Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Bachelorarbeit Nr. 275

Modellierung und Ausführung von REST Service Kompositionen mit BPEL

Marc Schmid

Studiengang: Softwaretechnik

Prüfer: Prof. Dr. Dr. h. c. Frank Leymann

Betreuer: Dipl.-Inf. Florian Haupt

begonnen am: 12.11.2015

beendet am: 11.05.2016

CR-Klassifikation: C.2.4, H.5.4, H.4.1

Zusammenfassung

In einer Vorarbeit wurde eine Erweiterung für BPEL definiert und implementiert, welche eine Orchestrierung von REST Services mit BPEL zu erlaubt. In einer weiteren Vorarbeit wurde für diese Erweiterung eine graphische Benutzerschnittstelle zur Modellierung von BPEL-Prozessen entworfen und implementiert. Obwohl beider Arbeiten auf dem selben theoretischen Datenmodell arbeiten, sind die jeweilige Implementierungen sehr unterschiedlich.

In dieser Arbeit sollen die beiden Implementierung der Datenmodelle analysiert werden. Basierend auf dieser Analyse soll ein neues Datenmodell geschaffen werden, welches die bestehenden Datenmodelle ersetzen soll. Dieses Datenmodell soll so beschaffen sein, dass eine Veränderung daran nur sehr geringe oder sogar gar keine Änderungen an den bestehenden Arbeiten mehr hervorruft. Diese vereinheitliche Datenmodell soll dann in die bestehenden Arbeiten integriert werden, ohne jedoch deren Funktionalität zu beeinflussen.

Inhaltsverzeichnis

1 Einleitung **7**
1.1 Ziel der Bachelorarbeit 8
1.2 Aufbau der Bachelorarbeit 8

2 Grundlagen **9**
2.1 BPEL 9
2.2 REST 13
2.3 Methodiken zur Softwareentwicklung 18
2.3.1 Schnittstellen/Interfaces 18
2.3.2 Code-Generierung 19

3 BPEL4REST **21**
3.1 Überblick der Arbeit 21
3.2 Aufbau der BPEL4REST Extension 22
3.2.1 Struktur eines Verb-Elements 22
3.2.2 Struktur des <context>-Elements 24
3.2.3 Struktur des <requestParameters>-Elements 25
3.2.4 Struktur des <responseParameters>-Elements 27
3.2.5 Struktur des <responseHeader>-Elements 28
3.3 Architektur von BPEL4REST 29

4 Modellierung von REST Service Kompositionen **33**
4.1 Überblick der Arbeit 33
4.2 Architektur der RESTModelingExtension 35

5 Analyse der Datenmodelle **37**
5.1 Modellimplementation *BPEL4REST* im Detail 38
5.1.1 ContextParser und ExtensionParser 40
5.1.2 ModelInterface und ModelImpl 41
5.1.3 ContextInterface und Context 42
5.1.4 ResponseHeaderInterface und ResponseHeader 43
5.2 Modellimplementation *RESTModelingExtension* im Detail 44
5.3 Entscheidung zum Entwurf des einheitlichen Datenmodells 46

6 Entwurf und Implementation des vereinheitlichen Datenmodells **49**
6.1 Entwicklungs- und Testumgebung 49
6.1.1 Eclipse 49
6.1.2 Apache Tomcat 49
6.1.3 Apache ODE 50
6.2 Variablen in der Apache ODE und dem Eclipse BPEL Designer . . . 51
6.3 Änderungen am Modell der *RESTModelingExtension* 53
6.3.1 Änderung am RESTActivityDeserializer 53
6.4 Änderungen am Modell *BPEL4REST* 54
6.4.1 ModelInteface und ModelImpl 55
6.4.2 ContexParser und ExtensionParser 57

6.4.3 ContextInterface und Context 57
6.4.4 ResponseHeaderInterface und ResponeHeader 58

7 Zusammenfassung und Ausblick 59
7.1 Ausblick 60

Abbildungsverzeichnis

1 Ein Beispiel BPEL-Prozess im Eclipse BPEL-Designer [1] 9
2 Die Ableitung von REST nach Fielding [11] 13
3 Graphisches Modell des Eclipse Modeling Frameworks [2] 20
4 Architekturübersicht von BPEL4REST nach Fischer [12] 29
5 Architektur des Modells nach Fischer [12] 30
6 Architektur des Contollers nach Fischer [12] 30
7 Reiter des graphischen Modellierungswerkzeugs für *BPEL4REST* . . 34
8 Übersicht der Architektur nach Kalach [16] 35
9 Klassendiagramm des Modellpakets von *BPEL4REST* 38
10 Sequenzdiagramm eines *BPEL4REST*-Aufrufes nach Fischer [12] . . . 39
11 Klassendiagramme der Parserklassen von *BPEL4REST* 40
12 Klassendiagramm des ModelInterfaces von *BPEL4REST* 41
13 Klassendiagramme der Parserklassen von *BPEL4REST* 44
14 Vereinfachte Darstellung der Variablenhaltung in der *Apache ODE* und dem *Eclipse BPEL Designer* 51
15 Klassendiagramm des vereinheitlichen Modells von *BPEL4REST* . . 54

Listings

1 Teil des BPEL-Codes zu Abildung 1 11
2 Beispiel einer Schnittstelle in Java 18
3 Syntax eines HTTP-Verbs anhand von POST nach Fischer [12] 23
4 Syntax des <context>-Elements nach Kalach [16] 24
5 Syntax des <requestParameters>-Elements nach Fischer [12] 26
6 Syntax des <responseParameters>-Elements nach Fischer [12] 27
7 Syntax des <responseHeader>-Elements nach Fischer [12] 28
8 Änderung des RESTActivityDeserializer in der *RESTModelingExtension* 53
9 BPEL Variable Resolver in *BPEL4REST* 53
10 Modell-Variablen in **BPEL4REST** vor der Vereinheitlichung des Datenmodells 55
11 Modell-Variablen in **BPEL4REST** nach der Vereinheitlichung des Datenmodells 56
12 Code-Ausschnitt der neuen BPEL-Parsers von *BPEL4REST* 57

1 Einleitung

Durch die fortschreitende Digitalisierung sind fast alle Unternehmen dazu gezwungen, Teile ihrer Aufgabenbereiche zu automatisieren um ihre Effizienz zu erhöhen und Kosten zu sparen. Diese Digitalisierungen erfordern es, unterschiedlichste Aufgaben in eine Form zu bringen, die von Computern verstanden und bearbeitet werden können. Da die Aufgaben oft aus unterschiedlichen Aktivitäten bestehen, welche in einer bestimmten Reihenfolge bearbeitet werden müssen, muss das bearbeitenden System dazu in der Lage sein, einen Prozess auszuführen, welcher aus verschiedenen Aktivitäten besteht. Dafür wurden mehrere Beschreibungssprachen entwickelt, welche Aktivitäten erstellen können, diese miteinander kombinieren und dann auch ausführen. Bei dieser Komposition von Aktivitäten, wird die Aktivität als zentrale Komponente betrachtet, da sie kombiniert und wiederverwendet werden können.

Durch die Globalisierung ist es für Unternehmen üblich, an verschiedenen Standorten auf der Welt zu operieren. Durch die Vernetzung der Standorte wird es für die Unternehmen interessanter und lukrativer in Softwarebereichen sich von Desktopanwendungen zu entfernen und auf Web-Anwendungen und Web-Services zu setzen. Im *Service Oriented Computing* wird die Verwendung von Services zur Anwendungsentwicklung als grundlegende Komponente bezeichnet. Um neue und komplexere Anwendungen zu erstellen, werden einfacherer schon bestehende Services miteinander kombiniert. Die neu entstandene Anwendung, kann dann wiederum als Service behandelt werden. Dieser Vorgang wird auch als *Service Komposition* bezeichnet [13].

Entwickler, die einen Prozess von Aktivitäten modellieren wollen und dabei den Prinzipien des *Service Oriented Computing* folgen, müssen auf eine passende Vorgehensweise bei der Zusammensetzung der Services zu Aktivitätsprozessen achten. Eine Beschreibungssprache, die hierfür mögliche wäre, ist BPEL, eine Beschreibungssprache für die Orchestrierung WSDL-basierter Web Services (siehe Kapitel 2.1). Allerdings erfordert BPEL von den Services, dass die eine WSDL-Schnittstelle besitzen, was allerdings nicht immer der Fall sein muss. Im Jahr 2000 wurde eine Architektur Stil (REST) eingeführt, welcher ein alternatives Vorgehen zur Erstellung von Webservices vorstellt und keine WSDL-Schnittstelle erzwingt. Siehe dazu Kapitel 2.2.

Da BPEL eine WSDL-Schnittstelle voraussetzt, wurde in einer vorhergegangenen Arbeit (Kapitel 3) eine Erweiterung für BPEL erstellt, welche es ermöglicht REST Services in BPEL zu orchestrieren. Da dieser Erweiterung allerdings kein graphisches Modellierungswerkzeug anbietet, ist ein wichtiger Aspekt für die Modellierung von Service Kompositionen nicht gegeben. Ein weitere Arbeit (Kapitel 4) behebt diese Problematik, in dem sie eine Erweiterung für den Eclipse BPEL Designer [1] anbietet, mit der sich REST Services für BPEL auch graphischer modellieren lassen.

Auch wenn beide Erweiterungen auf dem gleichen theoretischen Datenmodell (siehe Kapitel 3.2) basieren sind ihre Implementierungen doch unterschiedlich (siehe Abbildung 13 und Abbildung 9). Dies führt zu unnötigem Mehraufwand, sollte das Datenmodell je geändert werden. Zum einen müsste in jeder Erweiterung die Implementierung des Datenmodells einzeln verändert werden. Zum anderen könnten diese Änderungen weitere Änderungen am Rest der Implementierung der Erweiterungen

nach sich ziehen.

Deshalb sollen in dieser Arbeit die Datenmodelle analysiert und vereinheitlicht werden. Durch die Verwendungen einen gleichen Datenmodells kann dieses später einfacher verändert werden, da es nur einmal verändert werden muss.

1.1 Ziel der Bachelorarbeit

Die Ziele, die in dieser Bachelorarbeit erreicht werden, sind die folgenden:

Zuerst sollen die Datenmodelle der BPEL-Erweiterungen *BPEL4REST* und *RESTModelingExtension* aus den Erweiterungen extrahiert und dann analysiert werden. Bei der Analyse soll darauf geachtet werden, inwiefern die Struktur der Datenmodelle der Struktur der XML-Elemente in der Spezifikation von *BPEL4REST* entspricht. Auch soll in der Analyse darauf geachtet werden, wie gut die Datenmodelle verändert werden können und wie viel Schaden in der Erweiterung entsteht, wenn das Datenmodell ausgetauscht wird. Basierend auf dieser Analyse soll ein vereinheitlichtes Datenmodell entworfen werden, welches für beide BPEL-Erweiterungen verwendet werden kann. Dieser Entwurf eines vereinheitlichen Datenmodells soll dann in beiden BPEL-Erweiterungen implementiert werden.

1.2 Aufbau der Bachelorarbeit

Diese Arbeit ist in folgender Weise gegliedert:

In **Kapitel 2 - Grundlagen** werden die grundlegenden Begriffe REST und BPEL erklärt, die für das Verständnis dieser Arbeit notwendig sind. Außerdem werden einige Paradigmen für den Entwurf und die Implementation von austauschbaren Softwaremodulen kurz beschrieben.
Kapitel 3 - BPEL4REST beschreibt eine bestehende Erweiterung für BPEL, welche die Möglichkeit liefert, REST Services mit BPEL zu orchestrieren.
In **Kapitel 4 - Modellierung von REST Service Kompositionen** wir eine bestehende Erweiterung für den Eclipse BPEL Designer [1] beschrieben. Diese Erweiterung ermöglicht es die BPEL-Erweiterung aus Kapitel 3 graphisch in Eclipse [9] zu modellieren.
Kapitel 5 - Analyse der Datenmodelle analysiert die Datenmodelle der in Kapitel 3 und Kapitel 4 beschriebenen Erweiterungen. Dabei werden die Implementierungen der Datenmodelle daraufhin untersucht, inwiefern sie verändert werden müssen um für beide Erweiterungen zu funktionieren.
In **Kapitel 6 - Entwurf und Implementation des vereinheitlichen Datenmodells** wird aus der Analyse aus Kapitel 5 der Entwurf für ein einheitliches Datenmodell gefertigt und dieses dann implementiert. Auch wird ein Test BPEL-Prozess vorgestellt, welcher mit den veränderten Erweiterungen erstellt und ausgeführt wurde.
Kapitel 7 - Zusammenfassung und Ausblick fasst die Ergebnisse dieser Arbeit zusammen und liefert einen weiteren Ausblick.

2 Grundlagen

In diesem Kapitel werden Begriffe und Technologien behandelt, welche für das Verständnis dieser Arbeit nötig sind. Es werden erst die Begriffe BPEL und REST vorgestellt und erläutert und dann kurz auf zwei Methodiken der Softwareentwicklung eingegangen.

2.1 BPEL

Die *Web Service Business Process Execution Language*, kurz WS-BPEL oder auch nur BPEL, ist eine auf XML basierende Sprache zur Beschreibung und Modellierung von Geschäftsprozessen. Die erste Spezifikation von BPEL wurde im Jahre 2002 von OASIS vorgestellt und 2007 in der Version BPEL 2.0 [6] aktualisiert. Dabei kündigte das Technische Komitee an, das die Arbeit an der Spezifikation von BPEL abgeschlossen sei und damit nicht mehr verändert werden würde.

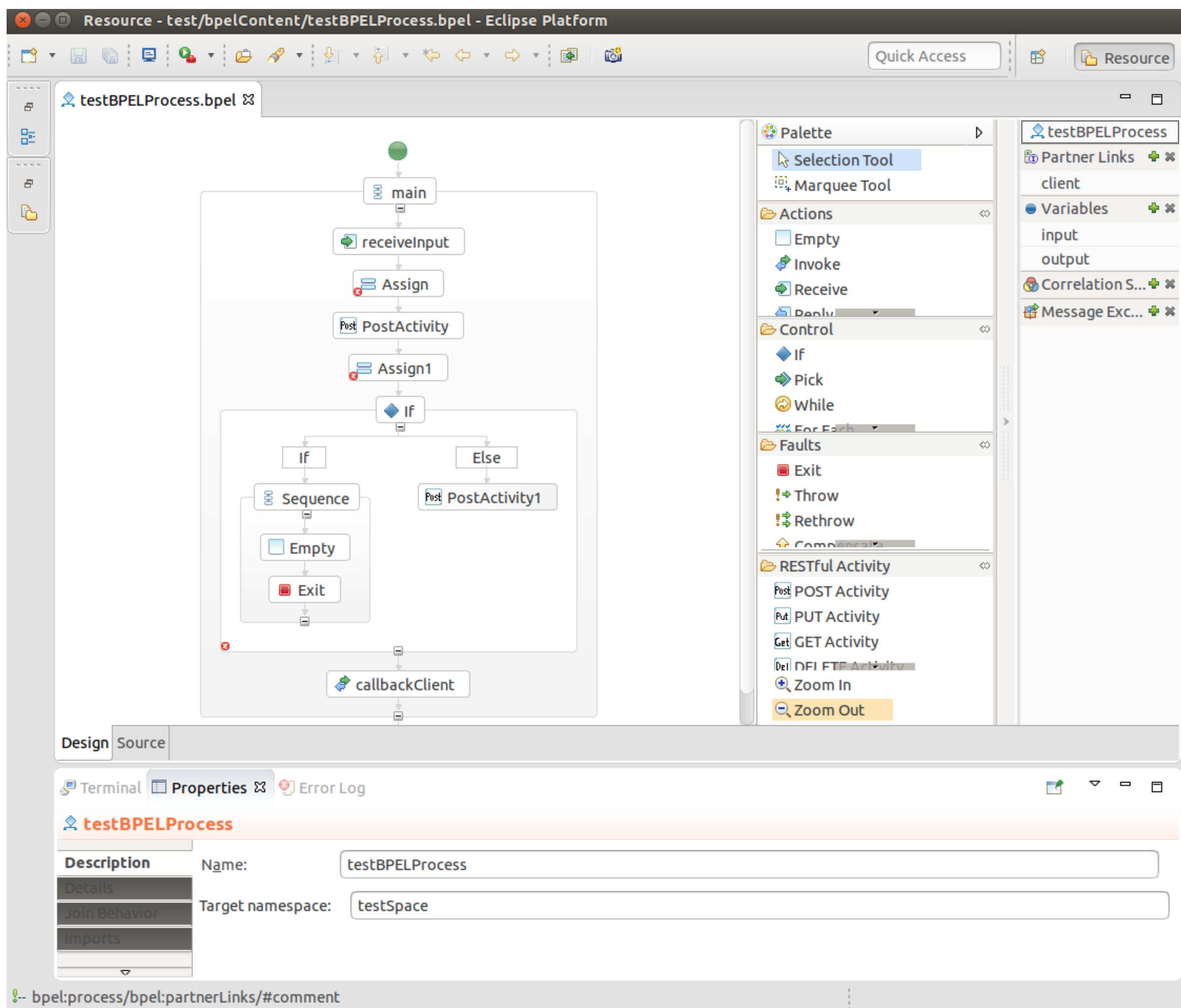

Abbildung 1: Ein Beispiel BPEL-Prozess im Eclipse BPEL-Designer [1]

Bei BPEL werden die einzelnen Aktivitäten eines Geschäftsprozesses als Webser-

vices implementiert. Diese Webservices sind durch dem W3C Standard WSDL 1.1 [10] beschrieben. Sämtliche Kommunikation von BPEL-Prozesses läuft über Webservices, neben den einzelnen Aktivitäten des Geschäftsprozesses und deren Partnern, wird auch die Beschreibung des Prozesses als Webservice bereitgestellt und kann damit von anderen BPEL-Prozesses verwendet werden. BPEL ist damit eine Sprache, welche zur Beschreibung der Orchestrierung von Webservices verwendet wird.

In BPEL selbst werden die Geschäftsprozesse in zwei Arten unterteilt: die abstrakten Prozesse und die ausführbaren Prozesse.

Die *abstrakten Prozesse* sind nicht für die Ausführung gedacht und werden auch nur teilweise spezifiziert. Sie haben allgemein eine beschreibende Rolle und dienen der Beschreibung des Verhalten des Prozesses, bieten allerdings auch spezielle Mechanismen zur Verbergung von technischen oder operationalen Details an. Auch können abstrakte Prozesse dafür verwendet werden um Templates zu erstellen.

Die *ausführbaren Prozesses* beschreiben den Ablauf eines Geschäftsprozesses über dessen Aktivitäten. Die Interaktionen zwischen den Aktivitäten des Prozesses und deren Partnern, sowie die Interaktion mit dem Prozess selbst läuft über Webservices. BPEL selbst muss keine Details zur Implementierung des Nachrichtentransports kennen. Für die Beschreibung des BPEL-Prozesses werden auch noch verschiedene XML-basierende Spezifikationen verwendet. Beispielhaft wird für die Beschreibung des Datenmodels die XML Schema Definition (XSD) oder für die Datenmanipulation XML Path Language (XPath).

Um einen BPEL-Prozess auszuführen, muss dieser einer passenden Software, einer BPEL-Engine, übergeben (deployed) werden. Momentan gibt es mehrere Open-Source, wie auch kommerzielle BPEL-Engine-Systeme auf dem Markt.

Ein BPEL-Prozess folgt einer klaren Struktur. Die Aktivitäten des Prozesses sind in verschiedene Arten von Aktivitäten unterteilt, welche verschiedene Aufgaben erfüllen und damit auch verschiedene Eigenschaften besitzen um diese zu erfüllen. Einige des möglichen Aktivitäten werden nachfolgend kurz aufgeführt.

Zur Interaktion mit den Webservices werden die Aktivitäten *<invoke>*, *<receive>* und *<reply>* definiert. Wobei *<invoke>* für den Aufruf eines Webservices gedacht ist, *<receicve>* auf die Nachricht eines Webservice wartet und *<reply>* eine Antwort auf eine Anfrage liefert. Um mit den Webservices kommunizieren zu können, definiert BPEL für die genannten Aktivitäten ein Reihe von Attributen, welche obligatorisch sind. Eine dieser Eigenschaften sind die *partnerLinks*, welche die Verbindung zum Webservice selbst darstellen und auch für den Webservice passend definiert werden müssen. Die Kommunikation zwischen dem BPEL-Prozess und den Webservices kann sowohl synchron, als auch asynchron ablaufen. BPEL bietet die Möglichkeit, den Inhalt von Nachrichten in Variablen abzuspeichern. Dafür ist die *<assign>* Aktivität definiert, welche die Manipulation des Inhaltes einer Variablen ermöglicht. Da BPEL eine auf dem XML-Standard basierende Sprache ist, ist auch die Nutzung von Variablen auf die von XML-Strukturen reduziert. Für die Manipulation der Variablen werden also weitere Sprachen wie zum Beispiel XPath benötigt.

