasse kümmert sich dabei um das Hinzufügen von neuen Modellen sowie das Laden und Ausführen von ETL-Skripten. Die konkrete Klasse zur Ausführung von ETL-Skripten beerbt dabei eine abstrakte Klasse. Die abstrakte Klasse kann erweitert werden um weitere *Epsilon*- Sprachen zur Laufzeit interpetieren zu können.

Die einzelnen Handler-Klassen für die Menüeinträge übernehmen dabei die nötige Benutzerinteraktion zur Auswahl der Zielmodelldatei nebst entsprechender Parameterisierung der Transformation.

Die Erstellung des Zwischenmodells zur Transformation von Domänen- nach Resourcen-Modell besitzt kein ETL-Skript, sondern verwendet die EMF-API zur Erstellung des Modells nach Angaben des Benutzers in den zugehörigen Assistenten (Wizard). Dies erlaubt eine schönere Führung des Nutzers durch die Optionen der Transformation, als es mit dieser Implementierung, der *IUserInput*-Schnittstelle möglich gewesen wäre.

6.1.3 Validator

Dient zur Überprüfung (Validierung) der einzelnen Modelle. Die Umsetzung erfolgt mittels EVL-Skripten. Die Prüfung der Modelle beschränkt sich auf einfache Regeln, um das Prinzip der Validierung zu verdeutlichen. Im Domänen-Modell wird überprüft, ob jedes Domänen-Objekt ein identifizierendes Attribut besitzt.

Die Validierung im Resourcen-Modell bedeutet, festzustellen, ob jede HTTP-Methode in einer Resource nur einmal verwendet wird und die eingestellten Rückgabewerte (HTTP Return Codes) zu den Methoden passen. So ist der Rückgabewert *204 - No Content* für eine GET-Methode in der implementierten Validerung nicht zugelassen, da GET im allgemeinen Fall immer etwas zurückliefern sollte. Technisch ist ein solcher Rückgabewert der GET-Methode in Ordnung, verhindert allerdings die Verwendung von HATEOAS, da der Client hier keine Verknüpfungen zu weiterführenden Aktionen mehr erhält. Es existieren Situationen, in denen eine GET-Methode nichts zurückliefern soll, diese sind allerdings sehr speziell. So kann eine Resource als Semaphore verwendet werden, wenn der Rückgabewert *204* oder *404 - Not found* genügt.

Im Deployment-Modell wird lediglich überpüft, ob allen übernommenen Resourcen eine URI zugeordnet worden ist.

In den beiden PSM werden alle für die Generierung notwendigen Felder, wie die verwendeten Projekt- und Paketnamen, auf valide Werte überprüft.

Abbildung 6.4 auf der nächsten Seite zeigt ein nicht valides Resourcen-Modell im zugehörigen GMF-Editor. Gefundene Fehler werden im Editor selbst markiert und in die Liste der Markierungen (*Markers*-View) in Eclipse geschrieben.

Alle in dieser Arbeit definierten Validierungsregeln sind als Zwangsbedingungen (*constraints*) definiert und damit verpflichtend einzuhalten. In EVL ist es allerdings auch möglich einzelne Regeln als nicht verpflichtend zu markieren (*critique*). Dies ist eine Möglichkeit zukünftig den Benutzer auf *best practices* hinzuweisen, die aber nicht verpflichtend einzuhalten sind. Bei der Validierung solcher Regeln wird ledigich eine *Warnung* und kein *Fehler* erzeugt.

6.1.4 Generator

Der Generator dient zur Erstellung der benötigen Artefakte für die gewählte Zielplattform. Alle Artefakte besitzen (mindestens) eine zugehörige JET-Datei. Dabei werden die Daten des zugehörigen Objekts aus dem entsprechenden PSM an die durch JET generierte Klasse übergeben und der ausgegebene Text über die Eclipse-API in entsprechende Dateien geschrieben. Um die Templates möglichst einfach zu halten, war es notwendig, die Informationen in eigenen Klassen erneut mit weiteren Informationen zu verknüpfen. Zu diesem Zweck wurden die Klassen *DomainSource*, *ProjectData*, *StorageSource* und *ResourceSource* für das JAX-RS-PSM erstellt. Die HTML-Dokumentation kann aufgrund der einfachen Struktur ohne zusätzliche Datenklassen generiert werden.