Um die Komposition von komplexeren Prozessen zu ermöglichen, bietet BPEL auch Aktivitäten zur Kontrollflusssteuerung an. Nach diesen Aktivitäten müssen weitere Kind-Aktivitäten definiert werden, welche nur ausgeführt werden, falls die

Kontrollflussaktivität eintritt. Die Kontrollflussaktivitäten *<if>* und *<switch>* führen die passende Kind-Aktivitäten nur aus, wenn passenden Bedingungen zutreffen. *<flow>* erlaubt die parallele Ausführung ihrer Kind-Aktivitäten. Mit *<sequence>* werden alle Kind-Aktivitäten sequentiell abgearbeitet. *<while>*, *<repeatUntil>* und *<foreach>* erlauben es die gleichen Kind-Aktivitäten mehrmals hintereinander zu wiederholen. *<pick>* und *<wait>* lassen den Prozess auf einen bestimmten Zeitpunkt, eine Zeitspanne oder ein externes Ereignis warten, um erst danach weiterzumachen.

Um Fehlerbehandlung im BPEL-Prozess zu ermöglichen, sind die Aktivitäten *<throw>* und *<rethrow>* definiert worden, welches es erlauben Fehler zu generieren. Die Aktivität *<compensate>* reagiert dann auf diese "geworfenen" Fehler. Mit *<exit>* lässt sich der ganze Prozess vorzeitig beenden und *<empty>* ist eine Aktivität bei der einfach nichts passiert.

Eine Liste aller möglichen Aktivitäten für BPEL ist in ihrer Spezifikation [6] aufgeführt. Alle im Prozess von Aktivitäten verwendete Elemente, wie zum Beispiel *Variablen* oder *partnerLinks* müssen zu Beginn des BPEL-Prozesses deklariert werden, bevor sie von den Aktivitäten verwendet werden können. Die Deklaration dieser Elemente kann global passieren oder durch die *<scope>* Aktivität auf einen bestimmten Bereich eingegrenzt werden.

Listing 1 Teil des BPEL-Codes zu Abildung 1

```
<bpel:sequence name="main">
        <bpel:receive name="receiveInput" partnerLink="client"
                        portType="tns:testBPELProcess"
                        operation="initiate" variable="input"
                        createInstance="yes"/>
        <bpel:assign validate="no" name="Assign"></bpel:assign>
        <bpel:extensionActivity>
                <b4r:post name="PostActivity">
                        <context></context>
                        <requestParameters></requestParameters>
                        <responseParameters></responseParameters>
                </b4r:post>
        </bpel:extensionActivity>
        <bpel:assign validate="no" name="Assign1"></bpel:assign>
        <bpel:if name="If">
                <bpel:sequence>
                        <bpel:empty name="Empty"></bpel:empty>
                        <bpel:exit name="Exit"></bpel:exit>
                </bpel:sequence><bpel:else>
                        <bpel:extensionActivity>
                                <b4r:post name="PostActivity1">
                                        <context></context>
                                        <requestParameters></requestParameters>
                                        <responseParameters></responseParameters>
                                </b4r:post></bpel:extensionActivity>
                        </bpel:else></bpel:if>
        <bpel:invoke name="callbackClient"
                        partnerLink="client"
                        portType="tns:testBPELProcessCallback"
                        operation="onResult"
                        inputVariable="output"/>
</bpel:sequence>
```

Da ein BPEL-Prozess in seiner Grundform nur als XML-Dokument vorliegt,

ist die Erstellung von BPEL-Prozessen über einen Texteditor ein umständliche Angelegenheit. Nicht müssen die Entwickler die Syntax von Hand programmieren, auch ist eine syntaktische Fehlererkennung erst während der Ausführung des BPEL-Prozesses möglich. Listing 1 zeigt einen Teil eines BPEL-Codes in seiner XML-Darstellung. Dabei wurde die Lesbarkeit noch durch die Verwendung eines Texteditors erhöht, welcher die XML-Syntax hervorhebt.

Um die Entwicklung von BPEL-Prozessen zu vereinfachen bieten Modellierungswerkzeuge auch eine graphische Oberfläche an. Der BPEL-Prozess liegt im Modellierungswerkzeug dann in zwei Modellen vor, einmal als XML-Dokument und einmal als graphisches Modell. Durch die graphische Darstellung wird die Modellierung von BPEL-Prozessen stark vereinfacht, da der Entwickler nicht mehr auf die syntaktische Darstellung achten muss, sondern sich auf die Semantik des BPEL-Prozesses konzentrieren kann. Abbildung 1 zeigt den BPEL-Code von Listing 1 als graphisches Modell des Eclipse BPEL Designers.

Viele Modellierungswerkzeuge besitzen auch die Fähigkeit den BPEL-Prozess auf syntaktische Fehler zu untersuchen, so dass die Entwickler schon vor der Ausführung Fehler erkennen können.

2.2 REST

REST, die Kurzform für *Representational State Transfer*, bezeichnet einen Architekturstil für verteilte Hypermedia Systeme, welcher von Roy Fielding in seiner Dissertation [11] im Jahr 2000 definiert wurde. In seiner Dissertation betrachtete Fielding verschiedene netzwerk-basierende Architekturstile und stellte gewisse Anforderungen wie Einfachheit, Erweiterbarkeit, Skalierbarkeit und weitere an diese. Von ihnen leitete er den REST Stil, welchen er in seiner Dissertation als hybriden Stil bezeichnet, ab.

Für die Definition von REST beginnt Fielding mit dem *Null Style*, welcher keine Einschränkungen besitzt und erweitert dieses inkrementell um weitere Einschränkungen bis REST vollständig definiert ist (siehe Abbildung 2).

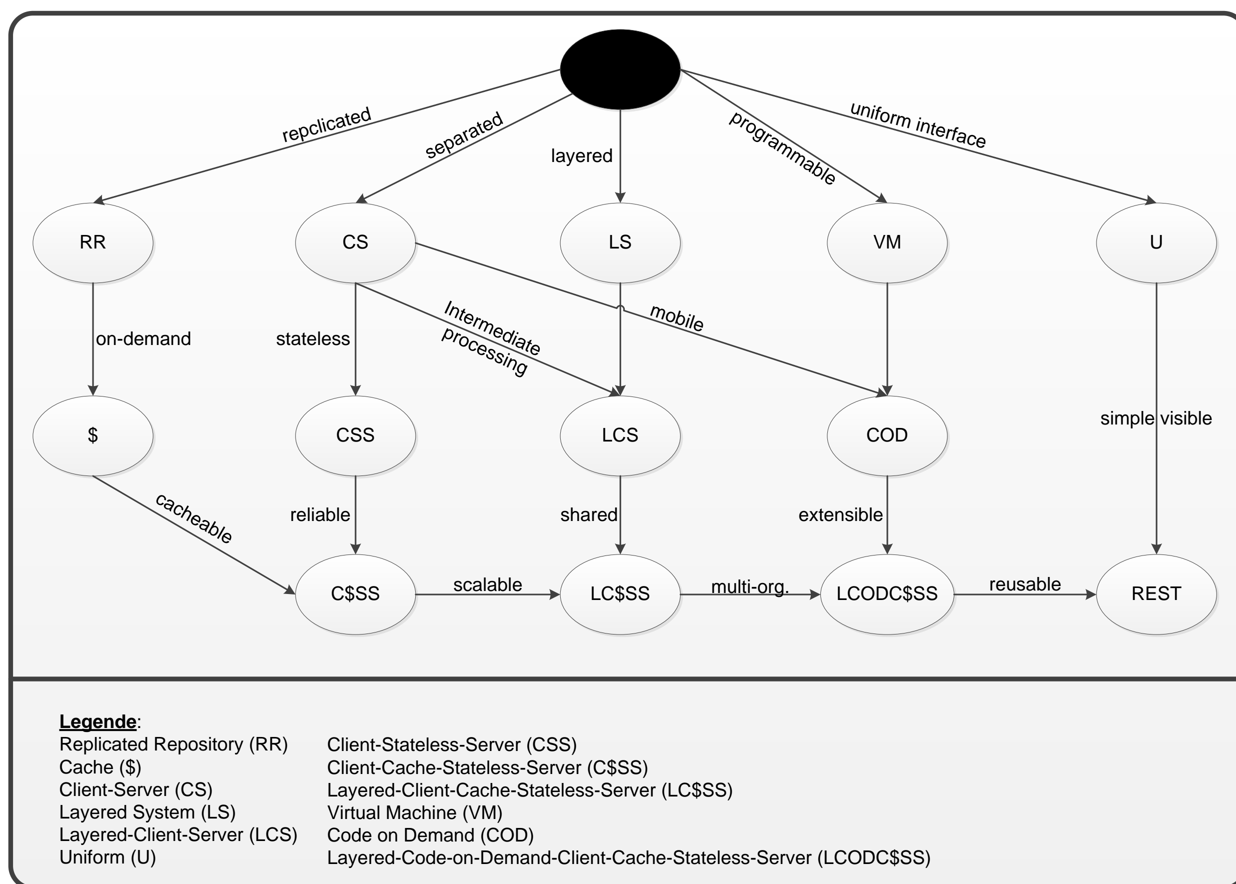

Abbildung 2: Die Ableitung von REST nach Fielding [11]

Die erste Einschränkung, die dem *Null Style* hinzugefügt wird, ist die Beschränkung von REST auf die *Client-Server-Architektur* (CS). Deren Kernprinzip ist die Trennung der Zuständigkeiten. Durch die Trennung der Zuständigkeit der Benutzerschnittstelle an den Client und die Zuständigkeit der Datenhaltung an den Server erhöht sich zum einen die Portabilität der Benutzerschnittstelle für verschiedene Plattformen, zum anderen wird die Skalierbarkeit des Servers durch Nutzung einfacherer Komponenten erhöht. Ein weitere Aspekt ist die Tatsache, dass so die verschiede-

nen Komponenten unabhängig voneinander entwickelt und verbessert werden können.

Als zweite Einschränkung, wird die Kommunikation beschränkt. Die Kommunikation in REST muss *stateless* sein. Bei der Interaktion zwischen Server und Client speichert der Server also keine Informationen, was den Client dazu zwingt in jeder Anfrage an den Server alle nötigen Informationen mitzusenden, damit der Server die Anfrage auch verstehen und verarbeiten kann. Dies vereinfacht die Implementierung des Servers, da er die Daten nicht speichern muss und erhöht nach Fielding die Skalierbarkeit, Sichtbarkeit und Zuverlässigkeit des Systems. Die Skalierbarkeit wird erhöht, da kein Status zwischen den Anfragen gespeichert werden muss. Falls der Server entscheidet, die Anfrage nicht selbst zu verarbeiten, sondern an einen anderen Server weiterzuleiten um die eigene Last zu verringern, ist dies einfach möglich. Da in der Anfrage alle nötigen Informationen enthalten sind, muss zwischen verschiedenen Servern, die die Anfrage beantworten können, keine Synchronisation bezüglich schon verarbeiteter Anfragen bestehen. Für den Client ist es egal ob die Antwort von Server A oder Server B kommt, unabhängig davon ob vorherige Anfragen von einem anderen Server beantwortet wurden. Die Sichtbarkeit wird dadurch verbessert, da allein durch die Anfrage selbst klar ist um was für eine Anfrage es sich handelt und was sie bezwecken möchte. Die Zuverlässigkeit wird erhöht, da es keinen Status gibt, der Verloren gehen kann. Damit können Verluste oder Probleme in der Kommunikation einfacher behoben werden.

Allerdings birgt das Prinzip der Zustandslosigkeit die Problematik, dass durch eine Sequenz von Anfragen eines Clients der Nachrichten-Overhead beim Server zunimmt, da in jeder Anfrage sämtliche Informationen enthalten sind. Dies führt zu einer verringert Netzwerkperformanz. Auch gibt der Server die Kontrolle über die Konsistenz der Anwendung ab, da der Client selbst den Status der Anwendung halten muss. Dies führt dazu, dass die Korrektheit und Konsistenz der Anwendung völlig in der Implementierung des Clients liegt.

Um die Netzwerkeffizienz wieder zu erhöhen, wird als dritte Einschränkung das *caching* ($) eingeführt. Dies erlaubt es die Daten einer Antwort einer bestimmten Anfrage als *cacheable* beziehungsweise als *not-cacheable* zu kennzeichnen. Eine als *cacheable* gekennzeichnete Antwort wird für einen gewissen Zeitraum im Cache gespeichert um bei einer erneuten Anfrage direkt wiedergeben zu werden um damit die Netzwerkauslastung und die Rechenkapazität des Servers zu schonen. Dies erhöht die Skalierbarkeit und Effizienz des Systems.

Allerdings kann das *caching* ($) auch die Zuverlässigkeit des Systems verschlechtern, da die Daten im Cache sich stark von denen einer regulären Anfrage unterscheiden könnten, falls die Daten auf dem Server im Zeitraum des Cachings verändert wurden.

Die vierte Einschränkung, die von Fielding hinzugefügt wurde, ist das *Layered System* (LS). Das *Layered System* (LS) ist eine Erweiterung der *Client-Server* (CS) Einschränkung und schreibt dem System vor, dass es in hierarchische Schichten aufgebaut werden muss. Die Komponenten der einzelnen Schichten dürfen dabei nur mit ihren unmittelbaren Nachbarn interagieren. In Kombination mit den anderen

Einschränkungen, soll diese Einschränkung die Performanz des Systems erheblich erhöhen.

Die fünfte Einschränkung, die *einheitliche Schnittstelle* (U), beschreibt Fielding als das zentrale Feature von REST, mit dem sich der REST-Architekturstil von anderen netzwerk-basierenden Architekturstilen unterscheiden. Diese Einschränkung zwingt alle Kommunikationspartner des Systems die gleichen Schnittstellen anzubieten und zu verwenden. Eine *einheitliche Schnittstelle* vereinfacht nach Fielding die Architektur des ganzen Systems, erhöht die Sichtbarkeit der Interaktionen im System und entkoppelt die Implementierung vom Service. Allerdings kann die Nutzung einer einheitlichen Kommunikation auch die Effizienz des Systems verschlechtern, da die Anwendungen über die standardisierten Methoden kommunizieren müssen, welche für ihren Zweck suboptimal sein können. Da REST laut Fielding als Schnittstelle hauptsächlich dafür entworfen wurde um bei Hypermedia Daten-Transfers im Internet effizient zu sein, sei dieser Nachteil ertragbar.

Die *einheitliche Schnittstelle* (U) für REST muss sich mit den folgenden vier Einschränkungen befassen: der *Identifikation von Ressourcen*, der *Manipulation von Ressourcen durch Repräsentationen*, *selbst beschreibende Nachrichten* und *Hypermedia as the engine of application state* (HATEOAS) [4].

Die grundlegende Abstraktionseinheit in einem System, das auf der REST-Architektur basiert, ist die *Ressource*. Fielding selbst beschreibt eine Ressource als eine nicht sichtbare, durch einen gemeinsamen Identifikator zusammengehaltene Menge von Repräsentationen. In REST kann eine Ressource also jegliche Art von Information sein, die einen Namen bekommen könnte. Um die Ressource auch identifizieren zu können und der Einschränkung der *Identifikation von Ressourcen* zu folgen, werden Ressourcenbezeichner verwendet. Eine mögliche Art der Ressourcenidentifikation ist die Nutzung des Uniform Resource Identifiers (URI) [17]. Da die Ressource nur ein Mapping auf eine Menge von Entitäten/Repräsentationen ist, müssen diese Repräsentationen die Informationen in einer Art vorliegen haben, damit der REST-Client sie auch verarbeiten kann. Die Repräsentation zeigt immer den aktuellen Stand einer Ressource auf dem Server an. Eine Möglichkeit zur Beschreibung des Datenformats einer Repräsentation wären die MIME Media Types [18], die in Form einer Bytesequenz sowie deren beschreibenden Metadaten vorliegen. Da eine einzelne Ressource mehrere unterschiedliche Repräsentationen haben kann, die es erlauben die Ressource in mehreren vordefinierten Formaten anzubieten, kann der Server entscheiden, auf eine Anfrage nach einer Ressource die passende Repräsentation als Antwort zurückzugeben. Die *Manipulation von Ressource durch Repräsentationen* erfolgt dadurch, dass der Client die Repräsentation selbst verändert und zurück an den Server sendet. Da die Ressource selbst von ihren Repräsentationen entkoppelt ist, tauscht der Server die empfangene, geänderte Repräsentation der Ressource aus. Nicht die Ressource selbst wird verändert sondern nur ihre Repräsentationen. Die *selbst beschreibenden Nachrichten* resultieren auf der *Statuslosigkeit* von REST und zwingen die Anfragen dazu, sämtliche nötige Informationen zu beinhalten. Da Repräsentationen auch Referenzen auf weitere Ressourcen enthalten können, muss der Client die Möglichkeit haben, diesen Referenzen zu folgen. *HATEOAS* erlaubt dem Client diesen Referenzen zu folgen und durch die URIs zu navigieren, welche

vom Server bereitgestellt werden. Da in den meisten Fällen die URIs via Hypermedia [3] vom Server bereitgestellt werden, muss der Client kein detailiertes Wissen darüber besitzen, wie er mit der Anwendung kommunizieren muss. Ein gundlegendes Verständnis über Hypermedia reicht völlig aus um mit den Referenzen arbeiten zu können.

Und als letzte Einschränkung führt Fielding noch *Code-On-Demand* (COD) ein, was zu einer Erweiterung der Funktionalität führen soll. Hierbei sollen Clients die Möglichkeit bekommen Code, in Form von Skripten, Applets oder ähnlichem, herunterzuladen und auszuführen. Durch diese Erweiterungen soll die Implementierung der Clients vereinfacht werden, da Funktionen und Features ausgelagert werden können. Diese Einschränkung ist aber nur optional.

Nach der Definition konzentriert sich REST also eher auf die Rollenverteilung und das Rollenverständnis der Kommunikationspartner und der Inhalte, als auf genaue Implementierungen wie etwa eine Syntax. Ein angebotener Dienst, der "RESTful" ist - also den Einschränkungen von REST folgt - muss dementsprechend nur die oben genannten Prinzipien erfüllen. REST steht damit eine Abstraktionsebene über der konkreten Implementierung.

In seiner Dissertation evaluiert Fielding den definierten REST-Stil und beschreibt, inwiefern das Hypertext Transfer Protocol (HTTP) [20] dem REST Stil entspricht. Betrachtete man HTTP in der Verbindung mit URI und den MIME Typen, lässt sich sagen, dass ein Großteil der Einschränkungen von REST schon erfüllt werden und macht damit das World Wide Web (WWW) unter Verwendung von HTTP zur "größten" Implementierung des REST Stils. Eine Änderung in der Entwicklung von APIs ergibt sich durch die Nutzung des REST Stils auch. Die Entwickler sind dazu angehalten ihr Augenmerk nicht darauf zu legen, welche Methoden und Operationen sie über die API anbieten, sondern in welcher Art die Ressourcen angeboten, gehalten und verwaltet werden. Der Schwerpunkt der Entwicklung sollte also auf korrekten Behandlung von Ressourcen liegen. Tilkov [21], zum Beispiel, unterteilt Ressourcen in folgende sechs verschiedene Kategorien:

Primärressourcen als die wichtigsten Entitäten, da sie den Kern einer Anwendung bilden. Beispielhaft wären hierfür ein Kochbuch oder der Römische Senat.

- `http://example.com/kartoffeln-kochen`
- `http://example.com/roemischer-senat`

Subressourcen sind Teile einer Ressource, die von ihrer Größe und ihrem Umfang her schon eine eigene Ressource sein könnten. Zum Beispiel ein einzelnes Rezept aus einem Kochbuch oder ein einzelner Senator des Senats.