Für den Prototypen wurden folgende Templates für die HTML-Dokumentation definiert:

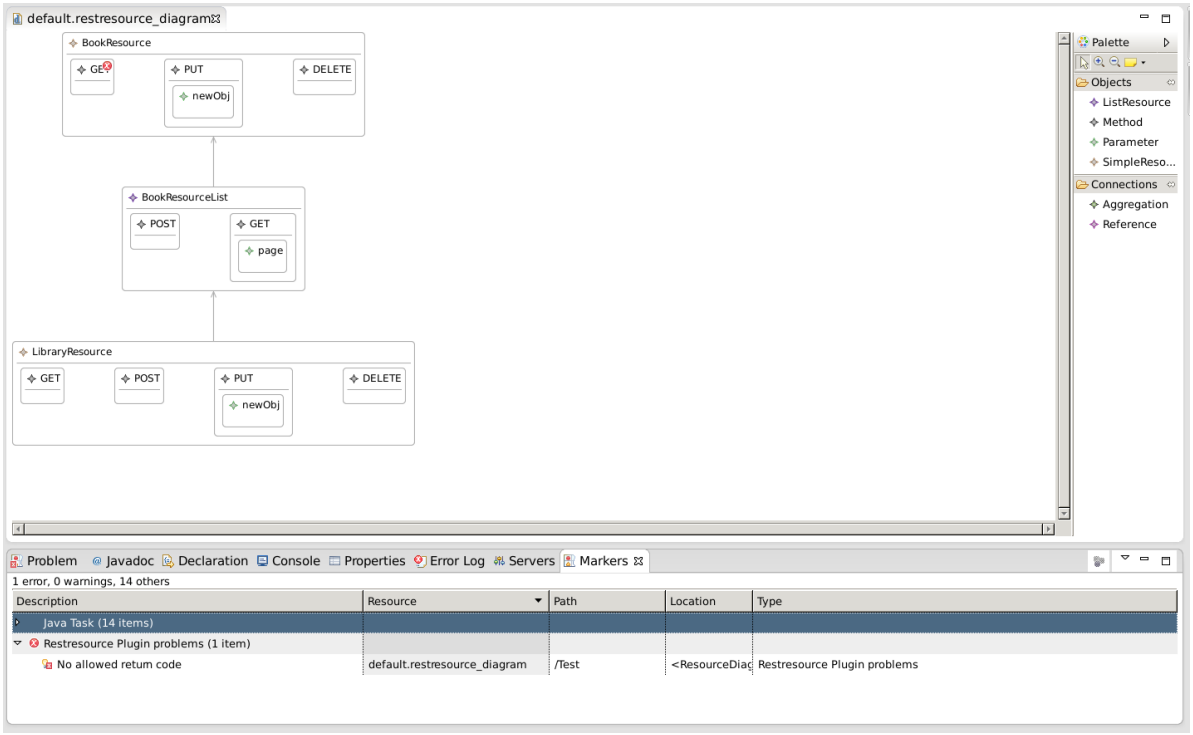

Abbildung 6.4 – Nicht valides Resourcen-Modell im zugehörigen GMF-Editor

Name	Beschreibung
`indexHtml`	HTML-Code der Dokumentation
`mainCss`	CSS-Code der Dokumentation

Es wird eine einzelne HTML-Seite mit Ankern zur Navigation generiert. Abbildung 6.5 auf der nächsten Seite zeigt einen Ausschnitt einer generierten Dokumentation der Beispielapplikation (siehe Abschnitt 6.2 auf Seite 38). Zur besseren Übersicht ist es möglich, in der Dokumentation die Abschnitte der einzelnen Resourcen aus- und einzublenden.

Des Weiteren wurden folgende Templates zur Generierung einer JAX-RS-Applikation erstellt:

Name	Beschreibung
`applicationClass`	Template einer `Application`-Klasse, die seit Jersey 2.0 zwingend notwendig wird
`classDomain`	Domänen-Klasse
`listResource`	Klasse einer Listen-Resource mit JAX-RS-Annotationen
`classResource`	Klasse einer einfachen (Wurzel-)Resource mit JAX-RS-Annotationen
`storeClass`	Storage-Klasse zu einer Listen-Resource
`webxml`	Deployment-Descriptor für Web-Applikationen

Zu jeder Listenresource wird für die JAX-RS-Applikation eine zusätzliche Klasse generiert, die den Inhalt der Liste verwaltet (*Storage-Klasse*).