- `http://example.com/kartoffeln-kochen/rezepte/pellkartoffeln`
- `http://example.com/roemischer-senat/senatoren/cato`

Listenressourcen sind eine Zusammenfassung von verschiedenen Ressourcen, beispielsweise eine Liste von Rezepten oder eine Liste an Senatoren.

- `http://example.com/kartoffeln-kochen/rezeptliste`
- `http://example.com/roemischer-senat/senatorenliste`

Filter können auf Listen angewendet werden, um die Auswahl der Ergebnisse auf bestimmte Kriterien einzuschränken. Diese Kriterien könnten eine Einschränkung auf rein vegetarische Rezepte sein.

- `http://example.com/kartoffeln-kochen/rezeptliste?filter=vegetarisch`
- `http://example.com/roemischer-senat/senatorenliste?filter=verheiratet`

Projektionen erlauben es, nur einen Teil der Informationen einer Ressource oder nur einige Elemente einer Liste abzufragen. Zum Beispiel die Zutaten eines Rezepts oder das Geburtsdatum eine Senators.

- `http://example.com/k-kochen/rezepte/pellkartoffeln?projection=zutaten`
- `http://example.com/r-senat/senatoren/cato?projection=geburtsdatum`

Aggregationen erlauben es die Information mehrerer Listen- oder Ressourcen zusammenzufassen.

- `http://example.com/kartoffeln-kochen/aggreggation?data=kapitel`
- `http://example.com/roemischer-senat/aggreggation?data=mitglieder`

2.3 Methodiken zur Softwareentwicklung

Nach der Definition von Ludewig und Lichter [15] ist die Entwicklung von Software ein stetiger Prozess. Auch nachdem ein Softwareprojekt vermeintlich abgeschlossen ist, fallen in der Wartung und der Weiterentwicklung Änderungen an der Implementierung der Software an. Dies erfordert beim Entwurf und der Architektur von Software Methodiken, welche es Entwicklern erlauben, Teile der Software mit geringem Aufwand auszutauschen.

Nachfolgend werden die Methodiken der *Schnittstellen* (oft auch *Interfaces* genannt) in der objektorientierten Programmierung und der Code-Generierung, basierend auf einem graphisch erstellten Modell, kurz vorgestellt.

2.3.1 Schnittstellen/Interfaces

Schnittstellen sind eine Methodik in der objektorientierten Programmierung. In der *Schnittstelle* werden die Methoden definiert, welche in den Klassen, die die *Schnittstelle* implementieren, enthalten sein müssen. Eine *Schnittstelle* liefert also eine Vorlage für die implementierenden Klassen, welche Methoden angeboten werden müssen. Die deklarierten Methoden liefern für die implementierenden Klassen allerdings nur eine syntaktische Definition. Bezügliche der Semantik der Methoden wird hierbei keine Aussage getroffen. In manchen Sprachen muss hierfür in eine externe Spezifikation oder die Kommentare der Entwickler geblickt werden, um an eine informelle Beschreibung der Semantik der Schnittstelle zu gelangen. Einige wenige Programmiersprachen bieten allerdings auch syntaktische Möglichkeiten die Semantik der Methoden eine *Schnittstelle* zu beschreiben.

Listing 2 stellte eine beispielhafte *Schnittstelle* in der Programmiersprache *Java* und deren Implementierung dar.

Listing 2 Beispiel einer Schnittstelle in Java

```
public interface SPQR
{
	public String getCitation();
}

public class Senator implements SPQR
{
	public String getCitation()
	{
		return "Ceterum censeo Carthaginem esse delendam";
	}
}
```

Durch die Verwendung einer *Schnittstelle* werden die angebotenen Methoden einer implementierenden Klasse schon eindeutig spezifiziert. Entwickler, die gegen eine solche Schnittstelle programmieren, haben den Vorteil, dass sie schon wissen, welche Methoden sie mit welchen Parametern aufrufen müssen und was sie zurück bekommen. Was genau in der implementierten Klasse passiert, muss sie nicht inter-

essieren, da die *Schnittstelle* schon alle für sie relevanten Informationen beinhaltet. *Schnittstellen* können also als "Grenze" zwischen Teilen der Software dienen. So können mehrere Implementierungen einer *Schnittstelle* existieren, welche verschieden implementiert wurden. Es können also ohne großen Entwurfsaufwand neue Klassen implementiert werden, da nur der Entwurf der Implementierung der *Schnittstelle* von Nöten ist, nicht die Kommunikation der Klasse mit dem Rest der Software. Wenn ein Datenmodell durch eine *Schnittstelle* mit dem Rest der Software kommuniziert, kann dieses Modell einfach ausgetauscht werden, da dass neue Modell nur gegen die *Schnittstelle* konsistent sein muss. Die Datenhaltung innerhalb der Klasse, kann völlig unterschiedlich zum ursprünglichen Modell sein, da nur die Methoden der *Schnittstelle* für den Rest der Software relevant sind. Dadurch lassen sich Teile des Codes einfach austauschen ohne Änderungen am Rest der Implementierung vorzunehmen. Diese Methodik erhöht die Wartbarkeit der Software und die Möglichkeit Teile der Implementierung auszutauschen.

2.3.2 Code-Generierung

Durch die Verwendung von höheren Programmiersprachen (Python, C++, Java) wird die Softwareentwicklung vereinfacht. Dies geschieht durch die Verwendung eines Compiler, der den Quellcode der höheren Programmiersprache in Zwischencode und/oder Maschinencode umwandelt, welcher dann vom System ausgeführt werden kann. Diese Abstraktion weg vom Maschinencode zu einer für Menschen besser lesbaren Sprache, erlaubt es Programmieren schneller und effizienter Software zu entwickeln. Durch die Abstraktion können auch komplexere Konstrukte einfacher dargestellt werden.

Eine weitere Abstraktion ist die Verwendung von Modellierungssprachen. Modellierungssprachen erlauben es Entwickler, den Entwurf und die Architektur ihrer Software in einem Modell detailliert und übersichtlich darzustellen.

Die meisten Modellierungsprachen bieten einen graphisches Modellierungswerkzeug an, mit dem der Entwurf graphisch erstellt und dann in Form verschiedener Diagramme betrachtet werden kann. Dies vereinfacht stark den Entwurf der Softwareentwicklung, da ohne syntaktische Kenntnisse eine Programmiersprache schon ein Modell der Software erstellt werden kann.

Die *Unified Modeling Language* (UML) [14] ist eine solche Modellierungssprache für die objektorientierte Programmierung. Sie erlaubt es die Klassen, Methoden, Attribute und weitere Komponente einer Programmiersprache in Klassendiagrammen detailliert darzustellen. Abbildung 3 zeigt ein graphisches Klassendiagramm im Eclipse Modeling Framework (EMF) [2]. EMF ist eine Erweiterung von des Eclipse Projects [9], welche es erlaubt Java-Klassen über Klassendiagramme zu modellieren.

Modellierungssprachen erlauben es aber nicht nur, die Architektur von Software zu modellieren. Sie bieten auch Werkzeuge an, um aus den graphisch modellierten Klassen Quellcode zu erzeugen. Dies kann von der einfachen Konstruktion der Klasse als reines Skelett, welches noch fertig implementiert werden muss, bis hin zum vollständigen, lauffähigen Programm reichen. EMF zum Beispiel erlaubt es, das modellierte Klassendiagramm vollständig in Java-Quellcode zu übersetzen.

Die Verwendung solcher Modellierungswerkzeuge vereinfacht die Softwareentwicklung, da Programmcode durch die graphische Modellierung erstellt werden kann.

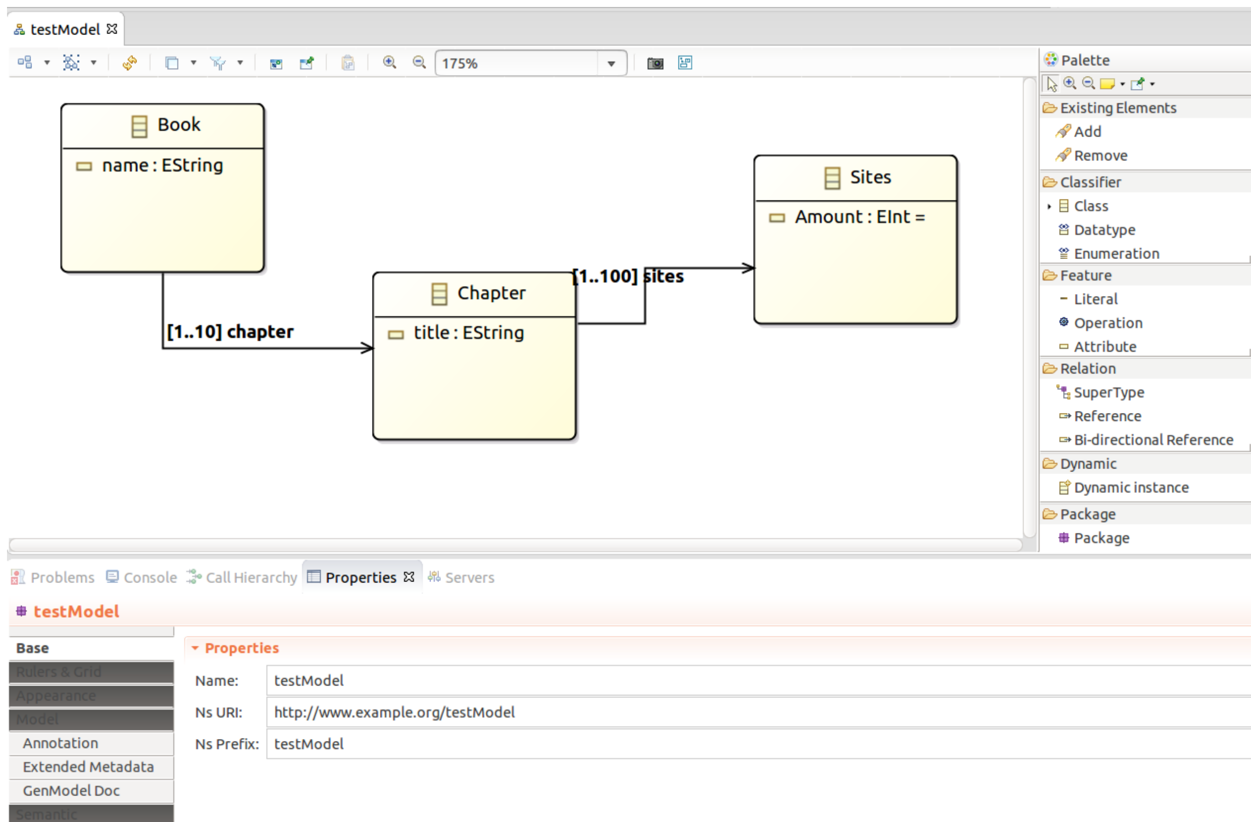

Abbildung 3: Graphisches Modell des Eclipse Modeling Frameworks [2]

Änderungen an einem Modell müssen deshalb nicht kompliziert im Code angepasst werden, sondern können in der graphischen Repräsentation einfach verändert werden, welche dann den funktionierenden Programmcode erstellt.

3 BPEL4REST

In diesem Kapitel wird einen existierende Erweiterung für BPEL-Prozesse vorgestellt, welche die Orchestrierung von Webservices bezüglich der Definition von REST ermöglicht. Diese Erweiterung, im weiteren einfach *BPEL4REST* genannt, definiert neue Aktivitätstypen für BPEL-Prozesse, welche die Eigenschaften von REST enthalten. Nachfolgend wird ein Überblick über *BPEL4REST* dargestellt, gefolgt von eine Beschreibung der neuen BPEL-Aktivitäten. Danach folgt noch eine kurze Beschreibung der Architektur der Erweiterung, welche für den Vergleich der Datenmodelle in späteren Kapiteln hilfreich ist.

3.1 Überblick der Arbeit

BPEL4REST ist eine Erweiterung von BPEL welche von Markus Fischer in der Bachelorarbeit *RESTful BPEL - Erweiterung von BPEL zur Orchestrierung von RESTful Web Services* [12] vorgestellt wurde. Fischer führte in seiner Arbeit eine Analyse über bestehende Erweiterungen von BPEL bezüglich der Anwendung von REST durch und stellte fest, dass die bestehenden Ansätze zur Orchestrierung von REST Webservices mit BPEL nicht vollständig konform mit der Definition von REST sind und damit Mängel aufweisen. Aus der Definition von REST entnimmt er drei wichtige Eigenschaften, die eine *RESTful* Erweiterung von BPEL erfüllen muss, welche von den aktuellen Ansätzen nicht erfüllt werden:

- Die URI von HTTP-Anfragen muss zur Laufzeit eines BPEL-Prozesses manipulierbar sein.
- Die Content-Negotiation muss für alle REST-Aufrufe in vollem Umfang unterstützt werden.
- HTTP-Header müssen für alle REST-Aufrufe eines BPEL-Prozesses manipulierbar sein.

Fischers Analyse zeigt, dass viele der bisherigen Ansätze auf die Verwendung von WSDL 1.1 [10] zurückgreifen. Dies führt zur Inkonsistenz mit Punkt 1 von Fischers wichtigen REST Eigenschaften, da WSDL 1.1 keinen Zugriff auf HTTP-Header zulässt. Die Nutzung von WSDL 2.0 [19] würde diese Problematik beheben, doch wird WSDL 2.0 von BPEL nicht unterstützt. Und da BPEL nach Aussage der Entwickler fertig gestellt ist, ist es nicht zu erwarten, dass diese Problematik behoben wird. Fischer stellt noch andere Ansätze vor, die versuchen diese beiden Problematiken zu umgehen, doch dies erfordert einen Eingriff auf die Implementierung von WSDL 1.1 beziehungsweise BPEL. Auch erfüllen diese Implementierungen, dann nicht vollständig die Anforderungen von REST.

Um die Anforderungen von REST zu erfüllen, greift Fischer auf die Möglichkeit von BPEL zu, Erweiterungen für BPEL zu erstellen, die *extensionActivities*. Diese erlauben es BPEL mit "Hausmitteln" zu erweitern und geben Fischer damit die Möglichkeit die Erweiterung wirklich *RESTful* zu machen. Die *extensionActivities* von *BPEL4REST* erlauben also die Manipulation von URIs und HTTP-Headern, sowie die Content-Negotiation. *BPEL4REST* definiert für die 5 HTTP-Verben:

HEAD, GET, POST, PUT und DELETE je eine eigene *extensionActivity* und einen manipulierbaren Kontext für die HTTP-Header um auf globale HTTP-Meta-Daten zuzugreifen.

Um einen BPEL-Prozess mit eigenen Erweiterungen ausführen zu können, ist es nötig, dass die BPEL-Engine selbst erweiterbar ist und diese Erweiterungen unterstützt. Fischer selbst benutzt in seiner Ausarbeitung jedoch die Apache ODE [7] um BPEL-Prozesse auszuführen, welche es ermöglicht, Erweiterungen zu integrieren, ohne die bestehende Implementierung der BPEL-Engine zu verändern.

3.2 Aufbau der BPEL4REST Extension

Da es ist möglich sein soll die BPEL Sprache zu erweitern, lässt es ihre Definition zu, dass Entwickler eigene Aktivitäten der Sprache hinzufügen können, auch wenn diese in der Spezifikation von BPEL nicht definiert sind. Dafür führt die BPEL-Spezifikation das XML-Element *<extensionActivity>* ein. Sämtliche neuen Aktivitäten müssen innerhalb dieses Elements deklariert und verwendet werden. Auch muss für jede Aktivität eine eigene *<extensionActivity>* erstellt werden, da die BPEL-Spezifikation pro *<extensionActivity>* nur ein Element zulässt, was allerdings wiederum weitere Elemente beinhalten kann. Im BPEL-Prozess muss dann auch definiert werden, welche Erweiterung für welche Art von *<extensionActivity>* zuständig ist, da die BPEL-Engine diese Aktivität dann an diese Erweiterung weiterreicht.

BPEL4REST definiert zum einen die *<extensionActivites>* für die Nutzung von REST Webservices in BPEL, zum anderen die Erweiterung für die Apache ODE, damit die BPEL-Engine diese *<extensionActivities>* auch verarbeiten kann. Dabei ist für jedes HTTP-Verb eine eigene *<extensionActivity>* mit dem jeweiligen Verbnamen entstanden. Die Aktivität beinhaltet Elemente für alle nötigen Informationen, die nötig sind um eine entsprechende *RESTfule* Interaktion auszuführen.

Die nachfolgenden Unterkapitel enthalten den Aufbau eines dieser HTTP-Verben, nämlich des Verbs *<post>*. Es werden die Struktur und die Elemente des XML-Elements erläutert und ein Vergleich zum HTTP-Verb dargelegt. Des weiteren hat Volha Kalach in ihrer Bachelorarbeit *Modellierung von REST Service Kompositionen* [16] Strukturierungen von Fischers XML-Elementen gefunden, die nicht sinnvoll sind. In dieser Ausarbeitung wurden die strukturellen Verbesserungen von Kalach gleich mitaufgenommen.

3.2.1 Struktur eines Verb-Elements

Nachfolgend wird die XML-Struktur der Erweiterungselemente dargelegt, sowie mit einer kurzen Erklärung auf die Bedeutung und Funktion der Elemente eingegangen.

Listing 3 beschreibt die vollständige Struktur einer der HTTP-Verb-Erweiterungen von Fischer anhand des HTTP-Verbs POST. Die Struktur der Erweiterungen für die HTTP-Verben GET, PUT, DELETE und HEAD ist analog, abhängig von der Funktion des HTTP-Verbs.

Listing 3 Syntax eines HTTP-Verbs anhand von POST nach Fischer [12]

```
<post host="" path="">
        <context ref="">
                ...
        <\context>
        <requestParameters>
                ...
        <\requestParameters>
        <responseParameters>
                ...
        <\responseParameters>
<\post>
```

Die Erweiterung für ein HTTP-Verb besteht grundsätzlich aus dem XML-Element des Verbs, hier *<post>*, dessen Attributen und den möglichen Kind-Elementen die das Verb haben kann. Abhängig des HTTP-Verbs sind verschiedene Kind-Elemente erlaubt. In diesem Beispiel wurde das *<post>*-Verb verwendet, da es neben *<put>* die meisten Kind-Elemente erlaubt. Die Kind-Elemente der Verben unterteilt Fischer in die Kategorien *verpflichtend* und *optional*, wobei die *verpflichtenden* Elemente verwendet werden müssen um die Erweiterung zu verwenden.

Die Attribute des Verbs selbst sind grundsätzlich *"host"* und *"path"*. Sie werden benötigt, damit eine Ressource eines REST Services adressiert werden kann. Fischer generiert aus diesen Attributen dann die URI, an die die HTTP-Anfragen gesendet werden. Dabei können in den Attributen die Werte direkt als String stehen, es kann aber auch nur ein Verweise auf eine BPEL-Variable sein, welcher dann von der BPEL-Engine während der Laufzeit aufgelöst wird. Eine genauere Betrachtung der Auflösung von BPEL-Variablen erfolgt in Kapitel 5

Die Erweiterung für ein HTTP-Verb beinhaltet nicht nur die URI an die die Anfrage gesendet werden soll, sondern auch noch weitere Variablen in denen Eigenschaften und Informationen der Anfrage, sowie Teile der Antwort der Anfrage gespeichert werden können. Nachfolgend folgt eine kurze Zusammenfassung der Kind-Elemente eines Verbs, die ausführlichere Beschreibung steht in den jeweiligen Unterkapiteln.

- *<context>*: Ein verpflichtendes Element. Hier kommen die Header-Elemente unter, die sich nicht mit der Content-Negotiation befassen, dementsprechend verschiedene allgemeine oder request-Header. *<context>* besitzt neben seinen Kind-Elementen ein einziges Attribut: *"ref"*. Dieses ist optional und verweist auf eine Variable, welche eine vollständige Instanz eines *<context>* beinhaltet. Die Nutzung dieses Attributes erlaubt es die Werte für verschiedene Anfrage zu verwenden, ohne sie jedes Mal neu erstellen zu müssen. Nach Fischers Definition, wird erst das Attribut eingelesen und danach werden die Kind-Elemente von *<context>* betrachtet. Falls sich die Kind-Elemente und die Werte des Attributs unterscheiden, wird der Wert des Attributes durch den Wert des Kind-Elementes überschrieben.

- *<requestParameter>*: Ein verpflichtendes Element. In diesem Element befinden sich alle Information, die für eine HTTP-Anfrage notwendig sind, wie weitere

Header und den HTTP-Message-Body der Request-Entity.