Abbildung 6.5 – Ausschnitt einer generierten Dokumentation

Um die Templates übersichtlich zu halten, wurden repetitive Teile, wie die Signatur von Methoden bei der JAX-RS-Anwendung, in TXT-Dateien ausgelagert. Die Änderung der Dateiendung ist notwendig um den JET-Generator von Eclipse nicht zu verwirren, da dieser alle Dateien mit der Endung *jet* versucht zu übersetzen und daraus Klassen zu generieren.

Für die Generierung der JAX-RS-Applikation hält das Generator-Plugin zusätzlich alle nötigen Bibliotheken wie *Jersey 2.4*, JAXB und deren Abhängigkeiten, bereit. Nach erfolgreicher Generierung werden diese Bestandteile in das Projekt kopiert und entpackt.

Das *Eclipse Web Tools Platform Project*[1] definiert sogenannte *Project Facets* für Web-Anwendungen. Die *Project Facets* definieren Charakteristiken und Anforderungen eines Projekts. Ursprünglich

[1] `http://projects.eclipse.org/projects/webtools` (Letzter Abruf: 11.12.2013)

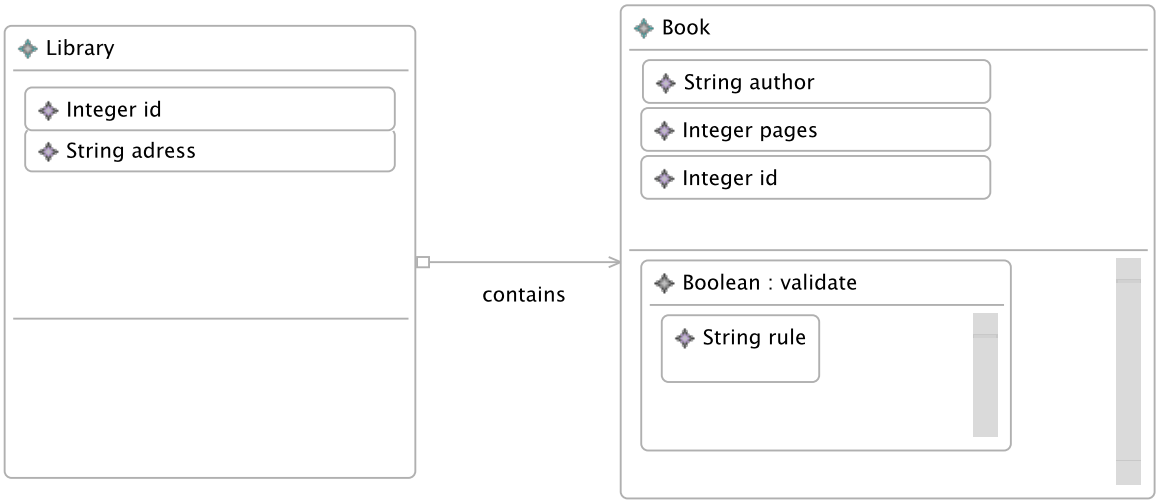

Abbildung 6.6 – Domänen-Modell der Beispielapplikation

waren sie lediglich für Java EE (Enterprise Edition) Projekte vorgesehen, aber das hat sich im Laufe der Zeit geändert.

Bei der Erstellung der JAX-RS-Applikation werden die *Facetten* für ein dynamisches Webprojekt sowie für ein JAX-RS-Projekt vom Generator hinzugefügt. Dadurch ist es Eclipse möglich, den *Deployment Descriptor* zu lesen sowie zu interpretieren und entsprechend in einer Baumansicht innerhalb der Projektübersicht anzuzeigen.

Zur Erstellung und Konfigurierung der generierten Projekte wurde die Klasse *ProjectCreator* erstellt. Dabei handelt es sich um eine statische Klasse, die von den Handler-Klassen der Menüpunkte aufgerufen wird. Ihre einzige öffentliche Methode erzeugt ein Projekt mit allen notwendigen Ordnern und *Project Facets*.