- *<responseParameter>*: Ein verpflichtendes Element. Hier befinden sich die Informationen, die nötig sind um erhaltene Antworten auf HTTP-Anfragen korrekt zu verarbeiten.

3.2.2 Struktur des <context>-Elements

In der ursprünglichen Version von *BPEL4REST* war unter *<context>* noch das Kind-Element *<conditionals>* geführt. Kalach legt in ihrer Ausarbeitung dar, dass die *<conditionals>* als XML-Darstellung der Bedingungen der request-Header aus dem *<context>*-Element entfernt gehören, da die *<conditionals>*-Elemente in vielen Fällen für verschiedene Aufrufe auch verschiedene Werte enthalten. Dies widerspricht allerdings der Grundidee des *<context>*-Elements, dafür verwendet zu werden, Daten zu beinhalten, welche für unterschiedliche Interaktionen wiederverwendet werden.

Listing 4 Syntax des <context>-Elements nach Kalach [16]

```
<context>
        <date value="boolean"/>
        <closeConnection value="boolean"/>
        <authentication>
                <user>...</user>
                <pass>...</pass>
        </authentication>
        <cachecontrol>...</cachecontrol>
        <additionalHeaders>
                <header name="" value=""/>
        </additionalHeaders>
</context>value
```

Nachfolgend werden die möglichen Kind-Elemente von *<context>* aufgeführt und kurz erläutert. Alle Kind-Elemente sind optional in ihrer Verwendung.

- *<date>* ist ein Element vom Typ *boolean* und grundsätzlich auf *true* gesetzt. Damit wird die Verwendung des *Date-Headers* der Request-Nachricht der HTTP-Anfrage geregelt. Falls der Wert mit *true* belegt ist, wird in der Request-Message der Header *Date* hinzugefügt und das Datum sowie die Uhrzeit der Anfrage hinzugefügt.

- *<closeConection>* ist ebenfalls ein Element vom Typ *boolean*. Über dieses Element wird gesteuert, ob der HTTP-Anfrage der HTTP-Header *Connection* hinzugefügt wird und welcher Wert in ihm steht. Ist dieser Wert *true*, wird mit der Anfrage mitgeteilt, dass die Verbindung nach Abschluss der Anfrage geschlossen werden soll. Hierbei wird der HTTP-Header *Connection* mit dem Wert *close* versehen und der Anfrage hinzugefügt.

- *<authentication>* ist ein Element, welches nur mit seinen beiden Kind-Elementen funktioniert. Es wird für eine gewünschte Authentifizierung bei Anfragen verwendet und wird in der Anfrage im *Authentication-Header* benutzt. Die Authentifizierung selbst wird unter Verwendung von *<user>* und *<pass>* von der Erweiterung selbst vorgenommen. Die Kind-Elemente *<user>* und *<pass>* beinhalten jeweils den Nutzernamen und das Passwort für die Authentifizierung als String.

- *<cachecontrol>* wird verwendet um in Anfragen den *cachecontrol-Header* zu setzten. *<cachecontrol>* selbst enthält einen String, welcher bei Verwendung direkt in den HTTP-Header *cachecontrol* übertragen wird. Der Inhalt des Strings muss dementsprechend konform mit der HTTP-Spezifikation für den *cachecontrol-Header* sein.

- *<additionalHeaders>* ist eine Element, welches nur aus Kind-Elementen des Typs *<header>* besteht. Es wird verwendet um die zusätzlichen Header für die request-Nachricht zu gruppieren.

- *<header>* beinhaltet alle weiteren Header, welche noch nicht im *<context>*-Element oder in den *<conditionals>* der *<requestParameters>* abgedeckt sind. Es erlaubt also auch eigene Header zu erstellen und zu verwenden. Ein *<header>*-Element besitzt zwei Attribute, welche auch verwendet werden müssen, um den *<header>* zu definieren. Über *name* wird der Name des Headers und über *value* der Inhalt des Headers beschrieben.

3.2.3 Struktur des <requestParameters>-Elements

Das *<requestParameters>*-Element beinhaltet nur Kind-Elemente. Diese Kind-Elemente beinhalten wiederum sämtliche Informationen, die nötig sind um eine HTTP-Anfrage zu erstellen, die verstanden und bearbeitet werden kann. Alle Kind-Elemente des *<requestParameters>*-Element sind optional.

Die Kind-Elemente des *<requestParameters>*-Element unterscheiden sich, abhängig davon, welches HTTP-Verb das Eltern-Element con *<requestParameters>* ist. So ist das *<requestEntity>*-Element nur für die Verben *POST* und *PUT* erlaubt. *HEAD*, *GET* und *DELETE* benötigen grundsätzlich keinen Messsage-Body in ihren Anfragen um ihre Aufgaben zu erfüllen. Bei *GET* wird diese Regel aufgeweicht, indem *GET* zu einem *conditional GET* verändert wird. In *BPEL4REST* geschieht dies durch die Nutzung des *<conditionals>*-Elements.

Listing 5 Syntax des <requestParameters>-Elements nach Fischer [12]

```
<requestParameters>
        <contentNegotiation>
                <acceptEntity type="" priority=""/>
                <acceptLanguage priority="">...</acceptLanguage>
        </contentNegotiation>
        <conditionals>
                <ifMatch>...</ifMatch>
                <ifModifiedSince>...</ifModifiedSince>
                <ifNoneMatch>...</ifNoneMatch>
                <ifRange>...</ifRange>
                <ifUnmodifiedSince>...</ifUnmodifiedSince>
        </conditionals>
        <requestEntity type="" entity="">
                <language>...</language>
        </requestEntity>
</requestParameters>
```

- *<contentNegotiation>* beinhaltet eine beliebige Anzahl von Kind-Elementen. Diese Kind-Elemente werden dazu verwendet und die HTTP-Content-Negotiation abzubilden.
  - *<acceptEntity>* definiert die Attribute, die nötig sind um den HTTP-Accept-Header zu konfigurieren. Mit *type* wird der gewünschte Media Type der Response Entity und mit *priority* die gewünschte Priorität bei verschiedenen akzeptierten Media Typen angegeben.
  - *<acceptLanguage>* beinhaltet die Attribute um den HTTP-Accept-Languages-Header abzubilden. Das Element definiert welche Sprachen mit welcher Priorität akzeptiert werden,
- *<conditionals>* beinhaltet jene Kind-Elemente die der HTTP-Standard für ein *conditional-GET* vorsieht. Da die Werte der Kind-Elemente von der Erweiterung direkt in die jeweiligen HTTP-Header übertragen werden, müssen auch hier die Werte der Elemente konform mit den passenden Werten der HTTP-Spezifikation für *conditional-GET* sein.
- *<requestEntity>* ist ein Element welchen nur von *<post>* und *<put>* verwendet wird. Es beinhaltet alle Informationen um für Anfragen den HTTP-Message-Body zu setzen. Dabei wird der referenzierte Inhalt des verpflichtenden Attributs *entity* in den HTTP-Message-Body geschrieben. Der Wert von *type* wird in den HTTP-Content-Type-Header, der Wert des Kindes *<language>* in den HTTP-Content-Language-Header geschrieben.

3.2.4 Struktur des <responseParameters>-Elements

Die Kind-Elemente des *<responseParamters>*-Element beinhalten Variablen für jene Informationen, die von den HTTP-Antworten stammen, welche auf die HTTP-Anfragen der Erweiterung geliefert werden. Dies beinhaltet Angaben zum Verhalten beim Empfang von bestimmten HTTP-Status-Codes und deren Mapping auf definierte BPEL-Fehler, als auch Angaben zur Überführung von HTTP-Message-Bodies in BPEL.

Auch das *<responseParamters*-Element enthält andere Kind-Elemente abhängig vom Eltern-Verb-Element. Da die HTTP-Spezifikation für *HEAD* keine Message-Bodies zulässt, ist auch bei *BPEL4REST* das *<acceptEntityMapping>*-Element nicht für das *<head>*-Element verfügbar.

Die Kind-Elemente des *<responseParameters>*-Element sind alle optional.

Listing 6 Syntax des <responseParameters>-Elements nach Fischer [12]

```
<responseParameters>
        <responseHeader variable=""/>
        <acceptEntityMapping type="" variable=""/>
        <catch status="" faultName=""/>
</responseParameters>
```

- *<responseHeader>* besitzt ein Attribut, welches auf eine Variable referenziert, in welcher die Header-Felder einer HTTP-Antwort gespeichert werden. Um die Header auch in BPEL verwenden zu können, werden in *BPEL4REST* diese Daten in Form eines XML-Dokuments des Schemas des *<responseHeader>*-Elements in eine BPEL-Variable gespeichert.

- *<acceptEntityMapping>* kann mehrfach als Kind-Element auftreten. Es wird verwendet um verschiedene, akzeptierte Media Types verschiedenen BPEL-Variablen zuzuordnen. Dies ist nötig, da Variablen in BPEL streng typisiert sind und ein mehrfach mapping nicht zulassen. Die Attribute *type* und *variable* sind verpflichtend und enthalten zum einen die Beschreibung des Media Types als auch die Referenz auf die Variable, in die der empfangene Message-Body geschrieben werden soll. Dabei muss der Media Type des Message-Bodies dem Media Type des *type*-Attributs entsprechen.

- *<catch>*-Elemente können beliebig oft auftreten. Sie sorgen dafür, dass die unter dem Attribut *status* definierten HTTP-Status-Codes von der Erweiterung mit dem unter dem Attribut *faultName* beschriebenen BPEL-Fehler behandelt werden.

3.2.5 Struktur des <responseHeader>-Elements

Das *<responseHeader>*-Element dient dem Zweck, mögliche Meta-Daten einer HTTP-Antwort zu speichern. Bis auf das *<status>*-Element sind alle Kind-Elemente optional.

Listing 7 Syntax des <responseHeader>-Elements nach Fischer [12]

```
<responseHeader>
        <status></status>
        <content>
                <type>...</type>
                <language>...</language>
                <location>...</location>
        </content>
        <date>...<date>
        <header name="" value=""/>
</responseHeader>
```

- *<status>* speichert den HTTP-Status-Code der HTTP-Antwort.
- *<content>* speichert über seine Kind-Elemente die Informationen zur HTTP-Content-Negotiation.
  - *<type>* enthält den Media Type aus dem HTTP-Content-Type-Header der HTTP-Antwort.
  - *<language>* enthält den Wert aus dem HTTP-Content-Language-Header der HTTP-Antwort.
  - *<location>* enthält den Wert aus dem HTTP-Content-Location-Header der HTTP-Antwort.
- *<date>* speichert das Datum und die Uhrzeit auf dem HTTP-Date-Header der HTTP-Antwort.
- *<header>* wird dafür verwendet um jene Header zu speichern, die noch nicht abgedeckt wurden, aber in der HTTP-Antwort als Header enthalten sind. Dabei enthalten die Attribute *name* und *value* jeweils den Namen und den Wert des Headers aus der HTTP-Antwort.

3.3 Architektur von BPEL4REST

Nachfolgend wird kurz die Architektur von *BPEL4REST* vorgestellt. Nachdem ein groben Übersicht über die Architektur gegeben wurde, werden die einzelne Komponenten der Architektur kurz vorstellt. Dabei wird noch ein kurzer Ausblick darüber geben, welche Komponenten bei einer Änderung des Datenmodells verändert werden müssen. Eine genauere Analyse des Datenmodells erfolgt in Kapitel 5.

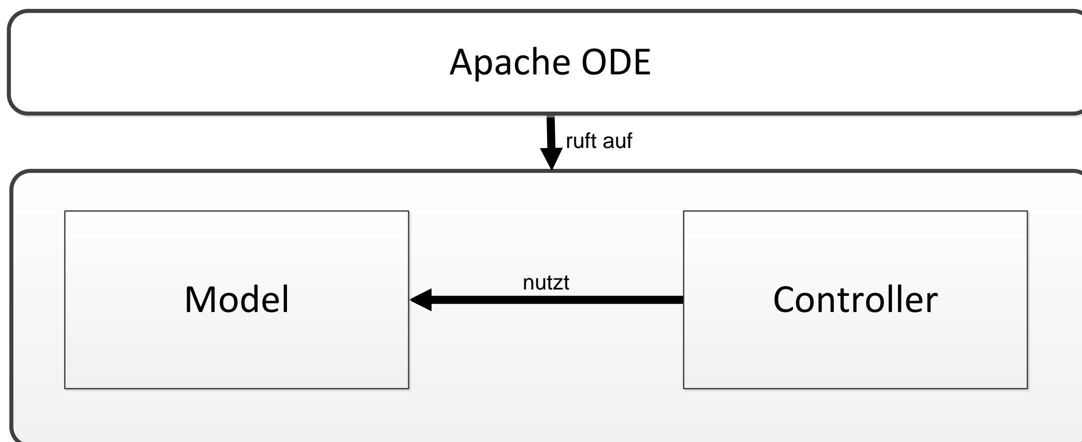

Abbildung 4: Architekturübersicht von BPEL4REST nach Fischer [12]

Abbildung 4 zeigt die grobe Architektur von *BPEL4REST*. Die Erweiterung selbst wird von der Apache ODE aufgerufen. Innerhalb der Erweiterung wird sie in zwei Komponenten aufgeteilt. Die *Model*-Komponente ist für die Datenhaltung zuständig. Die *Controller*-Komponente kümmert sich um das Steuern der Erweiterung, die Konvertierung der Daten, das Senden/Empfangen von HTTP-Nachrichten und die Kommunikation mit dem BPEL-Plan. Die beiden Komponenten sind durch eine Schnittstelle mit einander verbunden und können nur über diese miteinander kommunizieren.

Um das Datenmodell ersetzten zu können, muss hiernach also vor allem die *Model*-Komponente ausgetauscht werden. Da die *Controller*-Komponente allerdings eine Konvertierung der Daten vornimmt, kann es auch sein, dass hier auch Veränderungen vorgenommen werden müssen.

Abbildung 5 zeigt eine Verfeinerung der *Model*-Komponente. Nach Fischer wurde diese so entworfen, dass die tatsächliche Implementierung des Modells einfach ausgetauscht werden kann. Daher hat er die Funktionalitäten der Komponente in drei Schnittstellen aufgeteilt. *ContextInterface* definiert die Methoden zur Verwaltung des *<context>*-Elements der Erweiterung. *ResponeHeaderInterface* definiert die Methoden zur Verwaltung des *<responseHeader>*-Elements. *ModelInterface* definiert die Methoden zur Verwaltung weiteren Elemente, die *BPEL4REST* von der Apache ODE übergeben bekommt. Die Klasse *AbstractModel* fasst die drei Schnittstellen zusammen und die von ihr abgeleitete Klasse *ModelImpl* liefert die Funktionalitäten zur Verwaltung der von der Erweiterung benötigten Informationen.

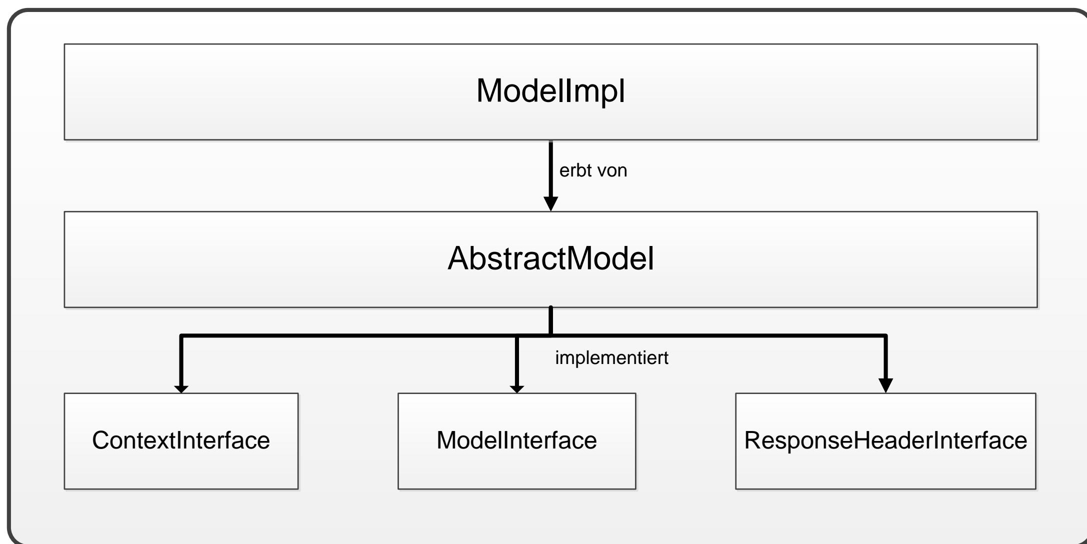

Abbildung 5: Architektur des Modells nach Fischer [12]

Diese Aufteilung der *Model*-Komponente bedeutet für eine Änderung des Datenmodell , dass nur die Implementierung **Interface*-Elemente ausgetauscht werden sollte und tiefere Eingriffe in die Erweiterung nicht notwendig sein müssten.

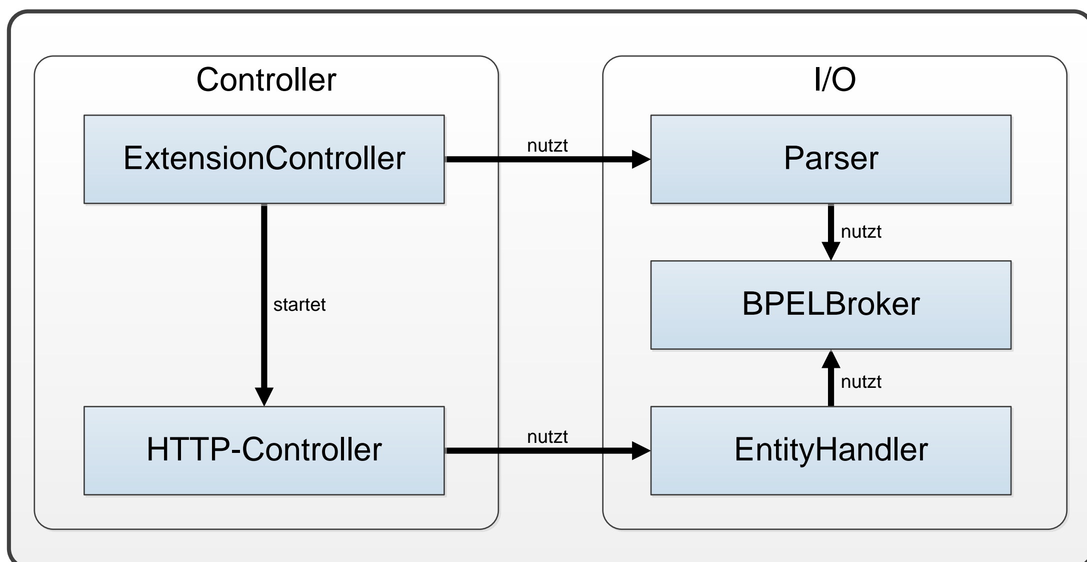

Abbildung 6: Architektur des Contollers nach Fischer [12]

In Abbildung 6 wird die *Controller*-Komponente dargestellt. Fischer teilt diese wiederum in zwei weitere Komponenten: die *I/O*-Komponente, die sich um die Kommunikation mit dem BPEL-Prozess und dessen Daten kümmert und die *Controller*-Komponente, die den Kontrollfluss der Erweiterung leitet. Der *HTTP-Controller* steuert dabei die Aufrufe und Verarbeitung sämtlicher HTTP-Kommunikation. Diese Komponente dürfte bei der Vereinheitlichung des Datenmodells nicht verändert

werden. Der *ExtensionController* steuert den Ablauf der ganzen Erweiterung. Also die Instanziierungen aller notwendigen Klassen und deren Steuerung. Auch an dieser Komponente sollte in dieser Arbeit nichts verändert werden müssen. Die Komponenten *BPELBroker* und *EntityHandler* befassen sich mit den Schnittstellen zwischen BPEL und der Apache ODE. Hierbei werden hauptsächlich BPEL-Variablen behandelt und HTTP-Entities erstellt. An diesen Komponenten sollte keine Veränderung vorgenommen werden müssen. Anders verhält es sich mit der *Parser*-Komponente. In dieser wird das eingelesen org.w3c.dom.Element verarbeitet und daraus ein Objekt im Datenmodell erstellt. Durch die Ersetzung des Datenmodells, wird also auch der *Parser* an das neue Datenmodell angepasst werden müssen.