6.2 Beispiel

Im Rahmen der Entwicklung wurde eine Beispielapplikation modelliert und anschließend generiert, um die Funktionsweise des Prototypen zu überprüfen. Das Beispiel modelliert eine Bücherei (*Klasse Library*), die Bücher (*Klasse Book*) aggregiert.

Abbildung 6.6 zeigt das Domänen-Modell dieser Applikation als Diagramm des entwickelten Editors aus dem Prototypen. Es besteht aus zwei Domänen-Objekten, die jeweils mehrere Attribute besitzen. Zusätzlich besitzt das Domänen-Objekt *Buch* eine Methode, die später in der GET-Methode der REST-API aufgerufen werden soll.

Abbildung 6.7 auf der nächsten Seite zeigt das daraus entstehende Resourcen-Modell bei den vorgegebenen Einstellungen des Wizard.

Den Resourcen wurden im Deployment-Modell folgende URI zugeordnet:

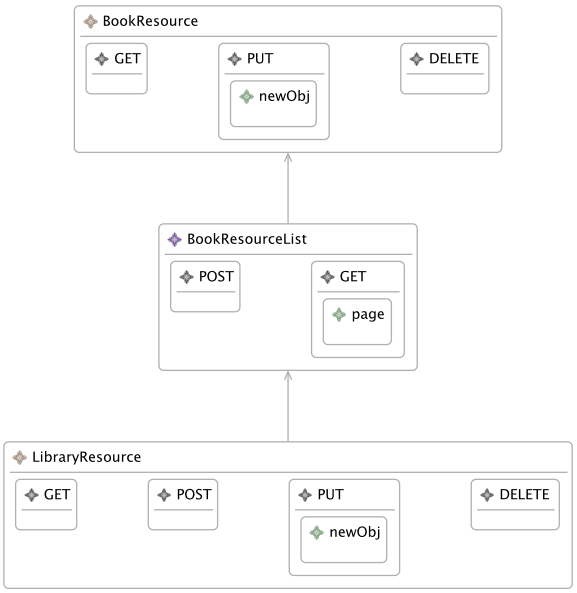

Abbildung 6.7 – Resourcen-Modell der Beispielanwendung

Name	URI
`Library`	library/
`Book`	books/
`Booklist`	{id}

Aus diesen Modellen wurde dann jeweils ein PSM für die HTML-Dokumentation und die JAX-RS-Anwendung erzeugt und entsprechende Einstellungen vorgenommen um daraus dann die entsprechenden Artefakte zu generieren.

Die HTML-Dokumentation wurde in einem Projekt mit dem Namen *com.examle.rest.documentation* gespeichert, für die JAX-RS-Anwendung wurde das Projekt mit dem Namen *com.example.rest* verwendet.

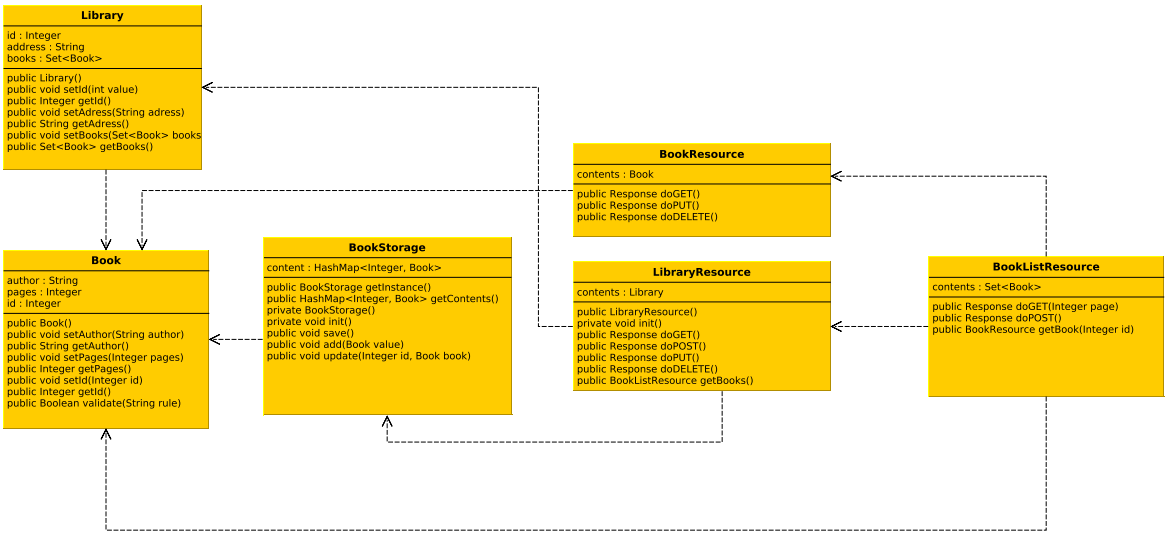

Abbildung 6.8 – Struktur der generierten JAX-RS-Anwendung

7 Zusammenfassung und Ausblick

Es wurde eine flexible Modellstruktur geschaffen, mit der es möglich ist eine REST-An-wendung zu modellieren. Hierbei wurde vor allem Augenmerk auf eine grundsätzliche Modellstruktur gelegt und zur Überprüfung der Flexibilität der Struktur zwei unterschiedliche Zielplattformen verwendet.

Die Modellstruktur erwies sich als geeignet, um einen Großteil der benötigten Grundinformationen der Anwendung einzufangen und übersichtlich aufzuteilen. Durch die getrennten Modelle sind komplexe Validierungen in jedem Teilbereich möglich. Weiter besteht die Möglichkeit die einzelnen Modelle zu weiteren Aufgaben zu verwenden. So kann das Resourcen-Modell dazu dienen, einen REST-Client für eine Applikation zu generieren.

Ausblick

Eine Erweiterung dieser Arbeit könnte die Implementierung von komplexeren Validierungsprozessen mittels EVL sein, um beispielsweise im Deployment-Modell überprüfen zu können, ob jede modellierte Resource über einen Pfad durch die Resourcen erreichbar ist. Außerdem ist bei dieser Arbeit kein expliziter Schwerpunkt in Richtung HATEOAS gesetzt worden, wie es bei der Arbeit von Nicolas Karrer und Marco Sonderegger der Fall war [KS12]. Hier könnten die erstellten Modelle zusammengeführt werden und die entsprechenden Konzepte in die erstellte Modellstruktur an passenden Stellen eingefügt werden, um auch REST-Anwendungen auf Level 3 des RMM zu unterstützen.

Des Weiteren ist in der vorliegenden Modellstruktur keine komplexe Verhaltensmodellierung möglich. Hier könnte ein weiteres Modell, wie auch von [Sch11] beschrieben, in die Modellstruktur eingefügt werden und die zusätzlichen Informationen bei der Generierung der Artefakte Verwendung finden.

Literaturverzeichnis

[Eps13] Epsilon: *Epsilon Documentation*. http://eclipse.org/epsilon/doc/. Version: 2013. – Online: Letzter Aufruf: 12.12.2013

[Fie00] Fielding, R.T.: *Archectural Styles and the Desgn of Network-based Software Architectures*. Irvine, University of California, Diss., 2000

[Fie08] Fielding, Roy T.: *REST APIs must be hypertext-driven*. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. Version: October 2008. – Online: Letzter Aufruf: 24.11.2013

[Fow10] Fowler, Martin: *Richardson Maturity Model: steps toward the glory of REST*. http://martinfowler.com/articles/richardsonMaturityModel.html. Version: 2010. – Online: Letzter Aufruf 24.11.2013

[GPR06] Gruhn, Volker ; Pieper, Daniel ; Röttgers, Carsten: *MDA – Effektives Softwareengineering mit UML2 und Eclipse –*. Springer-Verlag Berlin Heidelberg, 2006 http://books.google.de/books?id=L7w3_HOE1AQC. – ISBN 9783540287469

[KS12] Karrer, Nicolas ; Sonderegger, Marco: *MSDS und REST*. Dep. Informatik IFS, Hochschule für Technik (HSR), Oberseestrasse 10, CH- 8640 Rapperswil, Schweiz, Dep. Informatik IFS der Hochschule für Technik (HSR), Bachelorarbeit, 2012. – In Bearbeitung am Lehrstuhl Informatik, Institut für Softwareentwicklung (IFS) bei Herrn Prof. Dr. Peter Sommerlad

[Mus12] Musser, John: Open APIs, what's hot, what's not. In: *Keynote at Glue Conference*, 2012, S. 23–24