4 Modellierung von REST Service Kompositionen

Die in Kapitel 3 vorgestellte Erweiterung zur Orchestrierung von REST Services in BPEL hat eine großes Manko. Um einen solchen REST Service in einen BPEL-Prozess einzufügen, müssen die Entwickler nicht nur die *BPEL4REST*-Spezifikation kennen und deren Syntax beachten, sie müssen die benötigten XML-Elemente auch von Hand in den BPEL-Prozess einfügen. Dies erschwert die Erstellung von BPEL-Prozessen und macht diese fehleranfällig, da syntaktischer Fehler erst zur Laufzeit des Prozesses erkannt werden.

Da der Eclipse BPEL Designer [1] keinerlei graphische Unterstützung zur Modellierung von *<extensionActivity>*-Elementen bietet, wurde eine Erweiterung dafür entwickelt, welche es erlaubt die REST Services von *BPEL4REST* graphisch zu modellieren. Diese Arbeit wird in diesem Kapitel vorgestellt.

4.1 Überblick der Arbeit

Die Bachelorarbeit *Modellierung von REST Service Kompositionen* [16] von Volha Kalach beschreibt eine Erweiterung (*RESTModelingExtension*) für den Eclipse BPEL Designer [1]. In ihrer Arbeit analysierte sie die BPEL-Erweiterung *BPEL4REST* [12] von Markus Fischer und entwickelte ein graphisches Modellierungswerkzeug. Dieses Modellierungswerkzeug ist eine Erweiterung für den BPEL Designer der IDE Eclipse [9].

Kalach legte bei ihrem Entwurf der graphischen Erweiterung zwei Prinzipien fest, welche bei der Implementierung des Prototypen eingehalten werden mussten und damit die Darstellung der Erweiterung stark beeinflussten:

- Die Erweiterung soll so weit wie möglich dem Eclipse BPEL Designer ähneln, um dem Benutzer die Einarbeitungsphase zu verkürzen.
- Die Semantik der grafischen Elemente soll nicht geändert werden, um den Benutzer nicht zu verwirren.

BPEL4REST führt für 5 HTTP-Verben jeweils eine eigenes Kind-Element einer *<extensionActivity>* ein. Da der Eclipse BPEL Designer für jeder der *<extensionActivity>*-Elemente ein graphischer Element darstellt, führt Kalach für jeder dieser REST-Aktivitäten ein eigenes graphisches Element ein in den Eclipse BPEL Designer ein. Diese Aktivitäten werden in der Palette der möglichen Aktivitäten unter einer eigenen Gruppe zusammengefasst.

Da komplexe Aktivitäten in BPEL über einfachere Aktivitäten modelliert werden, welche dann in der modellierte Reihenfolge ausgeführt werden und damit die komplexe Aktivität ergeben, sind die meisten nativen Elemente eines BPEL-Prozesses einfache Aktivitäten. Betrachtet man jedoch die REST-Aktivitäten von *BPEL4REST* erkennt man, dass die REST-Aktivitäten aus einer großen Anzahl von Kind-Elementen bestehen. Würde man der semantischen Struktur der Elemente eines BPEL-Prozesses folgen, müssten die Kind-Elemente *<context>*, *<responseParameters>* und *<requestParameters>* jeweils als eigene Aktivität modelliert werden. Doch sind diese Kind-Elemente in ihrer Funktion keine eigenständigen Aktivitäten die ausgeführt werden können, sondern Eigenschaften der REST-Aktivität.

Nach eine Analyse der unterschiedlichen Möglichkeiten, wie die REST-Aktivitäten graphisch darzustellen sind, kommt Kalach zu dem Schluss die REST-Aktivitäten als gewöhnliches graphisches BPEL-Aktivitäten-Element darzustellen. Die Kind-Elemente und ihre Attribute lassen sich über dann über verschiedene Reiter in dem Fenster "Properties" der Eclipse IDE [9] steuern. Dies erlaubt nach Kalach eine einfache und platzsparende Darstellung der Kind-Elemente und fügt sich in die bestehende Semantik der restlichen BPEL-Elemente ein. Auch beeinträchtigt diese Darstellung die Benutzbarkeit des Eclipse BPEL Designers nicht und da die REST-Aktivitäten als natives BPEL-Aktivitäts-Element verwendet werden, wird deren Darstellung bei einer Weiterentwicklung und Veränderung des Eclipse BPEL Designers automatisch an dessen neues Design angepasst.

In Abbildung 1 ist die Gruppierung der Elemente in der Palette des Eclipse BPEL Designers und die Darstellung einer REST-Aktivität (PostActivity) als ein einzelnes graphisches Element zu sehen. Abbildung 7 zeigt die Darstellung des <*requestParameters*>-Element in dessen Reiter des "Properties"-Fensters.

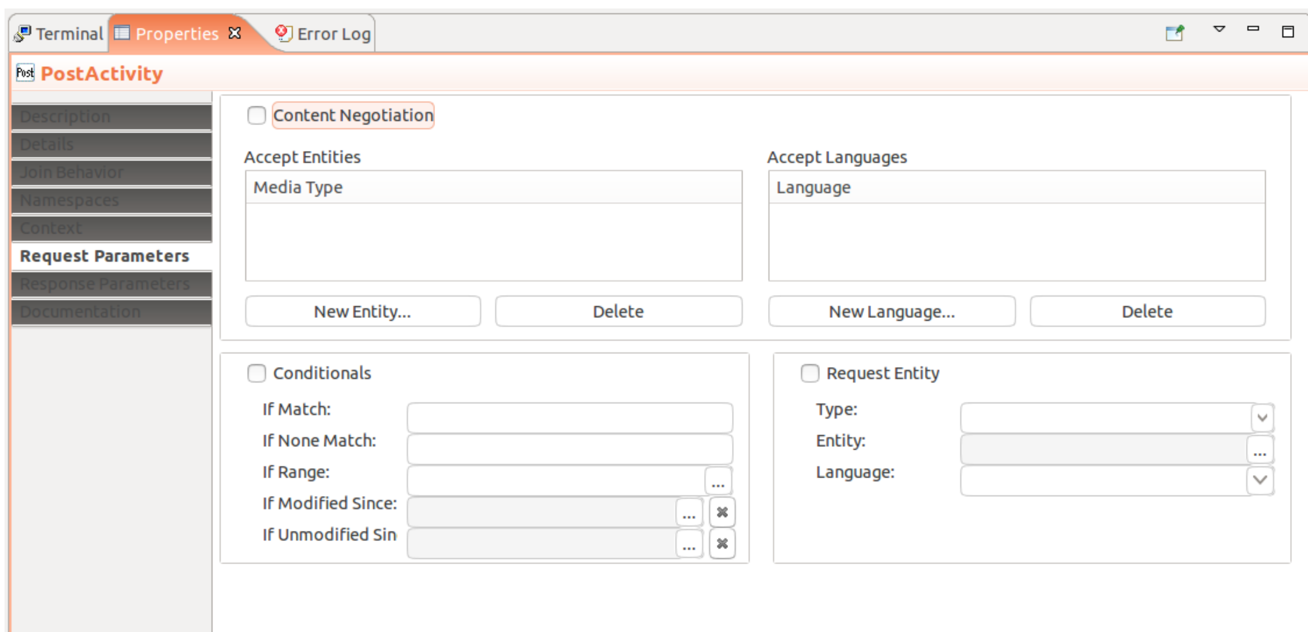

Abbildung 7: Reiter des graphischen Modellierungswerkzeugs für *BPEL4REST*

4.2 Architektur der RESTModelingExtension

Nachfolgend wird kurz die Architektur der *RESTModelingExtension* vorgestellt. Nachdem ein groben Übersicht über die Architektur gegeben wurde, werden die einzelne Komponenten der Architektur kurz vorstellt. Dabei wird noch ein kurzer Ausblick darüber geben, welche Komponenten bei einer Änderung des Datenmodells verändert werden müssen. Eine genauere Analyse des Datenmodells erfolgt in Kapitel 5.

RESTModelingExtension ist eine Erweiterung des Eclipse BPEL Designers. Eclipse IDE selbst wurde als erweiterbare Plattform entworfen, die die Entwicklung neuer Erweiterungen und Funktionalitäten über anschließbare Komponenten (auch Plug-ins genannt) ermöglicht [5]. Dabei funktioniert der grundlegende Mechanismus der Erweiterbarkeit von Eclipse darin, dass eine neue Erweiterung ihre Funktionalität und Elemente zu einer bestehenden Erweiterung hinzufügen kann, sofern diese *Extension-Points* dafür anbietet. Da die *RESTModelingExtension* eine Erweiterung für den Eclipse BPEL Designer ist, schließt sie an bestehende *Extension-Points* des BPEL Designers an. Durch den Aufbau und die bestehenden *Extension-Points* des Eclipse BPEL Designers, hat sich die in Abbildung 8 dargestellte Architektur der Erweiterung ergeben.

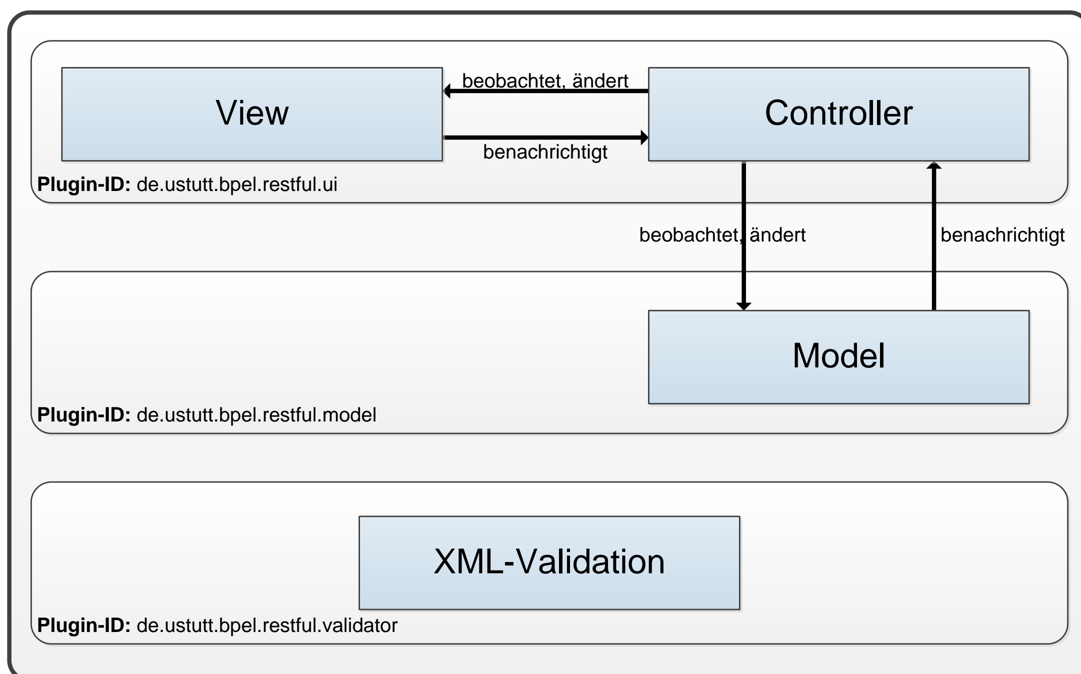

Abbildung 8: Übersicht der Architektur nach Kalach [16]

Dadurch ist die *RESTModelingExtension* keine Einzelkomponente, sondern besteht aus drei Erweiterungen *de.ustutt.bpel.restful.ui*, *de.ustutt.bpel.restful.model* und *de.ustutt.bpel.restful.validator*, die sich an den Eclipse BPEL Designer anschließen.

Dabei sind *de.ustutt.bpel.restful.ui* und *de.ustutt.bpel.restful.model* die Teile der Erweiterung, die die graphische Erweiterung bilden und damit das Hinzufügen von REST-Aktivitäten in einen BPEL-Prozess ermöglichen. Der *de.ustutt.bpel.restful.validator*

dient der Validierung des entstandenen BPEL-Prozesses bezüglich der Konformität mit der Spezifikation von *BPEL4REST*. Die Implementierungen der Erweiterungen basieren dabei auf dem Architekturprinzip des Model-View-Controllers. Dabei werden die Zuständigkeiten zwischen den Komponenten aufgeteilt. Dies sollte eine Austausch des Datenmodells vereinfachen, da die Zuständigkeiten der Komponenten strikt aufgeteilt sind und ihre Kommunikation miteinander auf bestimmte Wege festgelegt ist. In der *RESTModelingExtension* enthält die **Model-Komponente** die Daten, die während Ausführung persistent bleiben. In ihr wird also das Datenmodell der *BPEL4REST*-Erweiterung gespeichert. Es ist also anzunehmen, dass bei Änderung des Datenmodells hier der Hauptaugenmerk der Veränderung liegen wird. Die **View-Komponente** ist für die Darstellung aller sichtbaren Elemente der Erweiterung zuständig. Da in dieser Arbeit nur die Implementierung des Datenmodells verändert werden soll, aber nicht dessen Inhalt, dürfte diese Komponente für die Änderung des Datenmodells keine große Rolle spielen. Die **Controller-Komponente** verbindet die **Model-Komponente** und die **View-Komponente**. Sie wird von den beiden anderen Komponente benachrichtigt, wenn in ihnen etwas verändert wurde. Durch dieses Wissen kann sie die **Model-Komponente** und die **View-Komponente** bei Änderungen der jeweils anderen Komponente synchronisieren. Abhängig der Art der Kommunikation der **Controller-Komponente** mit der **Model-Komponente** muss sie bei der Änderung des Datenmodells auch verändert werden.

Der *de.ustutt.bpel.restful.validator* ist selbst nicht Teil der Model-View-Controller-Architektur. Da er dafür verwendet wird die Syntax des BPEL-Prozesses zu validieren, ist er für die Änderung der Implementierung des Datenmodells nicht relevant.

5 Analyse der Datenmodelle

In den beiden vorhergehende Kapiteln wurde die Architektur von *BPEL4REST* und der *RESTModelingExtension* vorgestellt. Durch die Betrachtung der beiden Architekturen erkennt man, dass bei den Erweiterungen das Modell jeweils in einer eigene Komponente modelliert wurde, welche mit einem Controller kommuniziert. Auf den ersten Blick müsste es also bei beiden Implementierungen nur das Modell und die Teile des Controllers verändert werden, welche mit dem Modell interagieren. Da beide Erweiterungen einen BPEL-Prozess einlesen können, welcher als XML-Dokument vorliegt, intern aber mit Java-Objekten arbeiten, muss es also jeweils eine Komponente in den Controllern geben, welche das XML-Dokument in ein Java-Objekt umwandelt und wieder zurück. Sollte also das Datenmodell, welches als Java-Objekt vorhanden ist, verändert werden, muss auch die jeweilige Übersetzerkomponente so angepasst werden, dass die XML-Elemente im richtigen Java-Objekt gespeichert werden. Es muss nicht nur die Kommunikationsschnittstelle der Controllers mit dem Datenmodell angepasst werden, sondern auch die Übersetzungskomponente.

Das Hauptziel dieser Arbeit ist es die Datenmodelle von *BPEL4REST* und der *RESTModelingExtension* zu vereinheitlichen. Dies bedeutet, dass die Implementationen der jeweiligen Datenmodelle ausgetauscht werden, durch eine Implementation, welche von beiden Erweiterungen verwendet werden kann. Dafür muss neben dem Datenmodell direkt auch noch die Kommunikation mit dem Datenmodell und die gerade erläuterte Übersetzungskomponente angepasst werden. Das vereinheitlichte Datenmodell soll so beschaffen sein, dass es ohne übermäßigen Aufwand in die beiden Erweiterungen eingebaut werden kann. Dafür muss es so konzipiert werden, dass die Änderungen am restlichen Code der Erweiterungen minimal bleiben. Es sollte auch so beschaffen sein, dass eine spätere Änderung des Datenmodells mit geringem Aufwand möglich ist. Auch sollte die Modellierung des Datenmodells der Spezifikation von *BPEL4REST* ähnlich sein, um weiteren Entwicklern den Einstieg zu vereinfachen.

Soll das Datenmodell der beiden Erweiterungen verändert werden, ergeben sich mehrere Möglichkeiten, wie dies geschehen könnte:

- Es werden beide bestehende Datenmodelle verworfen und ein **neues** Modell erstellt und implementiert.
- Die beiden bestehenden Datenmodelle werden zu einem einzigen Modell **verschmolzen** und die Implementierung der Erweiterungen auf dieses Modell angepasst.
- Ein Datenmodell wird verworfen und das **andere** Datenmodell wird in beiden Erweiterungen verwendet.
- Ein Datenmodell wird verworfen, dass andere Datenmodell wird an einigen Stellen angepasst und dann wird dieses **angepasste** Datenmodell in beiden Erweiterungen implementiert.

Die Auswahl auf welche Art das vereinheitlichte Datenmodell entstehen soll, basiert stark darauf, wie genau die bisherigen Datenmodelle modelliert und im-

plementiert wurden. Deshalb wird in diesem Kapitel eine genauere Analyse der Datenmodelle und ihrer Implementierung von *BPEL4REST* und der *RESTModelingExtension* dargelegt. Nachfolgend werden die Modelle und Implementierungen der Datenmodelle für beide Erweiterung vorgestellt. Dabei wird zum einen auf den Modellierungsaspekt eingegangen, zum anderen auch auf die direkte Implementierung und in wieweit diese einfach anpassbar ist. Es wird auch ein Vergleich der Modellierung und Implementierung des Modells zur Modelldarstellung in der Spezifikation von *BPEL4REST* gezogen. Am Ende dieses Kapitels wird dann ein Fazit getroffen, wie das vereinheitlichte Datenmodell nun modelliert und implementiert werden soll und warum so entschieden wurde.

5.1 Modellimplementation *BPEL4REST* im Detail

In den nachfolgenden Unterkapiteln wird die Modellierung und Implementierung des Datenmodells von *BPEL4REST* analysiert. Diese Analyse basiert auf dem Java-Quellcode welcher von Fischer während seiner Ausarbeitung erstellt hat. In Kapitel 3.3 wurde schon die grobe Architektur von *BPEL4REST* vorgestellt. Dabei fällt bei der Betrachtung von Abbildung 5 auf, dass das Modell von Fischer über Java-Schnittstellen (auch Java-Interfaces genannt) implementiert ist.