[Sch11] Schreier, Silvia: Modeling RESTful applications. In: *Proceedings of the Second International Workshop on RESTful Design* ACM, New York, NY, USA, 2011, S. 15–21

[SVEH07] Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Bd. 2. Zweite, aktualisierte und erw. Aufl. dPunkt Verlag, Heidelberg, 2007

[Til11] Tilkov, Stefan: *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*. Zweite, aktualisierte und erw. Aufl. dPunkt Verlag, Heidelberg, 2011

[Twi13] Twitter: *REST API v1.1 Resources*. https://dev.twitter.com/docs/api/1.1. Version: 2013. – Online: Letzter Aufruf: 06.12.2013

[ZBD11] Zuzak, Ivan ; Budiselic, Ivan ; Delac, Goran: Formal Modeling of RESTful Systems Using Finite-state Machines. In: *Proceedings of the 11th International Conference on Web Engineering*. Berlin, Heidelberg : Springer-Verlag, 2011 (ICWE'11). – ISBN 978–3–642–22232–0, 346–360

Glossar

EuGENia Werkzeug zur automatischen Generierung von GMF-basierter Editoren 24

Hypermedia Multimediale Variante von Hypertext (zusammengesetzt aus Hypertext und Multimedia) 5

JAX-RS Programmierschnittstelle (API) in JAVA zur Unterstützung bei der Entwicklung von Web-Services nach REST 20, 45

REST Representational State Transfer, bezeichnet ein Programmierparadigma zur Entwicklung von Webanwendungen 1, 5

RESTEasy JAX-RS Implementierung von JBoss 21

RESTful Bezeichnung für Dienste, die die Anforderungen von REST vollständig genügen 1, 7

RESTlet JAX-RS Implementierung von Jerome Louvel 21

Akronyme

API Application Programming Interface 1–3, 7, 20, 21, 28, 29, 34, 35, 38

ATL Atlas Transformation Language 28

CIM Computational Independend Model 8

CRUD Create, Read, Update, Delete 15

CSS Cascading Style Sheets 36

CXF CeltiXFire 21

DRY Don't repeat yourself 7

EMF Eclipse Modeling Framework 3, 23, 24, 29, 31, 32, 34

EOL Epsilon Object Language 23, 24, 26, 27

ETL Epsilon Transform Language 15, 27, 28, 34

EVL Epsilon Validation Language 26, 35, 41

GEF Graphical Editing Framework 28, 29

GET Idempotente HTTP-Methode 6, 7, 35, 38

GMF Graphical Modeling Framework 3, 24, 26, 28, 29, 31–33, 35, 36, 45

HATEOAS Hypermedia as the Engine of Application State 5–7, 10, 35, 41

HTML Hypertext Markup Language 1, 3, 5, 14, 19–22, 34–36, 39

HTTP Hypertext Transfer Protocol 1, 5, 6, 15, 19, 20, 35

JAX-RS JAVA-API for REST 1, 3, 13, 20–22, 34–40

JAXB Java Architecture for XML Binding 21, 37

JET Java Emitter Templates 28, 35, 37

JSON Javascript Object Notation 19

JSP Java Server Pages 28

MDSD Model-Driven Software Development 9

NEA nichtdeterministisch endlicher Automat 3, 9, 11

PIM Plattform Independent Model 8, 20, 21

POST Nicht idempotente HTTP-Methode 3, 6, 7, 15–17

PSM Plattform Specific Model 3, 8, 14, 20–22, 34, 35, 39

QVT Query View Transformation 28

REST Representational State Transfer 1–3, 5–7, 9–11, 14, 15, 19, 20, 38, 41

RMM REST Maturity Model 7, 41

RPC Remote Procedure Call 7, 48

SOAP Simpe Object Access Protocol 1, 6

UML Unified Modeling Language 3, 10

URI Uniform Resource Identifier 5–7, 9, 10, 20, 35, 38, 39

URL Uniform Resource Locator 19, 21

XMI XML Metadata Interchange 29

XML eXtensible Markup Language 19, 29, 32, 48

XMLRPC eXtensible Markup Language Remote Procedure Call 6

XSD XML Schema Definition 29

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.

Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

(Ort, Datum)	(Unterschrift)