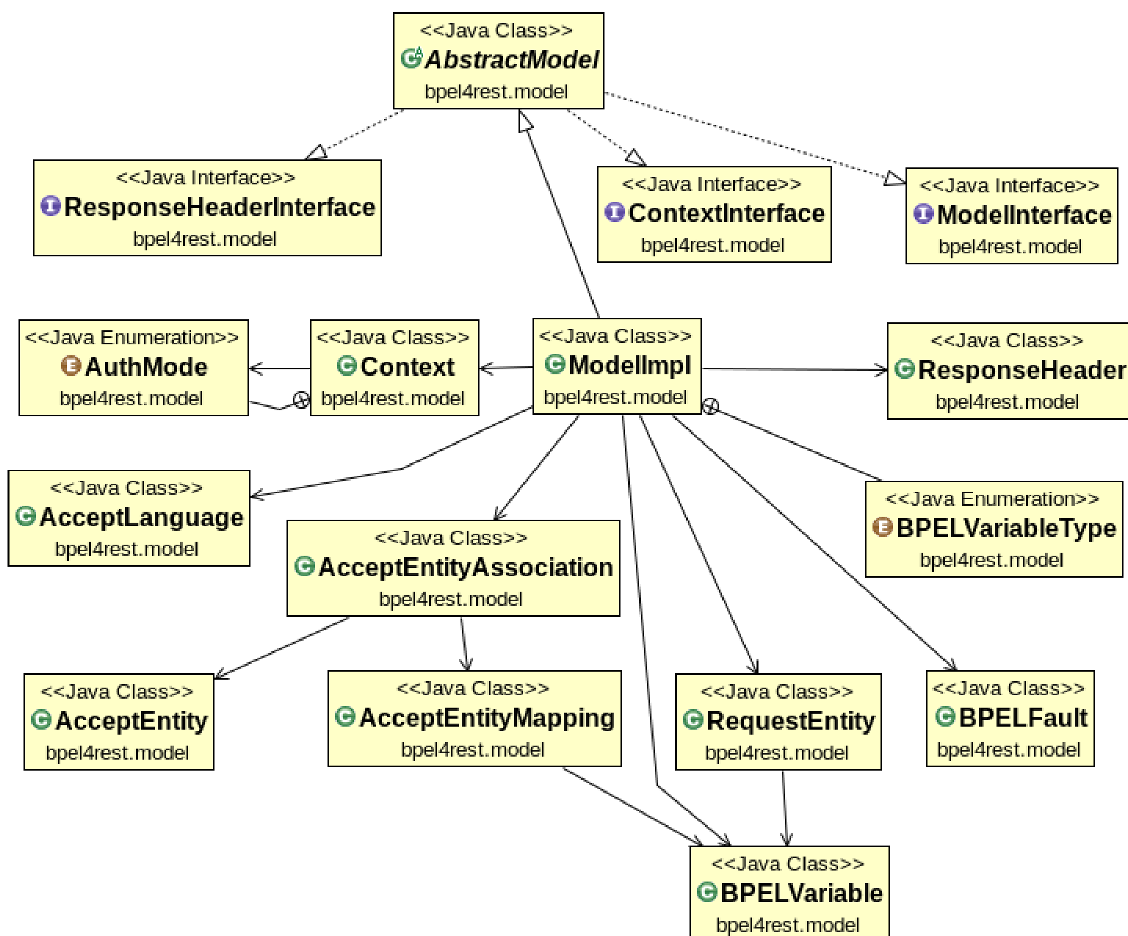

Abbildung 9: Klassendiagramm des Modellpakets von *BPEL4REST*

Vergleicht man die Architektur des *AbstracModels* mit der Struktur der XML-Elemente von *BPEL4REST*, ist es erkennbar, dass keine direkte Übertragung der XML-Elemente auf die Java-Interfaces geschehen ist. Aus der Beschreibung der Architektur ist zu entnehmen, dass das *ContextInterface* sich mit dem *<context>*-Element beschäftigt, was eine direkte Übertragung des XML-Elements auf die Java-Objekte darstellt. Das *ModelInterface* beinhaltet alle weiteren XML-Elemente der Erweiterung. nämlich das *<responseParameters>* und das *<requestParameters>*-Element. Im *ResponseHeaderInterface* wird das in Listing 7 beschriebene *<responseHeader>*-Element implementiert. Dieses Element taucht in den Verb-XML-Elementen allerdings gar nicht auf, sondern wird nur zum Speichern der Informationen verwendet, welche die Erweiterung bei der Ausführung der Web-Services des BPEL-Prozesses erhält.

Abbildung 9 zeigt ein vereinfachtes Klassendiagramm aller Java-Klassen und Java-Interfaces der Modellpaketes von *BPEL4REST*.

Die Java-Klasse *ModelImpl* implementiert das *AbstractModel* und enthält alle Informationen, die die XML-Elemente enthalten können. Das *<context>* und das *<responseHeader>*-Element, welche ein eigens Interface besitzen, werden als eigene Klassen implementiert, welche von *ModelImpl* verwendet werden. Eine Änderung des Modells würde also eine Neuimplementierung der Interfaces erfordern. Damit müssen die *ModelImpl*-Klasse und die von ihr verwendeten Klassen verändert oder ersetzt werden. Da diese Klassen allerdings nur Interfaces implementieren würde eine Veränderung der Implementierung geringe Änderung des restlichen Programmcodes nach sich ziehen. Abhängig der Tiefe der Änderung der Interfaces ist auch die Tiefe der Änderungen am restlichen Programmcode.

Die Betrachtung eines Ablaufdiagramms der Erweiterung (siehe Abbildung 10) zeigt, dass die Apache ODE die Erweiterung (über die Klasse *AbstractSyncExtensionOperation*) aufruft, wenn sie auf eine *<extensionActivity>* trifft, welche in *BPEL4REST* definiert wurden.

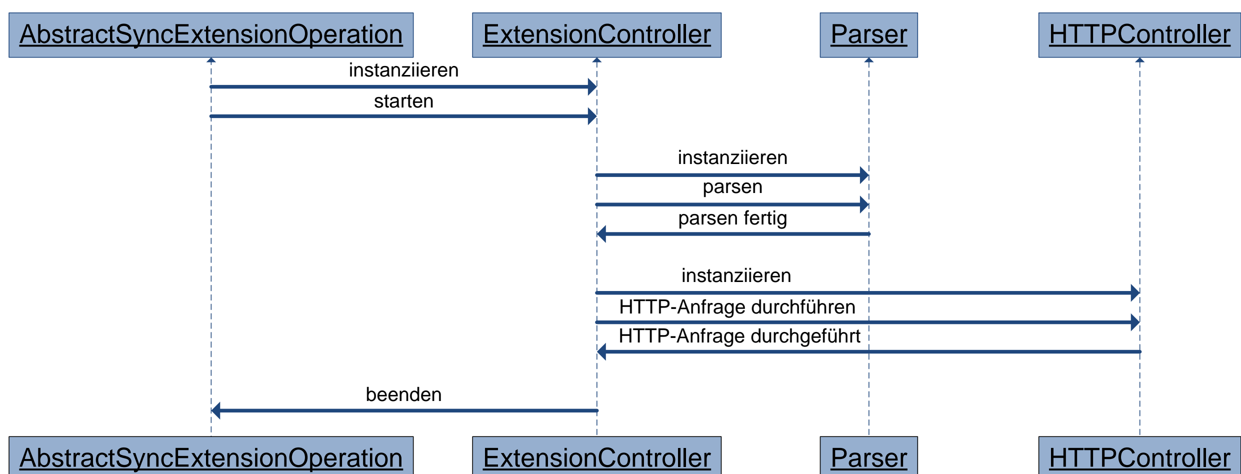

Abbildung 10: Sequenzdiagramm eines *BPEL4REST*-Aufrufes nach Fischer [12]

Abhängig vom erkannten HTTP-Verb wird die passende Implementierung aufgerufen, welcher den passende *ExtensionController* der Erweiterung startet. Damit wird das Modell erstellt und der *Parser* gestartet, welcher die Werte der XML-Elemente des BPEL-Prozesses in das Modell überträgt. Wenn also die Implementierung der Interfaces des Modells verändert werden, müssen auch die Parser so angepasst werden, dass sie mit der Neuimplementierung der Interfaces arbeiten können.

Da so folglich nur die Interfaces und die Parser angepasst werden müssten, bietet sich die Modellimplementierung von *BPEL4REST* dazu an ausgetauscht zu werden. Von diesem Aspekt betrachtet ist der Aufbau des Modells kein Hindernis dafür, dass vereinheitliche Datenmodell auf der Struktur dieses Datenmodell aufzubauen, da es sehr modifizierbar ist.

Weitere Aspekte für das vereinheitliche Datenmodell sind die Nähe zur Spezifikation von *BPEL4REST* und wie einfach die Datenstruktur modifiziert werden kann. Um diese Aspekte betrachten zu können, wird die Datenstruktur von *BPEL4REST* in den folgenden Unterkapiteln für die Parser, die Interfaces und deren Implementierungen analysiert.

5.1.1 ContextParser und ExtensionParser

Wenn die Apache ODE im BPEL-Prozess ein *<extensionActivity>*-Element erkennt und dieses *BPEL4REST* zuordnet, wird die Erweiterung von der ODE aufgerufen. Dabei wird der Erweiterung die aktuelle Instanz des BPEL-Prozesses sowie das ganze *<extensionActivity>*-Element von der ODE übergeben. Für die *<extensionActivity>* wird von *BPEL4REST* ein *ExtensionController* erstellt, welcher das Modell initialisiert. Nachdem das Modell erstellt wurde, wird für dieses Modell ein *ExtensionParser* erstellt, welcher die XML-Elemente und die Instanz des BPEL-Prozesses betrachten und die zugehörigen Datenfelder im Modell damit füllen.

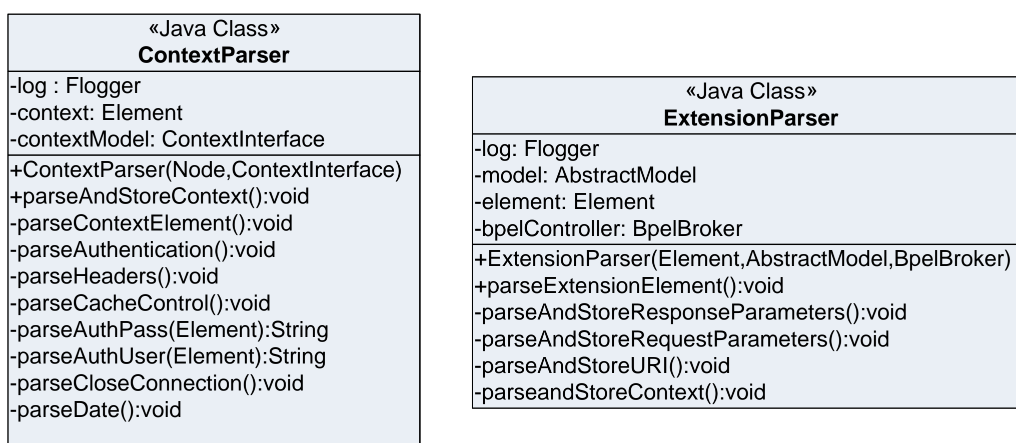

Abbildung 11: Klassendiagramme der Parserklassen von *BPEL4REST*

Abbildung 11 zeigt die Klassendiagramme mit den Methoden und Klassenvariablen

für die Klassen *ContextParser* und *ExtensionParser*.

Die Methoden des *ExtensionParsers* iterieren über die übergebenen XML-Elemente und überprüfen, abhängig von der Methode, ob die XML-Elemente mit einem bestimmten Element der Spezifikation übereinstimmen. Ist dies der Fall werden Werte der Attribute des XML-Elements, oder im Falle einer XML-Variable die Werte der BPEL-Instanz, an die passenden *setter*-Methoden der Interfaces übergeben und damit die Werte in das Modell übertragen. Die Methode *parseAndStoreURI* extrahiert beispielsweise vom HTTP-Verb-Element die Werte von *host* und *path* und erstellt damit die URI des Modells. Für das *<context>*-Element existiert ein eigener Parser, der die Werte in das *Context*-Objekt überträgt.

Wenn bei der Vereinheitlichung des Datenmodells die Interfaces von *BPEL4REST* nicht verändert werden, sondern nur deren Implementierungen, muss an den Parserklassen nichts verändert werden. Sollte allerdings mehr als nur die Implementierung der Interfaces verändert werden, müssen auch die Parserklassen auf das neue Modell angepasst werden.

5.1.2 ModelInterface und ModelImpl

Die Klasse *ModelImpl* implementiert das *AbstractModel*. Damit hält die Klasse fast alle Informationen der Erweiterung. Für jede *<extensionActivity>* wird vom *ExtensionController* ein eigenes Modell erstellt. Dieses beinhaltet dann sämtliche Informationen des BPEL-Prozesses für dieses XML-Element und dessen Kinder. Das Modell-Objekt enthält alle Informationen der HTTP-Verb-Elemente der *BPEL4REST*-Spezifikation, als auch die Informationen des *<responseHeader>*-Elements, welche während der Ausführung der Web-Services erstellt werden können. In dem Modell-Objekt sind die Informationen dann in verschiedenen Variablen gespeichert. Die Art der Variablen reichen von einfachen Datentypen die im Modell-Objekt gehalten werden, bis hin zu vollständigen Klassen, deren Objekte für das Modell erstellt werden. Eine Auflistung dieser Klassen, welche im Modell-Objekt verwendet werden ist in Abbildung 9 zu finden.

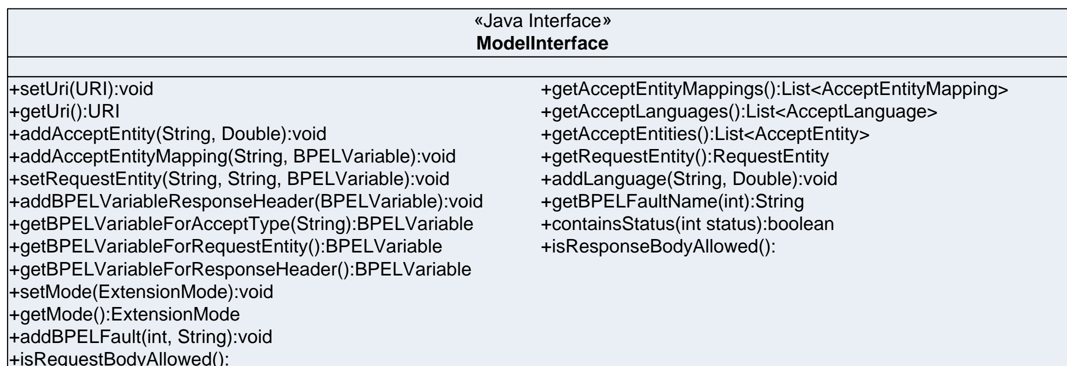

Abbildung 12: Klassendiagramm des ModelInterfaces von *BPEL4REST*

In Abbildung 12 sind die Methoden des *ModelInterface* als Klassendiagramm abgebildet. Durch die Übersicht der Methoden ist es ersichtlich, dass das die öffentli-

chen Methoden der *ModelImpl*-Klasse nur *getter* und *setter*-Methoden sind, welche den Zugriff auf die Attribute des Modells erlauben.

Obwohl die *ModelImpl*-Klasse viele öffentliche Methoden beinhaltet, welche bei einer Nichtveränderung der Interfaces gleich bleiben würden, sind die Attribute des Modells sehr stark an die Struktur der Interfaces gebunden. Sollte also das Datenmodell ausgetauscht werden ohne die Interface-Klassen zu ändern, müsste das vereinheitlichte Datenmodell sich von der Struktur an den Interface-Klassen orientieren oder eine Struktur besitzen, welches es ermöglicht auf dessen Attribute so zugreifen zu können, damit sie in eine Form umgewandelt werden können, die mit den Interfaces konform sind.

Vergleicht man den strukturellen Aufbau des Datenmodells mit der Spezifikation der XML-Elemente von *BPEL4REST*, erkennt man, dass die Gruppierungen der XML-Elemente in der Implementierung nicht eingehalten werden. Während das *<context>*-Element in eine weitere Klasse ausgelagert wurde, sind die Kind-Elemente der *<requestParameters>* und *<responseParameters>*-Elemente auf verschiedene Datentypen im Modell verteilt und nicht sauber gekapselt. Sollten also Änderungen am Modell vorgenommen werden, würde dies zu einer aufwändigen Arbeit werden, da die passenden Attribute erst gefunden werden müssen und es dann bei der Änderung zu Problemen mit Abhängigkeiten und sich durchschleifenden Fehlern kommen kann.

Das Modell von *BPEL4REST* ist durch seine Interfaces dazu prädestiniert, austauschbar zu sein. Auch der verstreute Aufbau der Attribute und die Abweichung zum Aufbau der XML-Struktur führen dazu, dass das bestehende Modell von *BPEL4REST* innerhalb des vereinheitlichen Datenmodells keine Rolle spielen wird.

5.1.3 ContextInterface und Context

Das Interface *ContextInterface* und die Klasse *Context* sollen die Informationen des XML-Elements *<context>* beinhalten. Wie allerdings schon von Kalach festgestellt wurde, befinden sich in der originalen Spezifikation Elemente im *<context>*-Element, die *<conditionals>*-Elemente, welche ihrer Funktion nach unter das *<requestParameters>*-Element gehören. Da diese Änderung an der Spezifikation erst nach der Implementierung von *BPEL4REST* durchgeführt wurde, sind in dem Datenmodell diese Elemente noch in Klasse *Context* zu finden. Da das vereinheitlichte Datenmodell sich streng an der Struktur der XML-Elemente orientieren soll, müssen dort diese Attribute an eine andere Stelle verschoben werden.

Das Interface *ContextInterface* selbst wird in der Klasse *ModelImpl* implementiert. Allerdings wird dort nur eine Objekt der Klasse *Context* erstellt, auf welche die Methoden des Interfaces zugreifen. Die Speicherung der Daten der Kind-Elemente des *<context>*-Elements und die auf diesen basierende Logik findet dann direkt im *Context*-Objekt statt.

Auch Attribute der *Context*-Klasse sind stark an den Rückgabewerte der Methoden des *ContextInterfaces* orientiert. Da sie aber auch den Wertetypen der XML-Elemente entsprechen, ist hier das vereinheitliche Datenmodell eh an definierte Typen gebunden, um die Information des BPEL-Prozesses zu speichern. Das vereinheitliche Datenmodell muss diese Datentypen allerdings in solch einer Art zu Verfügung stellen, dass die Methoden des *ContextInterfaces* darauf zugreifen können. Auch sollte entweder das Interface auf die *<requestParameters>*-Informationen des Modells zu-

greifen können, oder die Implementierung von *BPEL4REST* muss so umstrukturiert werden, dass die *<conditionals>*-Elemente auf eine andere Weise der Erweiterung zu Verfügung gestellt werden. Abhängig vom verwendeten vereinheitlichen Datenmodell könnte die *Context*-Klasse dadurch wegfallen.

5.1.4 ResponseHeaderInterface und ResponseHeader

Das *ResponseHeaderInterface* und die Klasse *ResponseHeader* sollen die Informationen des *<responseHeader>*-Elements abspeichern und verwalten. Das *ResponseHeaderInterface* wird auch in der Klasse *ModelImpl* implementiert. Auch hierbei wird dort nur eine Objekt der Klasse *ResponseHeader* erstellt, auf welche die Methoden des *ResponseHeaderInterface* zugreifen. Die Speicherung der Daten der Kind-Elemente des *<responseHeader>*-Elements und die auf diesen basierende Logik findet dann im *ResponseHeader*-Objekt statt.

Allerdings speichert das *<responseHeader>*-Element nur die HTTP-Header-Informationen, welche bei der Ausführung eines Web-Services als Antwort zurückkommen. Es ist kein direkter Teil in der Modellierung von BPEL-Prozessen und wird in der *RESTModelingExtension* direkt auch nicht verwendet. Das *<responseHeader>*-Element ist damit zwar ein wichtiger Teil in der Funktionalität von *BPEL4REST* hat aber in dem vereinheitlichen Datenmodell keine Verwendung. Damit sollte es bei der Implementierung des vereinheitlichen Datenmodells extrahiert werden und an einer anderen Stelle in *BPEL4REST* eingesetzt werden.

5.2 Modellimplementation *RESTModelingExtension* im Detail

In diesem Unterkapitel wird die Modellierung und Implementierung des Datenmodells der *RestModelingExtension* betrachtet. Diese Analyse basiert auf dem EMF-basierenden Modell und dessen Java-Quellcode welche von Kalach während ihrer Ausarbeitung modelliert und erstellt wurden. Im Eclipse BPEL Designer lassen sich BPEL-Prozesse sowohl in graphischer als auch in textueller Form betrachten und bearbeiten. Dies bedeutet, dass der BPEL-Designer immer zwei Modelle des BPEL-Prozesses besitzt. Das graphische Modell (repräsentiert durch ein Eclipse Modeling Frameworks (EMF) Modell) und das textuelle Modell (repräsentiert durch ein XML/DOM-Baum) müssen dabei immer synchronisiert werden.

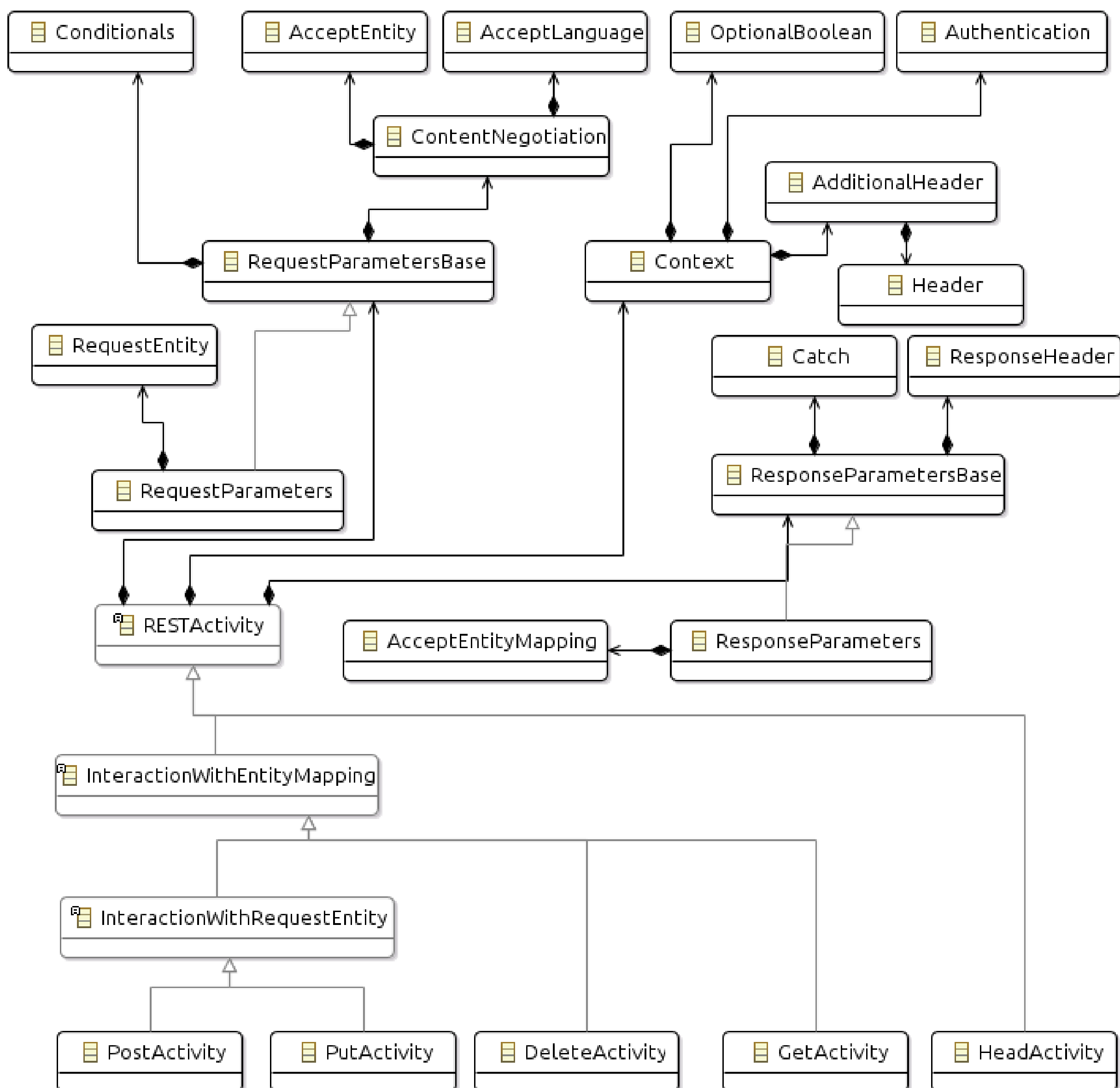

Abbildung 13: Klassendiagramme der Parserklassen von *BPEL4REST*

Für die *RESTModelingExtension* wurde das graphische Modells mittels EMF erstellt und der XML/DOM-Baum durch den Texteditor des Eclipse BPEL Designers. Das EMF-Modell soll dabei die *BPEL4REST*-Spezifikation repräsentieren und auf dem Meta-Modell für REST Extension Activity [13] basieren. Dadurch soll die Datenpersistenz gewährleistet sein und bei Änderungen im Modell die passenden Benachrichtigungen sofort zu erhalten.

In Kapitel 4.2 wurde schon die grobe Architektur der *RESTModelingExtension* vorgestellt. Abbildung 13 zeigt nun ein vereinfachte Klassendiagramm des Datenmodells der *RESTModelingExtension*. Vergleicht man nun den Aufbau des EMF-Modells der *RESTModelingExtension* mit der Struktur der XML-Elemente von *BPEL4REST*, so erkennt man, dass für jedes XML-Element der *BPEL4REST*-Erweiterung im Modell der *RESTModelingExtension* eine eigene Klasse existiert, welche die Informationen über alle Kind-Elemente und Attribute beinhaltet.

So existiert, abhängig vom verwendeten HTTP-Verb, ein *RESTActivity*-Objekt, in welchem sämtliche Informationen der XML-Elemente gespeichert werden. Dabei erweitern alle *RestActivity*-Klassen die Klasse *ExtensionActivity* des BPEL Designers. Allerdings werden komplexere HTTP-Verben wie *PUT* oder *POST* durch weitere Klassen in der Vererbungshierarchie beeinflusst, als ein einfaches *HEAD*. Durch die Verwendung dieser Struktur entspricht dieses Datenmodell ziemlich genau der Struktur der Spezifikation der XML-Elemente von *BPEL4REST*. Dies ist ein Indikator dafür, das Modell der *RESTModelingExtension* für das vereinheitliche Datenmodell zu verwenden. Da das Datenmodell auf einem graphischen Modell basiert, ist es für Entwickler einfach zu erweitern. Nach eine Änderung im graphischen Modell werden die Quell-Code-Klassen automatisch vom Framework erstellt. Die Entwickler müssen dann nur noch die Inkonsistenz des neuen Modells mit dem restlichen Code überprüfen, sich aber nicht um manuelle Änderungen im Modell-Code kümmern. Auch dies ist ein Indikator dafür, dass Datenmodell der *RESTModelingExtension* für das einheitliche Datenmodell zu verwenden. Für die Austauschbarkeit des Datenmodells mit geringen Änderungen im restlichen Code, betrachte man die Architekturbeschreibung in Abbildung 8. Darin erkennt man, dass das *Model* nur mit dem *Controller* kommuniziert. Nach Kalach greift der *Controller* nur über die Klassen *ModelPackage* und *ModelFactory* auf das *Model* zu. Beide Klassen sind für jedes *Model*-Objekt einzigartig. Wollte man das Datenmodell für die *RESTModelingExtension* ersetzen, müsste das neue Modell auch diese Klassen implementieren, damit die Erweiterung weiter auf das Modell zugreifen kann. Auch besitzt die Erweiterung Klassen, welche die synchronisieren zwischen dem EMF-Modell und dem XML/DOM-Modell ermöglichen. Dadurch besitzt auch die *RESTModelingExtension* eine Klasse, welche die XML-Elemente Java-Objekte für das Datenmodell übersetzt. Eine Ersetzung des Modells würde damit auch bedeuten, dass die Helfer-Klassen der Synchronisierung auf das neue Datenmodell angepasst werden müssen.

5.3 Entscheidung zum Entwurf des einheitlichen Datenmodells

Nachdem in den vorherigen Unterkapiteln die verschiedenen Datenmodelle von *BPEL4REST* und der *RESTModelingExtension* betrachtet und analysiert wurden, soll nun in diesem Unterkapitel eine Entscheidung getroffen werden, wie das vereinheitliche Datenmodell entworfen und implementiert werden soll.

Die Analyse des Datenmodells von *BPEL4REST* hatte ergeben, dass das Modell durch Interfaces repräsentiert wird, deren Implementierungen einfach auszutauschen sind. Da die Interfaces hauptsächlich aus *getter* und *setter*-Methoden bestehen, sollten die *getter*-Methoden der Interface unverändert bleiben, da sonst große Teile des restlichen Programm-Codes verändert werden müssten. Die *setter*-Methoden werden allerdings nur von den *Parser*-Klassen und einer weiteren Klasse aufgerufen. Für sie gilt das Änderungsverbot also nicht, wenn für das vereinheitliche Datenmodell neue *Parser*-Klassen erstellt werden sollen. Auch sollte im neuen Modell das *ResponseHeaderInterface* und der *ResponseHeader* aus dem Modell extrahiert werden und an eine anderen Stelle neu implementiert. Die Struktur des Modells unterscheidet sich auch erheblich von der Struktur der XML-Elemente. Dies erschwert das Verständnis des Programm-Codes für Entwickler erheblich. Auch ist die Implementierung und Datenhaltung der Information im Modell sehr verstreut. Sollen an der Modellimplementierung jemals eine Änderung vorgenommen werden, ist es sehr aufwändig die passenden Code-Abschnitte zu finden und anzupassen.

Aufgrund dieser Eigenschaften wird für diese Arbeit das bestehende Datenmodell von *BPEL4REST* verworfen und durch ein neues ersetzt. Dabei sollen die *getter*-Methoden der Interfaces unangetastet bleiben um die Änderungen auf den Rest der Codes zu minimieren. Abhängig des neuen Datenmodells können die *setter*-Methoden und damit auch die *Parser*-Klassen angepasst werden.

Bei der *RESTModelingExtension* wird das Datenmodell in zwei Arten gespeichert. Einmal als XML/DOM-Dokument und einmal als EMF-Modell. Beide Darstellungen werden von der Erweiterung synchronisiert. Dadurch besitzt die Erweiterung einen Klasse, welche aus XML-Elementen passende Java-Objekte für dieses Modell erstellt. Das EMF-Modell wurde nach der *BPEL4REST*-Spezifikation entworfen und entspricht sehr stark dem Aufbau der XML-Elemente. Dadurch wird es Entwicklern vereinfacht von der Spezifikation auf die passenden Java-Klassen zu kommen. Durch die Darstellung des Modells als EMF-Modell kann es auch einfacher verändert werden, da hauptsächliche graphische Änderungen von Nöten sind und kaum direkte Eingriffe in den Quellcode des Modells. Das die *Model*-Komponente nur von der *Controller*-Komponente über zwei bestimmte Klasse aufgerufen wird, könnte das Modell auch ausgetauscht werden, sofern die Funktionalität dieser beiden Klassen erhalten bleibt.

Da dieses Modell den Indikatoren entspricht, die in dieser Arbeit für ein einheitliches Datenmodell verwendet werden sollen, soll für diese Arbeit das Datenmodell der *RESTModelingExtension* erhalten bleiben. Falls kleinere Änderungen an dem Modell nötig sind, sollen diese möglich sein.

Dadurch, dass das Datenmodell der *RESTModelingExtension* erhalten bleibt, fällt die Alternative die bestehenden Datenmodelle zu verwerfen und ein ganz **neues** Modell zu kreieren weg. Auch wäre dies mit übermäßig viel Arbeit an beiden Erweiterungen verbunden gewesen, was den Zielen dieser Arbeit widerspricht. Da die beiden Datenmodelle sehr unterschiedlich sind und das Modell von *BPEL4REST* nicht den gewünschten Kriterien eines vereinheitlichen Datenmodells entspricht, ist auch eine **Verschmelzung** der Datenmodelle nicht möglich. Auch kann Aufgrund der unterschiedlichen Behandlung von Variablen in BPEL zwischen dem BPEL Designer und der Apache ODE (siehe hierzu Kapitel 6.2) das Datenmodell der *RESTModelingExtension* nicht direkt in *BPEL4REST* **übernommen** werden. Es sind kleinere *Änderungen* am Modell vonnöten, damit es von beiden Erweiterungen verwendet werden kann.

Es soll in dieser Arbeit also das EMF-Modell der *RESTModelingExtension* angepasst und dann in *BPEL4REST* implementiert werden. Dabei wird an der Funktionalität der *RESTModelingExtension* nichts verändert. Da in der *RESTModelingExtension* sämtliche Informationen in einem *RESTActivity*-Element gespeichert sind, muss das Modell von *REST4BPEL* darauf angepasst werden. Dabei müssen auch die *Parser* von *BPEL4REST* an die *Parser* der *RESTModelingExtension* angepasst werden, da diese aus einem XML-Element ein *RESTActivity*-Objekt erstellen.

6 Entwurf und Implementation des vereinheitlichen Datenmodells

Nachdem in Kapitel 5 die bestehenden Datenmodelle analysiert wurden, wurde auch ein vereinheitlichtes Datenmodell entworfen. In diesem Kapitel soll dieser Entwurf und die Implementierungen des neuen Datenmodells in die Erweiterungen beschrieben werden.

Das Datenmodell der *RESTModelingExtension* soll zuerst angepasst werden um auch in der Apache ODE zu funktionieren. Danach soll das Datenmodell von *BPEL4REST* verworfen werden und durch das angepasste Datenmodell der *RESTModelingExtension* ersetzt werden. Da das neue Datenmodell einen eigene Klasse mitbringt, welche die XML-Elemente in Java-Objekte übersetzt, wird ein großer Anteil der *setter*-Methoden nutzlos und damit verworfen. Die *getter*-Methoden bleiben erhalten um die Eingriffe in die Erweiterung nicht all zu sehr zu vertiefen. Die *Parser*-Klassen von *BPEL4REST* müssen ebenfalls an die neuen *Parser*-Klassen angepasst werden. Auch soll das *ResponseHeaderInterface* und die *ResponseHeader*-Klasse aus dem Modell heraus an eine andere Stelle verschoben werden.

Die Struktur der Implementierung des Modells der *RESTModelingExtension* erlaubt es, das Modell und die benötigten Hilfsklassen als Java-Bibliothek zu exportieren. Um das Modell dann in *BPEL4REST* verwenden zu können, müssen neben dem Modell noch einige Java-Bibliotheken importiert werden, welche von der *RESTModelingExtension* verwendet werden.

In den nachfolgenden Unterkapitel werden die Änderungen an den Erweiterungen genauer beschrieben.

6.1 Entwicklungs- und Testumgebung

In diesem Unterkapitel werden die Werkzeuge und Technologien kurz beschrieben, welche bei der Entwicklung und dem Test des vereinheitlichen Datenmodells eingesetzt worden sind.

6.1.1 Eclipse

Eclipse [9] ist eine OpenSource Entwicklungsumgebung, für deren Entwicklung die Eclipse Foundation verantwortlich ist. Für diese Arbeit wurde die Version *Eclipse IDE for Java EE Developers Mars 4.5 Service Release 1* verwendet. Als weitere Installationspakete wurden für die Entwicklung noch das *Eclipse Modeling Framework* [2] und der *Eclipse BPEL Designer* [1] verwendet.

6.1.2 Apache Tomcat

Apache Tomcat [8] ist eine OpenSource Implementierung der Java Servlet und JavaServerPages Technologien, welche von der Apache Software Foundation entwickelt und betreut wird. In *BPEL4REST* wir der Apache Tomcat benötigt um als Servlet Container für die Apache ODE zu fungieren. Für diese Arbeit wurde die Version 7.0.65 verwendet.

6.1.3 Apache ODE

Die Apache ODE (Orchestration Director Engine) [7] ist eine BPEL-Engine. Sie wird wie der Apache Tomcat von der Apache Software Foundation betreut und entwickelt. Die Aufgaben der ODE sind die Kommunikation mit Webservices, das Senden und Empfangen von Nachrichten und die Datenmanipulation gemäß der Beschreibung durch einen BPEL-Prozess.

Für *BPEL4REST* muss die Apache ODE mit dier WAR Distribution "experimental branch 2.0-beta8" verwendet werden, da nur in dieser Distribution die ODE um *<extensionActivities>* erweitert werden kann.

6.2 Variablen in der Apache ODE und dem Eclipse BPEL Designer

In diesem Unterkapitel werden grob die Unterschiede in der Auflösung von Variablen zwischen der Apache ODE und dem Eclipse BPEL Designer erklärt. Diese unterschiedliche Behandlung von Variablen ist der Grund, warum das Datenmodell der *RESTModelingExtension* nicht direkt in *BPEL4REST* verwendet werden kann.

Der Hauptunterschied in der Variablenbehandlung liegt darin, dass im Eclipse BPEL Designer das Prozessmodell des BPEL-Prozesses betrachtet wird, während die Apache ODE die Prozessinstanzen des BPEL-Prozesses behandelt. Spricht man im Eclipse BPEL Designer also von Variablen, werden damit deren Variablendefinitionen betrachtet. In der Apache ODE wird das Prozessmodell aber ausgeführt und erstellt damit einen Instanz des BPEL-Prozesses. Dadurch werden die Variablen von einer Variablendefinition zu einer Speicherstelle mit konkreten Daten. Für die ODE werden diese konkreten Daten verwendet und weniger die dahinterstehenden Typendefinitionen.

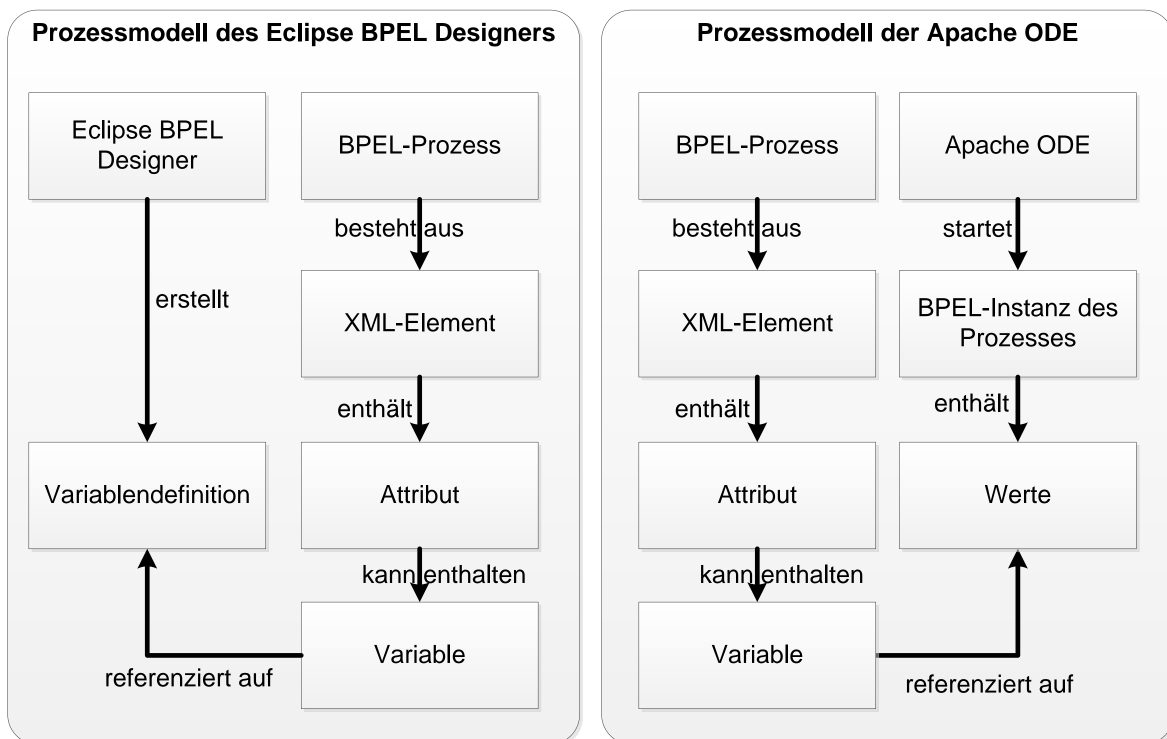

Abbildung 14: Vereinfachte Darstellung der Variablenhaltung in der *Apache ODE* und dem *Eclipse BPEL Designer*

Abbildung 14 stellt schematisch dar, wo in einem BPEL-Prozess Variablen auftreten können und wie sie jeweils von der Apache ODE und dem Eclipse BPEL Designer gehalten werden. Ein BPEL-Prozess besteht aus XML-Elementen. Diese können Attribute enthalten. In den Attributen sind die Namen der Variablen enthalten.

Über diese Namen sind dann die Definitionen der Variablen im Prozessmodell zu finden. Wird nun ein BPEL-Prozess ausgeführt, wird aus dem Prozess eine Instanz erzeugt, welche neben dem Modell noch zusätzliche Daten beinhaltet. Beispielsweise enthält eine Variable neben der Variablendefinition im Prozessmodell nun auch einen konkreten Wert.

Soll nun ein BPEL-Prozess, dessen Prozessmodell mit dem Eclipse BPEL Designer erstellt wurde, in der Apache ODE ausgeführt werden, kann dies zu Problemen führen. Wird nun das Modell der *RESTModelingExtension* in *BPEL4REST* verwendet, werden Variablen die im Prozessmodell des Eclipse BPEL Designers erstellt wurden in der Apache ODE ausgeführt. Das ursprüngliche Modell versucht nun für die Variablen Werte zu finden, was allerdings nicht möglich ist, da auf Variablendefinitionen verwiesen wird, die in einem anderen Prozessmodell definiert wurden. Da der Apache ODE nur der BPEL-Prozess, aber nicht das Prozessmodell des Eclipse BPEL Designers übergeben wurde, können die Variablen nicht aufgelöst werden.

Um das Datenmodell der *RESTModelingExtension* dennoch unter der Apache ODE zu verwenden, müssen an der Klasse, welche die *BPEL-Variablen* auflöst, Veränderungen vorgenommen werden.

6.3 Änderungen am Modell der *RESTModelingExtension*

In diesem Unterkapitel werden die Änderungen der *RESTModelingExtension* beschrieben, welche durch die Nutzung des vereinheitlichen Datenmodells entstehen.

Dadurch, dass das Datenmodell der *RESTModelingExtension* erhalten bleibt müssten an dessen Implementierung eigentlich nichts geändert werden. Da allerdings die Variablenhaltung zwischen dem Eclipse BPEL Designer und der Apache ODE unterschiedlich ist, wurde der *BPELVariableResolver* der *RESTModelingExtension* angepasst.

6.3.1 Änderung am RESTActivityDeserializer

In originalen Modell wurde der *BPELVariableResolver* des Eclipse BPEL Designers verwendet. Da dieser unter der Apache ODE nicht funktioniert, wurde der Klasse, die den *BPELVariableResolver* verwendet, eine Methode hinzugefügt um den *VariableResolser* durch eine anderen zu ersetzen. Da diese Methode explizit aufgerufen wird, wird dadurch in der normalen Verwendung des Modells in der *RESTModelingExtension* noch der *BPELVariableResolver* des Eclipse BPEL Designers verwendet.

Listing 8 Änderung des RESTActivityDeserializer in der *RESTModelingExtension*

```
private VariableResolver VARIABLE_RESOLVER = new
   BPELVariableResolver();

public void setVariableResolver(VariableResolver vr){
       this.VARIABLE_RESOLVER = vr;
}
```

Listing 8 zeigt den Java-Code, welcher die einzige Änderung in der *RESTModelingExtension* beinhaltet und den *BPELVariableResolver* überschreiben kann.

Soll das Modell der *RESTModelingExtension* jetzt unter der Apache ODE in *BPEL4REST* verwendet werden, muss dort ein neuer *VariableResolver* definiert werden, welcher auch unter der Apache ODE funktioniert. Da die Variablenauflösung in der Apache ODE durch den Namen der Variable funktioniert, muss der neue *VariableResolver* nur eine *BPEL-Variable* mit dem passenden Namen anlegen. Der Inhalt ist dann zwar leer, aber dieser würde von der Apache ODE nicht gelesen werden.

Listing 9 zeigt den Java-Code, welcher in *BPEL4REST* im neuen *VariableResolver* verwendet wird um eine *BPEL-Variable* mit passendem Namen zu erstellen.

Listing 9 BPEL Variable Resolver in *BPEL4REST*

```
public Variable getVariable(EObject arg0, String arg1){
       Variable x = BPELFactory.eINSTANCE.createVariable();
       x.setName(arg1);
       return x;
}
```

6.4 Änderungen am Modell *BPEL4REST*

In diesem Unterkapitel werden Änderungen in *BPEL4REST* beschrieben, welche entstanden, als das vereinheitlichte Datenmodell implementiert wurde.

Das Datenmodell der *RESTModelingExtension* erstellt für jede *<extensionActivity>* ein *RESTActivity*-Objekt, welches alle Informationen des XML-Element speichert. Dafür besitzt das Modell auch eine eigene *Parser*-Klasse, die die Informationen aus den XML-Elemente in das *RESTActivity*-Objekt überträgt. Damit müssen schon einmal die *Parser*-Klassen in *BPEL4REST* verändert werden. Da das neue Datenmodell sich völlig auf das *RESTActivity*-Objekt konzentriert, *BPEL4REST* jedoch mit diesem Objekt nicht arbeiten kann, müssen die Methoden des *AbstractModels* die Informationen aus dem Objekt extrahieren und dann in der Form wiedergeben, die die Interface vorgeben. Dabei können aber fast alle *setter*-Methoden der Interfaces entfernt werden, da das Modell nun durch die *Parser*-Klasse erstellt wird.

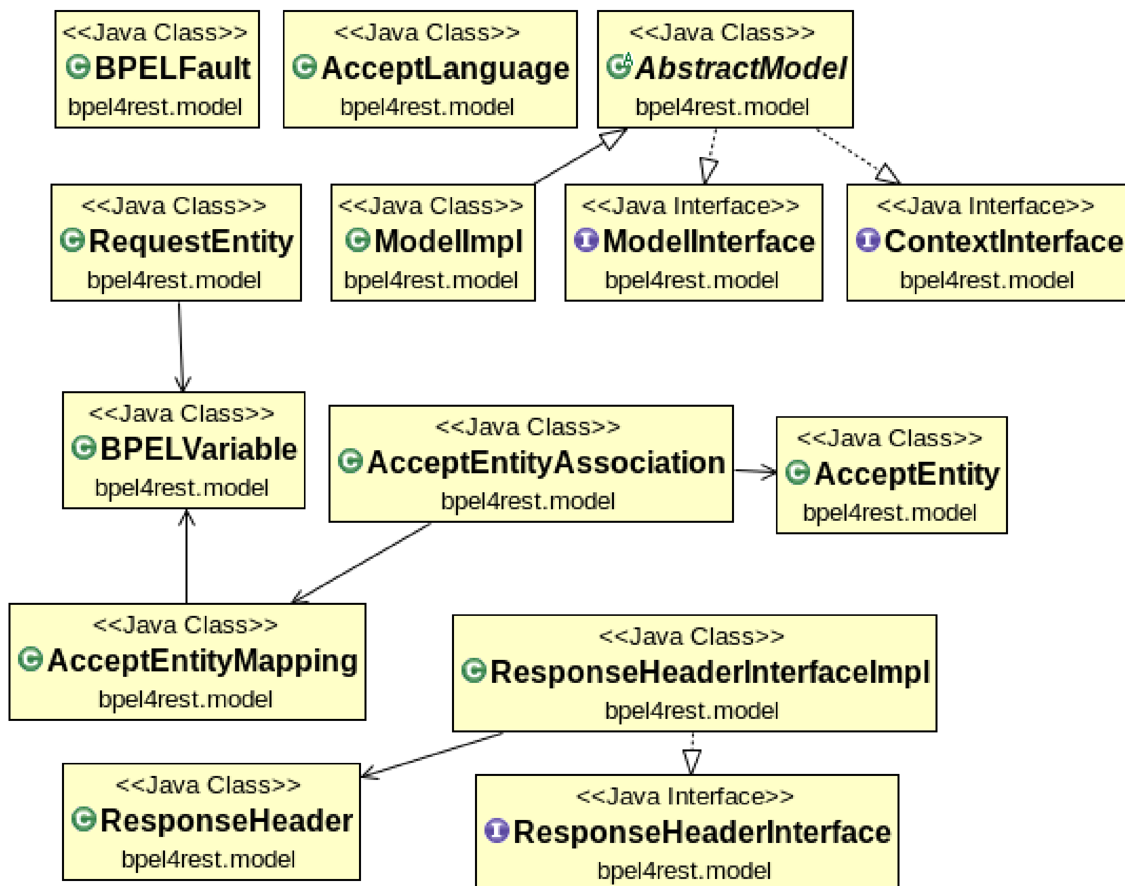

Abbildung 15: Klassendiagramm des vereinheitlichen Modells von *BPEL4REST*

In Abbildung 15 ist das Klassendiagramm des Modell-Pakets von *BPEL4REST* nach der Implementierung des vereinheitlichen Datenmodells zu sehen. Vergleicht man es mit Abbildung 9 wird erkenntlich, dass einige Klasse aus dem Modell verschwunden sind. Diese werden nun im *RESTActivity*-Objekt direkt gehandhabt.

In den nachfolgenden Unterkapitel wird genauer auf die Änderungen der verschiedenen Interfaces und Parser eingegangen.

6.4.1 ModelInteface und ModelImpl

Auch nach der Verwendung des neuen Datenmodells sind die Methoden des *ModelInterfaces* noch vorhanden. Es wurden allein die *setter*-Methoden entfernt, da das Modell nun vom *ExtensionParser* erstellt wird. Die Klasse *ModelImpl* implementiert immer noch das *AbstractModel*, welches weiterhin vom *ExtensionController* erstellt wird.

Allerdings fällt der Klasse *ModelImpl* nun eine andere Aufgabe zu als bei der Verwendung des alten Modells. Hat *ModelImpl* früher die Informationen in seinen Attributen abgespeichert und an den Rest der Erweiterung weitergegeben, ist seine Aufgabe nun das *RESTActivity*-Objekt des *ExtensionParsers* zu speichern und dessen Informationen über das Interface an die restliche Erweiterung weiterzugeben.

In Listing 10 sind die Attribute der Klasse *ModelImpl* zu sehen, bevor das vereinheitliche Datenmodell in *BPEL4REST* implementiert wurde.

Listing 10 Modell-Variablen in **BPEL4REST** vor der Vereinheitlichung des Datenmodells

```
public class ModelImpl extends AbstractModel
{
	private ExtensionMode mode = null;
	private URI uri = null;
	private Context context = null;
	private ResponseHeader resHeader = null;
	private HashMap<String, AcceptEntityAssociation>
	   acceptMappings = new HashMap<String,
	   AcceptEntityAssociation>();
	private HashMap<String, BPELVariable> bpelVariables = new
	   HashMap<String, BPELVariable>();
	private HashMap<Integer, BPELFault> bpelFaults = new
	   HashMap<Integer, BPELFault>();
	private boolean isRequestBodyAllowed;
	private boolean isResponseBodyAllowed;
	private List<AcceptLanguage> languages = new
	   ArrayList<AcceptLanguage>();
	private RequestEntity requestEntity = null;
	...
}
```

In Listing 11 sind die Attribute der Klasse *ModelImpl* zu sehen, nachdem das vereinheitliche Datenmodell in *BPEL4REST* implementiert wurde.

Listing 11 Modell-Variablen in **BPEL4REST** nach der Vereinheitlichung des Datenmodells

```
public class ModelImpl extends AbstractModel
{
	RESTActivity RestActivity;
	...
}
```

Vergleicht man die beiden Listings wird klar, dass die *RESTActivity* im Mittelpunk des neuen Datenmodells steht. Während vorher die Informationen in verstreuten und unterschiedlichen Datentypen gespeichert waren, sind sie nun in einem einzigen Datentyp gespeichert. Das Datenmodell ist also völlig in sich gekapselt. Die *getter*-Methoden der Klassen *ModelImpl* müssen also die Informationen aus dem *RESTActivity*-Element extrahieren und so aufarbeiten, dass sie mit dem Rückgabewert der Interfaces übereinstimmen.

6.4.2 ContexParser und ExtensionParser

Der Mittelpunkt des neuen Datenmodells ist das *RESTActivity*-Objekt. In ihm werden alle nötigen Informationen des BPEL-Prozesses abgespeichert und verwaltet.

Während im alten Modell der *ExtensionParser* über die DOM-Elemente des BPEL-Prozesses iteriert ist, ist dies nun nicht mehr nötig. Die *RESTModelingExtension* definiert einen *RESTActivityDeserializer*, welcher die XML-Elemente der *<extensionActivity>* einließt und daraus das Modell erstellt. Der *RESTActivityDeserializer* erstellt im Endeffekt ein *RESTActivity*-Objekt aus den eingelesenen XML-Elementen. Der *ExtensionParser* von *BPEL4REST* muss also nur diesen *RESTActivityDeserializer* aufrufen und ihm die XML-Elemente der Apache ODE übergeben um ein *RESTActivity*-Objekt, und damit das Modell, zu erstellen. Damit dies funktioniert muss der *ExtensionParser* allerdings erst noch einen neuen *VariableResolver* für den *RESTActivityDeserializer* erstellen, damit die *BPEL-Variablen* des Modells auch mit der Variablenauflösung der Apache ODE übereinstimmen.

Listing 12 Code-Ausschnitt der neuen BPEL-Parsers von *BPEL4REST*

```
public void parseExtensionElement() throws ParsingException
{
        ...
        VariableResolver vr = new BPEL_ODE_VARIABLE_RESOLVER();
        RestDeserial.setVariableResolver(vr);
        restActivity = (RESTActivity) RestDeserial.unmarshall(...);
        ...
        this.model.setModelRestActivity(restActivity);
}
```

In Listing 12 ist der Java-Code zu sehen, mit dem der *ExtensionParser* einen neuen *VariablenResolver* erstellt, diesen dem *RESTActivityDeserializer* übergibt, diesen dann aufruft und die entstandene *RESTActivity* an die Modellimplementierung von *BPEL4REST* übergibt.

Dies muss geschehen, das die Modellimplementierung die Informationen aus der *RESTActivity* auf die Methoden der Interfaces anpasst.

Da das *<context>*-Element schon im *ExtensionParser* ins Modell eingefügt wurde, entfällt die Klasse *ContextParser* und wird deshalb aus *BPEL4REST* entfernt.

6.4.3 ContextInterface und Context

Durch den *ExtensionParser* ein *RESTActivity*-Element erstellt wird, in welchem alle Informationen des *<context>*-Elements gespeichert sind. Für das *ContextInterface* gibt es also keine Grund mehr die *setter*-Methoden zu behalten. Dementsprechend wurden sie bei der Implementierung des vereinheitlichen Datenmodells entfernt. Das *ContextInterface* wird weiterhin von der Klasse *ModelImpl* implementiert. Allerdings greifen die Methoden jetzt direkt auf das *RESTActivity*-Element zu. Es besteht also keine weitere Verwendung für Objekte der Klasse *Context*. Deshalb wurde sie aus *BPEL4REST* entfernt.

6.4.4 ResponseHeaderInterface und ResponeHeader

Bei die Analyse der Datenmodelle und der XML-Spezifikation von *BPEL4REST* wurde entschieden das *ResponseHeaderInterface* und die Klasse *ResponseHeader* vom Modell zu trennen, da sie nur die HTTP-Antworten der Webservices speichern. Da diese HTTP-Antworten allerdings auch wichtige Informationen für den Ablauf der Erweiterung beinhalten können, können diese Klassen nicht weggelassen werden.

Da die Informationen der HTTP-Antworten hauptsächlich von der Erweiterung verwendet werden, wurde entschieden die beiden Klassen an den *ExtensionController* zu binden, da dieser den Ablauf der Erweiterung steuert. Es wird nun mit für jeden *ExtensionController* ein *ResponseHeader* erstellt, in dem dann die HTTP-Antworten der Webservices dieses *ExtensionControllers* gespeichert werden.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Datenmodell erstellt und implementiert, welches auf der Spezifikation der BPEL-Erweiterung *BPEL4REST* basiert und in der BPEL-Erweiterung *BPEL4REST* und der BPEL-Designer-Erweiterung *RESTModelingExtension* verwendet werden kann. Das Datenmodell wurde in beiden Erweiterungen implementiert und hat an der Funktionalität und Korrektheit der Erweiterungen nichts verändert. Dieses Datenmodell wurde so erarbeitet, dass es einfach in beide Erweiterungen eingebaut werden konnte und Änderungen an diesem Modell einfach getätigt werden können.

Im Kapitel 2 wurden erst die verwendete Begriffe wie REST, BPEL und zwei Methodiken in der Softwareentwicklung, Schnittstellen/Interfaces und Code-Generierung, erläutert, da diese zum Verständnis dieser Arbeit notwendig sind.

Anschließen wurde eine Erweiterung von BPEL, *BPEL4REST*, vorgestellt. Diese Erweiterung ermöglicht es Benutzern BPEL-Prozesse nicht mit auf WSDL-basierenden Services, sondern auch mit REST Services zu orchestrieren. Erst wurde beschrieben, wie *BPEL4REST* die REST Services anhand von BPEL-*<extensionActivity>* für fünf HTTP-Verben definiert. Danach wurde auf die Architektur von *BPEL4REST* eingegangen.

In Kapitel 4 wurde eine Erweiterung des Eclipse BPEL Designers, die *RESTModelingExtension* beschrieben. Da *BPEL4REST* keine graphischen Modellierungswerkzeuge anbietet und es auch keine Validierung des BPEL-Prozesses vor dessen Ausführung gab, wurden diese Arbeit entworfen. Die *RESTModelingExtension* ist ein graphisches Modellierungs- und Validierungswerkzeug für *BPEL4REST*. Hier wurde dessen Funktionalität und Architektur beschrieben.

Da die beiden Erweiterungen auf der selben Spezifikation arbeiten, jedoch ihre Datenmodelle unterschiedlich entworfen und implementiert haben, wurden in Kapitel 5 diese Datenmodelle genau betrachtet und analysiert. Basierend auf dieser Analyse wurde eine Entscheidung getroffen wie das vereinheitliche Datenmodell für beide Erweiterungen entworfen werden soll und welche Änderungen an den Erweiterungen dafür in Kauf genommen werden.

In Kapitel 6 wurde dann die Implementierung des vereinheitlichen Datenmodells dargelegt. Neben der Entwicklung- und Testumgebung wurden noch die Unterschiede bei der Haltung von BPEL-Variablen im BPEL Designer und in der Apache ODE aufgeführt. Anschließend wurden aufgeführt, was sich an der Architektur und an den Implementierungen der Erweiterungen verändert hat, als das vereinheitliche Datenmodell in die Erweiterungen eingesetzt wurde.

7.1 Ausblick

In dieser Arbeit wurde nach einem Datenmodell gesucht, was sich sehr an der Struktur der Spezifikation von *BPEL4REST* orientiert und mit geringem Aufwand in beide Erweiterungen einbauen lässt. Das EMF-Modell der *RESTModelingExtension* ist dieser Spezifikation schon sehr nahe. Es wäre natürlich möglich ein Datenmodell zu entwerfen, welches direkt der Spezifikation von *BPEL4REST* entspricht, auch wenn dafür größer Aufwand an den Erweiterungen betrieben werden muss.

Obwohl das Datenmodell der *RESTModelingExtension* durch das EMF-Modell schon mit geringem Aufwand verändert werden kann, können gewisse Änderungen immer noch zu größerem Schaden im Rest der Erweiterung führen. Hierbei müsste die Kapselung des Modells noch verstärkt werden.

In *BPEL4REST* sollten die Interfaces überarbeitet werden um besser an das neue Datenmodell angepasst zu sein. Auch wenn dies keinen funktionalen Vorteil bringt, würde sich die Lesbarkeit der Erweiterung dadurch erhöhen. Auch sollte die Rolle des *<responseHeader>*-Elements in der Spezifikation und auch im Datenmodell überarbeitet werden, um Klarheit zu schaffen, wozu das Element gehört.

Literatur

[1] Eclipse BPEL Designer Project. http://www.eclipse.org/bpel.

[2] Eclipse Modeling Framework. https://eclipse.org/modeling/emf/.

[3] Hypermedia. http://de.wikipedia.org/wiki/Hypermedia.

[4] Hypermedia as the Engine of Application State (HATEOAS). http://en.wikipedia.org/wiki/HATEOAS.

[5] Notes on the Eclipse Plug-in Architecture. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, 2003.

[6] OASIS Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.

[7] Apache Software Foundation. Apache ODE. http://ode.apache.org.

[8] Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org.

[9] Eclipse Foundation. Eclipse (IDE). www.eclipse.org.

[10] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. https://www.w3.org/TR/wsdl, 2001.

[11] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. Master's thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

[12] Markus Fischer. RESTful BPEL - Erweiterung von BPEL zur Orchestrierung von RESTful Web Services, 2013.

[13] Florian Haupt, Markus Fischer, Dimka Karastoyanova, Frank Leymann, Karolina Vukojevic-Haupt. Service Composition for REST, 2014.

[14] Grady Booch, Ivar Jacobson, James Rumbaugh. Unified Modeling Language. www.omg.org/spec/UML/.

[15] Jochen Ludewig, Horst Lichter. *Software Enginieering - Grundlagen, Menschen, Prozesse, Techniken.* dpunkt.verlag, 2010. 2. Auflage.

[16] Volha Kalach. Modellierung von REST Service Kompositionen, 2014.

[17] L. Masinter, T. Berners-Lee, R. Fielding. Uniform Resource Identifier (URI). http://www.ietf.org/rfc/rfc3986.txt, 2005.

[18] N. Freed, N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. http://www.ietf.org/rfc/rfc2046.txt, 1996.

[19] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. `https://www.w3.org/TR/wsdl20/`, 2001.

[20] T. Berners-Lee, P. Leach, L. Masinter, H. Frystyk, J. Mogul, J. Gettys, R. Fielding. Hypertext Transfer Protocol – HTTP/1.1. `http://www.ietf.org/rfc/rfc2616.txt`, 1999.

[21] Stefan Tilkov. *REST und HTTP - Einsatz der Architekur des Web für Integrationsszenarien.* dpunkt.verlag, 2011. 2. Auflage.

Alle Links wurden das letzte Mal am 05.04.2016 auf ihre Funktionalität und Korrektheit überprüft.

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.

Unterschrift: