Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3601

A multilayered model for REST applications

Jens Petersohn

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Dipl.-Inf. Florian Haupt
Commenced:	December 13, 2013
Completed:	June 13, 2014
CR-Classification:	I.7.2

Acknowledgements

First and foremost I offer my sincerest gratitude to Prof. Dr. Frank Leymann and particularly my supervisor Dipl.-Inf. Florian Haupt who made this thesis possible at the Institute of Architecture of Application Systems (IAAS). Without his commitment and support this thesis would not have been feasible. His assistance, advice and guidance were extremly valuable while accomplishing and completing this task.

I am incredibly thankful for the support of my family which stood by my side during the duration of study. Especially my parents which allowed me the most worriless time that student could possibly imagine. Their support and advice for every situation in my life was and will be invaluable. For this I will always be grateful.

Further I want to thank all of my friends which accompanied me for years and those I met during my studies. You enriched this part of my life and made it memorable.

Finally I want to show my gratitude to RedScreen which provided me the possibilitiy to gain my best practical work experience overseas. Thank you so much for an interesting and versatile working environment and for your constant support.

Abstract

Representational State Transfer (REST) web services rapidly increased their importance in the last years. Nowadays a lot of services use this architectural style to benefit from the advantages and characteristics of a RESTful system. To call a web service REST compliant several constraints have to be fulfilled by software developers. The compliance of these guidelines is often not ensured even though many services call themselves RESTful. By applying Model Driven Software Development aspects to the design of REST applications a system was developed to decrease the effort which is needed to follow those constraints and reduce the time it takes to design and implement a REST web service. The process involves the modeling and editing of several abstract models which act as an input for a code generator that creates a JAX-RS web service. The models are defined by using the Eclipse Modeling Framework (EMF) with additions for graphical modeling.
Since the current outcome of the existing prototype is not completely REST compliant certain improvements have to be made. This thesis reveals flaws during the modeling process in several meta models and in their transformations. It states enhancements how to improve and ease the process as well as increasing the quality and REST compliance of the generated outcome. To verifiy an improved state of the software a show case is used as a reference throughout the document. Finally the applied changes to the approach and the system structure are realized and demonstrated regarding this show case.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

The Representational State Transfer (REST) architectural style became an important guideline for web services, especially when accessing hypermedia information. Besides SOAP (Simple Object Access Protocol), XML-RPC (Extensible Markup Language - Remote Procedure Call) or JavaScript APIs, REST is the most used protocol according to ProgrammableWeb's [1] API database. However, this thesis only concerns with the REST API and its characteristics.
The architectural style is mainly compound of constraints which influence the components and consequently the whole web service in its structure and behavior. The architectural properties which should ensue by following those guidelines are primarily, among others, network performance, scalabalitiy of components that interact with each other and simplicity of their accessible interfaces.

To achieve this, the development of the service should preferably fit all, or at least a set of these constraints stated by Roy Fielding in his dissertation in 2000 [Fie00]. A lot of services accessible by the internet call themselves REST compliant but they do not follow the given guidelines entirely. Thus, they are only in parts "REST" compliant. To provide a web service with the mentioned advantages a web service has to fulfill all specified REST constraints. Leonard Richardson developed a model (Richardson Maturity Model [WPR10]) to classify a web service by the set of its applicable REST constraints. This model will be used to classify different compliance conditions. It will be introduced and described in detail in chapter 2 (Background).

One of the best examples applying the REST service style is the World Wide Web (WWW) itself. The REST characteristics and constraints were defined and specified by analyzing today's web. Hence there are a lot of obvious similarities when analyzing the WWW based on the REST architecture style. Some constraints are already fulfilled by the Hypertext Transfer Protcol (HTTP) which is the required basis when interacting within the web. Still one of the main issues is the realization of the constraints that have to be explicitly followed by a software architect or software developer and that are not originally covered by any of the WWW standards. Common ones are tunneling requests through a single HTTP method (GET, POST, ...), ignoring hypermedia (HATEOAS), ignoring multiple MIME types for resources and a single URI as the service endpoint [Vit10]. This is currently the part of the REST web service

[1] ProgrammableWeb - http://programmableweb.com/

development where most of the “RESTful” characteristics get lost.

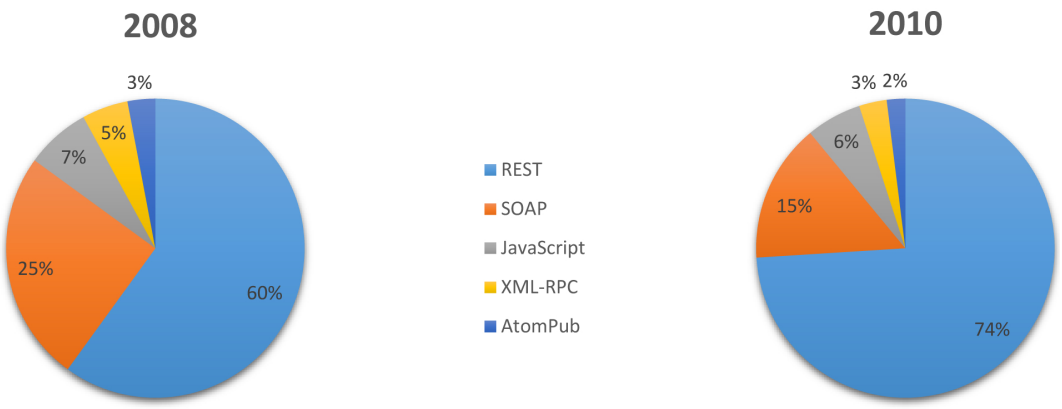

Figure 1.1.: API Protocol Types (ProgrammableWeb 2010 [DuV10])

As stated above it turns out that there is a need of improvement for the REST web service development. There has to be a structured approach to increase the quality and REST compliance of new web services and additionally decrease the effort and complexity to design them. During the development there most probably emerges an abstract design document which will generically describe the components and their interactions or relations with each other. This document will then be the foundation for the upcoming code development. The code realization of it can be quite error-prone due to misunderstandings, indistinctness or in some cases even ignorance. To reduce or even better avoid those implementation errors and to create a clearly defined connection between the abstract software model and the software code itself, the use of Model Driven Software Development (MDSD) is recommended or in some cases compulsory.

Precondition for an executable software are the models that have to be formal. A formal model should describe an aspect of the software completely by following predefined rules. Transforming the model automatically means that it can be used as an input for other models or code generation without any user interaction. The transformation can be performed by a script, an engine or an automated mechanism. Hence there is only little or no user input at all, the model provides the key aspect of the transformation source. The final software does most often consist of auto-generated parts combined with manually implemented code. In the following a common process of software development without using MDSD is described.

Usually the connection between a draft or an abstract design of a software and the final version of the code is the documentation. The documentation is normally created by a developer or by a quality assurance instance reinterpreting the already existing draft. However the

challenge that has to be faced is to create code and documentation out of drafts. Hereby a lot of effort is needed for repetitive work like creating files, writing duplicate code or maintaining software quality. In case some piece of code contains an error and it is copied all over the software, the resulting bug has to be fixed in every file or document separately.

Using Model Driven Software Development these tasks can be done much faster, more unsusceptible to errors and with a definite way of quality. Therefore, the draft has to be a formal model that explicitly reflects predefined characteristics of the software. A widely used modeling language which is standardized and applicable for formal models is UML (Unified Modeling Language)[2]. By transforming the model it is possible to generate executable code in an automated manner by interpreting the model based on the capability of its expressiveness. The transformation process can be performed with different technologies, as long as they can parse and process models or diagrams which use the UML notation. The essential about the generation is that it is automated as far as possible and it only needs few or even no human interaction at all. The precondition for this is a correct and structured transformation process which has to be individually developed for each target model or code.

By combining REST and MDSD there is a possible solution of guided RESTModeling that can approach the varying REST compliance in a special way. If there would be a single or multiple suitable formal model descriptions for a REST service a transformation could generate other models or even executable source code to provide the designed web service. The huge benefit from combining these two technologies is that constraints which have to be followed by the developers can now be fulfilled by correct and formal model definitions as well as the predefined model-to-model translations. So in case a model is valid for a transformation we can assume that the outcome is as “RESTful” as the transformation process alters it. As a start and proof of concept in [Sch13] an Eclipse[3] plugin has been developed which allows modeling a REST web service and transform it into other models or even to JAX-RS code by using a custom code generator. Since it was a first approach towards model driven REST service development some constraints which have to be fulfilled are not yet designed and integrated in this solution.

[2]Unified Modeling Language - http://www.uml.org/
[3]Eclipse IDE - http://Eclipse.org/

1.1. Motivation

The current state of the modeling plugin is insufficient to build a fully REST compliant web service. Certain basic approaches and assumptions that were made have to be improved and specified in detail. The focus of this thesis is to identify inconsistencies during the modeling process and increase the REST compliance of the generated outcome. To achieve this the current system is analyzed and tested by using an example show case. By applying this show case, several issues were revealed and documented. These stated deficits were improved and corrected during the process of this thesis.

1.2. Outline

The approach towards proper REST modeling is structured as follows:

Chapter 2 – Background about the thesis' topics and foundation of the following work.

Chapter 3 – Condition Of The Prototype which is used for RESTModeling.

Chapter 4 – Deficits Of The Prototype which were revealed.

Chapter 5 – Enhancements that have to be made for an improvement.

Chapter 6 – Technologies which are used for the modeling process.

Chapter 7 – Implementation which was necessary to realize the stated enhancements.

Chapter 8 – Conclusion about the completed work of this thesis.

Chapter 9 – Perspectives for future improvements or development.

1.3. Abbreviations

Within this thesis several abbreviations are used. To provide clarification of the used terms and to avoid confusions with similar abbreviations the following list outlines the ones which may not be commonly known and used within the whole thesis. Other abbreviations which appear in a specific context are described right before they are used.

Abbreviation	**Meaning / Description**
IDE	Integrated Development Environment
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
JAX-RS	Java API For RESTful Web Services
Ecore	EMF Core Meta Model
Emfatic	Ecore Syntax
EuGENia	Ecore Model Processing Engine

1.4. Typography

Within this thesis there are several references to examples, figures, meta models and to the show case. To explicitly underline them they are written with `spaced characters` to avoid any different or confusing meanings.

2. Background

This chapter provides a more detailed introduction to the REST constraints as well as the background behind model driven software development and provides the foundation of the thesis.

2.1. Representational State Transfer (REST)

The REST architectural style provides a lot of benefits which are hard to achieve using other web service patterns which will be shown within this section. It differs significantly to other well known architectural styles, what brings up the subsequent advantages if we compare it for example to RPCs (Remote Procedure Calls).

Loose Coupling RPC systems or components have to know a lot of their counterpart (procedure signature, ip address, port, ...) to interact with it. That will most likely cause issues when a modification of one of the parties is required. The new knowledge about the changed parties' attributes has to be updated in every communicating component which should still be able to interact properly with that system. To avoid unnecessary and time-consuming changes a uniform interface is used by REST compliant services. By following the constraints, the knowledge regarding the API structure can be reduced to a minimum between two systems.

Interoperability The desired quality of a highly interoperable system is that it does not matter what kind of implementation two communication systems have, as long as they have a distinct set of standards which both of them follow. This is why REST uses the typical web standards HTTP, URIs and XML (HTML). No matter which technology is used for internal purposes of a system, while it uses HTTP as the transfer protocol and URIs to identify resources it can be easily connected to others. Remote procedure calls in contrast are often based on some sort of middleware or Remote Procedure Invocation (RMI) protocols which need a lot of adjustability for communication if the protocol differs.

Performance and Scalability Due to the fact that HTTP is stateless and REST service requests carry their current state within themselves, a lot of requests can be answered by a cache and request overheads can be reduced to a minimum. In some cases the requests do not have to be answered by the same server what brings the possibility to balance the workload.

To benefit from these entities the REST architecture demands a set of constraints that have to be fulfilled by the web service. These constraints were specified and designed by Roy Fielding [Fie00]. Outlining the constraints results in the following list according to Stefan Tilkov [Til11]:

Unique Resource Identification Using a global naming scheme (URIs) every resource should be uniquely identifiable. Every provided abstraction of the system should be accessible by an URI independent whether it is an individual entry or an amount of resources.

Different Representations In case there are multiple instances of a resource available the client should be able to specify an accepted content type to get the data representation he desires. This requirement is specified in the header of an request and provides more flexibility for different purposes.

Hypermedia As The Engine of Application State (HATEOAS) To navigate back and forth between two REST states and to connect resources with each other Link objects should be provided in a response. By using them the application flow can be controlled by the server and the navigation is obvious for the requesting system.

Stateless Communication Due to the usage of the HTTP protocol a request contains its own state. Hence it is not required that further requests have to be processed by the same server that returned previous responses. This results in a high scalability of a REST web service and increases its performance.

Standard Methods The Usage of the available HTTP operations (POST, GET, PUT, DELETE, HEAD, ...) realizes the CRUD (Create, Read, Update, Delete) strategy for a REST web service. Other web services often only support POST and GET which does not exploit the richness of the HTTP protocol.

R. Fielding stated that a web service is REST compliant if it fulfills all of his listed characteristics and constraints. Since the constraints are not ordered in a preferred hierarchy or dependence most of the web services which should be REST compliant only satisfy a disordered subset of them. Therefore Leonard Richardson [WPR10] developed a model that breaks down the principal elements of a REST application into three levels. Each level reflects an improvement of the web service quality to get closer to a fully compliant REST service. Using the Richardson Maturity Model a self-proclaimed REST web service can be classified in a structured approach. Figure 2.1 displays the different levels of the model. A service is valid in a level X if it satisfies all requirements of level 0 to level X itself whereas level 0 is the lowest and initial level.

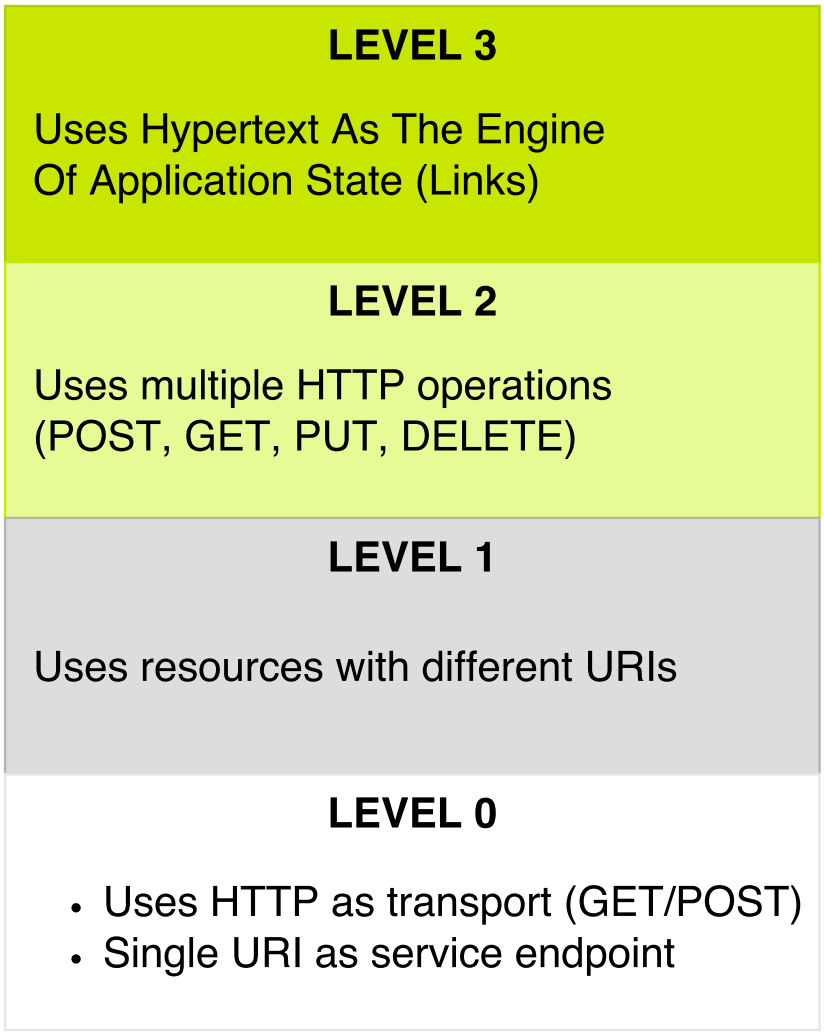

Figure 2.1.: Richardson Maturity Model Levels

Level 0 To achieve this level the HTTP protocol has to be used as a transport system with an arbitrary operation (GET/POST). Not using HTTP makes the model not applicable and does not match to the REST constraints at all. All interactions happen with a single endpoint that represents every resource in the system.

Level 1 This level requires the accessibility of different resources. By querying the initial root resource the service has to return further resources which can be addressed individually.

Level 2 Using HTTP and providing access to different resources this level covers the usage of multiple HTTP operations or verbs. The WWW does not use all the HTTP verbs itself but it is required that REST web services use (POST, GET, PUT, DELETE).

Level 3 The final step to a fully compliant REST web service is the provisioning of links (HATEOAS). This supports discoverability for any client of the service and makes it more self-documenting.

2.1.1. Best Practices For RESTful APIs

To ease the realization of a REST web service and increase the constraint compliance a list of best practices were outlined by Stefan Jauker [Jau14]. This list may help developers to improve their conception of a RESTful API.

Use Nouns No Verbs Every web service end point should be identified with a noun. In case a list of cars wants to be requested via HTTP GET, a reasonable URL for an end point would be `/cars` instead of `/getAllCars` or `/listAllCars`.

Idempotentce of HTTP GET Since HTTP GET is idempotent [FIa99] influencing or altering states should not be possible by performing GET requests. For this purpose HTTP POST, PUT and DELETE should be used. Requests like `GET /accounts/3212/deactivate` should not be used to deactivate an account.

Plural Nouns To keep simplicity singular and plural nouns should not be mixed up. Using the plural of every resource's noun keeps it simple even though only a certain resource is requested (e.g. `/accounts/3212 instead of /account/3212`).

Sub-Resource Usage To show relations between different objects sub-resources should be used. In case an account has multiple owners, the connection should be obvious by querying a certain URL. Requesting the URL `/accounts/3212/owners` via HTTP GET returns all owners of the account which is identified by 3212. Respectively requesting `/accounts/3212/owners/3` should return the owner identified by 3 for the account 3212.

Format Communication To identify the format which is used for a client-server-communication HTTP headers should be used to identify the MIME type. Via the header attributes Content-Type and Accept it should be defined what format defines the request body and what formats are acceptable as a response body.

Use HATEOAS This guideline directly addresses the previously mentioned constraint to provide a better navigation through the API.

Collection Response Customization Since collections (e.g. `/accounts`) may have a lot of entries, it should be possible to filter and sort them, as well as querying for specific fields and limit the amount of their returned entries by adding query parameters. As an example: requesting `/accounts?active=true&sort=+ID&start=0&stop=10` would return at maximum the first ten accounts which are active and in ascending order by their ID.

API Versioning API versioning should be mandatory since no API should be released unversioned. Hereby the dot notation should be avoided. It is advised to use "v" as a prefix of a version identifier (e.g. `/blog/api/v2`).

Proper HTTP Status Codes There should not be a generel status code for errors or states of a web service. Error handling should not be neglected by simply returning one HTTP code (e.g. 500). HTTP provides over 70 status codes which are categorized into different groups. Table 2.1 shows an extract of HTTP status codes and their meaning. Besides returning a correctly identifying status code, an error payload can be returned to the requesting client as well. The format of this payload may be XML or JSON providing additional information about an occured error.

Overriding HTTP Methods Some proxies do not allow to use different HTTP methods than POST and GET. To still maintain the RESTfulness of an API despite that limitation, the API needs a way to override the POST and extend its use since HTTP GET must remain idempotent. One way of implementing this is using the custom HTTP header `X-HTTP-Method-Override` for the POST method.

Code / Code Group	Description
1xx Group	Information. Request received, continuing process.
2xx Group	Success. Request received, accepted and processed successfully.
200 OK	Standard response.
201 Created	Request fulfilled and new resource created.
202 Accepted	Request accepted for processing but not processed yet.
...	
3xx Group	Redirect. Client must take additional action to complete request.
300 Multiple Choices	Multiple options for a certain resource are available (e.g. formats).
301 Moved Permanently	This and all future request should be directed to the returned URI.
...	
4xx Group	Client Error. Returned in case it seems that the client has errored.
400 Bad Request	Request can not be fulfilled due to bad syntax.
...	
403 Forbidden	Request was valid but server refuses response.
404 Not Found	The requested resource could not be found.
...	
5xx Group	Server Error. Server failed to fulfill an appranetly valid request.
500 Internal Server Error	Generic error message.
501 Not Implemented	Server does not recognize the requested method.
...	

Table 2.1.: HTTP Status Codes Extract

2.2. Model Driven Software Development

According to [SVEH07] the definition given in 2.2.1 describes the characteristics of Model Driven Software Development.

Definition 2.2.1 (Model Driven Software Development [SVEH07])
"Model Driven Software Development is a genus for techniques which create executable software out of formal models."

It is based on three important aspects that result in MDSD when composed.

Formal Models Models do not necessarily have to describe all aspects of a software but they should describe a specific part of it. Hence an accurate description about the expressiveness of the model is compelling. So it has to be obvious which model can be used for what type of problem.

Generate Runnable Software In general a distinction between a code generator and a code interpreter is necessary. For this thesis an interpreter is not relevant so every statement about generating runnable software addresses the code generator, which uses a given model to generate a separate piece of code by processing it. In case a model is not formally described the transformation performed by the generator will most likely cause problems.

Automated Processing Even though a model has to be designed manually the generation of code should happen automated or at least with very little effort. So changes that should affect the system or the code itself can be made in the model and transformed into code afterwards.

MDSD or Model Driven Architecture in general is based on two key types of models [Mei05]. The Platform Independent Model (PIM) identifies the target domain where only specific aspects of a software are modeled and no statements about their realization are made. Although this model does not have to result in a piece of code or software it is still valid and provides a foundation for prospective discussions or developments. In contrast the Platform Specific Model (PSM), which is the result of combining a Platform Description Model (PDM) and the PIM, describes a specific implementation or representation for a specific platform. The process of using a PDM with a PIM would be a proper transformation which can result in a new model or in a piece of code, even though a piece of code would obviously be just another representation of a model.
In case a scenario involves multiple transformations (Figure 2.2) of a PIM, the PIM would act after its transformation to a PSM as a PIM for the next transformation layer. This can result in a repetitive flow so that every PIM is the PSM of a previous transformation.

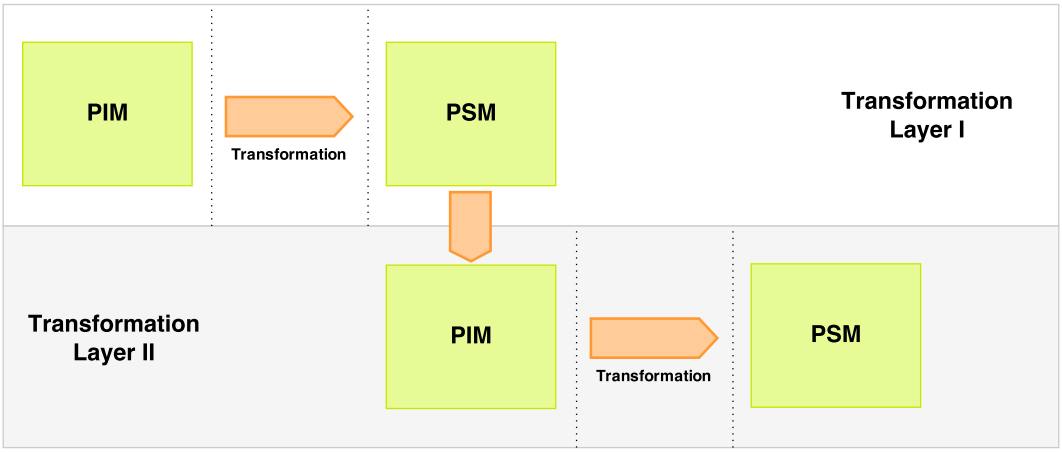

Figure 2.2.: MDA Multiple Model Tranformations

2.2.1. Areas Of Application

A common use of Model Driven Software Development is to generate Applications out of instances of predefined meta models. A target language independent tool for this is Actifsource[1]. A specified meta model can be easily transformed into an graphical editor. The modeled output of that editor can be used to generate exectuable code (e.g. Java). It is built on Eclipse and provides a large set of tools for the meta modelling process.

Besides the generic use of application development for desktop machines or servers there are other fields which gain more and more importance. The mobile application development industry is rapidly growing due to the intensive use of mobile devices. Especially mobile applications for Android[2] operating systems increases significantly as it can be seen in Figure 2.3.

[1]Actifsource - http://www.actifsource.com/
[2]Android - http://www.android.com/

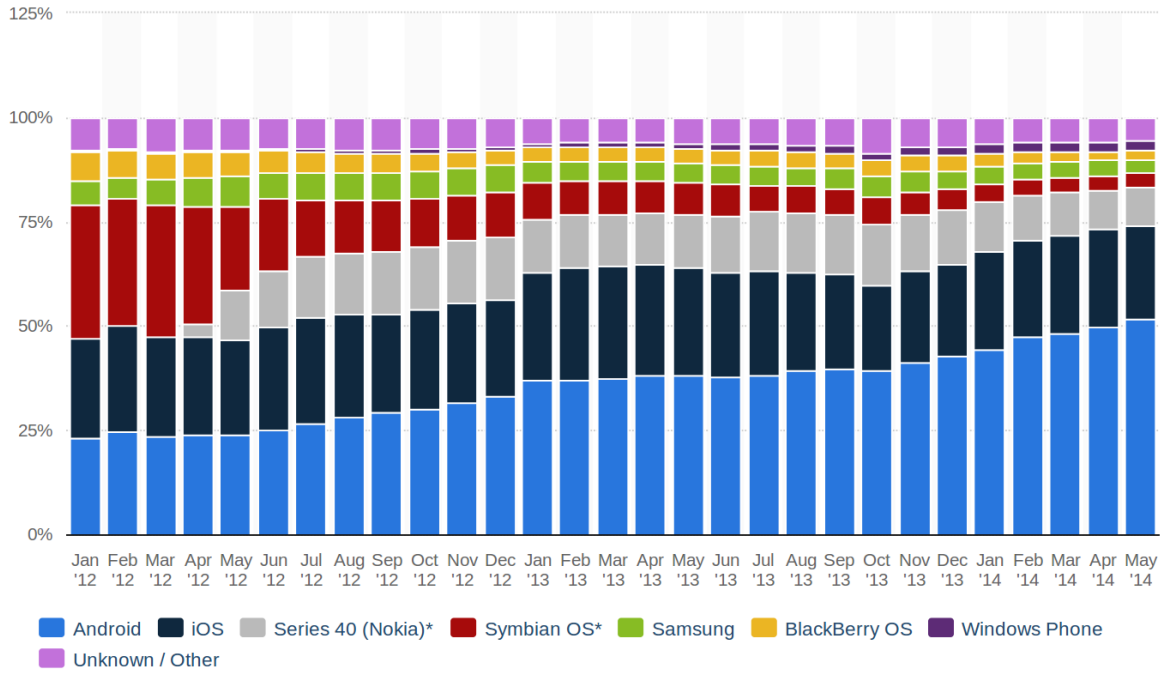

Figure 2.3.: Market Of Mobile Operating Systems 2012-2014 (Source: statista.com)

For the development of mobile Android applications Google provides a Java based Application Interface (API). Using this API A. Parada and L. Brisolara developed an approach towards model driven Android application development in [PB12]. The two types of models which are needed are completely based on UML syntax. UML class diagrams are used to define classes and their relations and sequence diagrams describe their behavior and semantic guidelines. The models are used as input for the also self-developed project GenCode which generates Java code out of UML syntax based models. Figure 2.5 shows their presented show case of the mobile device game Snake. The grayly highlighted classes represent classes of the API, classes having a transparent background are self modeled classes of the game.

An already existing software for model driven software development for Android and iOS[3] is MD[2]. It is a set of plugins for Eclipse which allow the cross-plattform development of native business apps.

A completely different and one of the most challenging fields in software engineering is the development of embedded systems, especially in the automotive sector. Due to the increasing

[3]iOS - https://www.apple.com/de/ios/

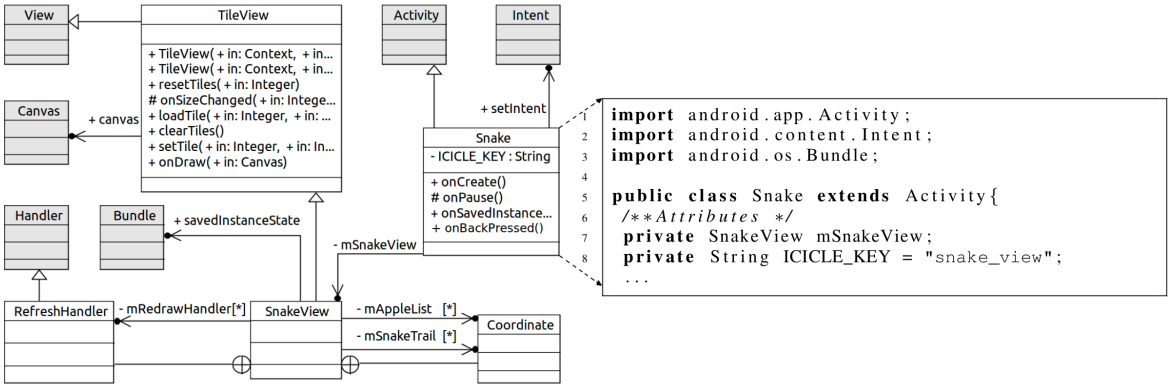

Figure 2.4.: Snake Example - UML Class Diagram To Code Transformation

amount of electronic driven innovations, software systems emerge to a central aspect in this industry. Challenging about the application of MDSD to embedded systems is that almost everything wihtin such systems must be developed from scratch. Only some very little parts can be reused, compared to a common desktop software. Since most of the code is often written in C or even lower level programming languages many object oriented patterns can not be applied to them.

In [VSK06] the focus is the communication between several embedded systems. The approach is to create a specific container which lays upon an embedded device's operating system and provides a certain level of abstraction with reusable components and constructs. Therefore the meta model specifies different aspects (component types, interfaces, connectors among components, hardware topology, network types, etc). A single component (C)has still be designed and developed manually regarding the given container interfaces.
Modeled containers are used by a code generator to create the required infrastructure code for the embedded environment.

Another field is approached in [MSGR09] which addresses the security of Smartcards. Smartcards are used in electronic passports, payment or loyalty cards or mobile sim cards. The major ingredient of them are their cryptographic protocols to secure linked data. The developed software SecureMDD provides the development of secure Java card codes based on a set of UML based meta models (class, deployment and activity diagrams). By defining reusable constructs a certain level of security can be granted due to model-to-code transformations which focus on security. A concrete software is currently not available on the market.

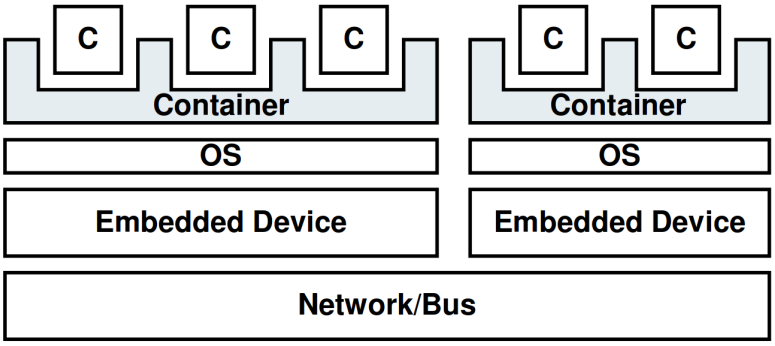

Figure 2.5.: MDSD Approach In An Intercommunicable Embedded System

2.2.2. Advantages And Disadvantages

By using MDSD developers gain a lot of ease when creating new complex systems. Apparently it is not possible or very unlikely to create a system only based on automated code generation out of formal models. Specific input from developers is still needed but it is significantly reduced. According to [SV06] there are some obvious benefits but likewise some serious disadvantages.

Advantages resulting of MDSD:

Development Speed Achieved through the already mentioned automated code generation.

Software Quality The predefined models and the automated transformation result in a determined state of code and software quality.

Manageability of Technical Changes Changes that have to be made to a piece of code or software are simply made in the modeled representation of it.

Redundancy Avoidance As long as the automated code generation does not replace or alter any user specified code a lot of changes can be done quickly by editing the model.

Reusability Defined models and transformations can be reused to imitate a software production line.

Disadvantages resulting of MDSD:

High Initial Effort Repetitive work is significantly reduced, but the initial effort of defining a model and creating transformations and generations is very high.

Complex Code Generator The generator has to differentiate manually added code and auto-generated code.

Code Loss If the generator does not take care of manually added code it will be overwritten most probably.

Complex Configuration Management Not versioning auto generated code saves resources but it needs a lot of customized configuration adjustments.

Runtime Debugging Using an interpreter to process a model could result in inevitable runtime debugging.

Multiple Technologies Often multiple technologies are necessary for modeling and processing the models and the compatibility between them can not be taken for granted.

The advantages and disadvantages of Model Driven Software Development have to be considered and prioritized individually for every project or task. However, there are inevitable risks which may occur in different situations when using a MDSD approach or solution which were stated by Johan Den Haan [DH09]. In section 2.2.1 different arreas of application are described which definitely have individual risks but some risks apply to all of them.

Increased Rigidity In comparison to manual code development a large degree of design freedom gets lost when using MDSD. Since a model is generally a higher level of abstraction the developer or designer has to specify less but generate more. Smaller changes can often not be applied to a generalizing model.

Predefined Flexibility Using predefined models does not provide much flexibility unless it is explicitely designed. Limiting factors are the usage of the tool (or tools) to perform MDSD and the Domain-Specific Languages that are used. The higher the level of a DSL is, the more hard-coded code fragments will be used from the framework.

Project Member Roles The common software development process has software programmers and architects which code the and design the software. Since MDSD builds a certain bridge between Business and IT new roles are defined for such a project. Programmers used to move into a “meta-team” that defines the factory of a model to code generation. The solution is now build by “business engineers” which have to combine background knowledge and the business requirements into such a model. Acquiring these rare employees having this skill set can be very difficult.

Difficult Model Version Control Conventional version control systems like Subversion[4] or Git[5] work fine for actual code or textual documents. The versioning of a large and complex software in a graphical representation can be much more difficult when working in large teams.

Incomplete Modeling Tool In case a modeling tool (or model-driven software factory) is not finished at the point of its initial use in a project huge risks can be the result of applying it. In case there are DSL or meta model changes in the middle of a MDSD process it may result in incompatibility to the already modeled software.

Awareness Of Modeling Possibilities A huge risk is the missing awareness of technical possibilites of the modeling process. In case some unaware employee makes promises to a customer about features or constructs which are not realizable, a lot of credibility and trust can get lost.

Neglect Of Project Objectives Using a MDSD tool and focusing on its advantages may result in neglecting other important project objectives like project management or sticking to some certain process.

[4]Apache Subversion - http://subversion.apache.org/
[5]Git - http://git-scm.com/

3. Condition Of The Prototype

This chapter describes the state of the already existing technology. The exploratory work [Sch13] covers certain simple model definitions and transformations which are realized in a set of Eclipse plugins. This prototype provides modeling of REST applications on a very basic level and conduces as a first proof of concept. Since the used technologies remains the same during the enhancements of this thesis all technical details are described in chapter 6.

The tool allows to graphically model a REST web service from different views, whereby a view corresponds with at least one model or a combination of models. By designing abstract relations between objects and mapping them to resources for the web service, `JAX-RS`[1] stubs are a possible result of the system. Additionally a HTML documentation can be generated which lists the available paths and the provided HTTP methods of the generated web service.

The concept rests upon several models that are related to each other. Every model pictured in Figure A.2 can be generated by using its predecessor from an upper level. Models which have an opaque green background are supported by a graphical diagram editor, gray and orange models only have an XML representation which can be altered. The generation itself is only unidirectional so transforming back and forth is not possible. Starting on a generic basis, the first model types are platform independant. Modeling within these levels does not influence any specific implementation for desired target applications. Further, to obtain a specific implementation or `Platform Specific Model` certain transformations have to be applied to the previous `Platform Independant Models`.

[1] JAX-RS - https://jax-rs-spec.java.net/

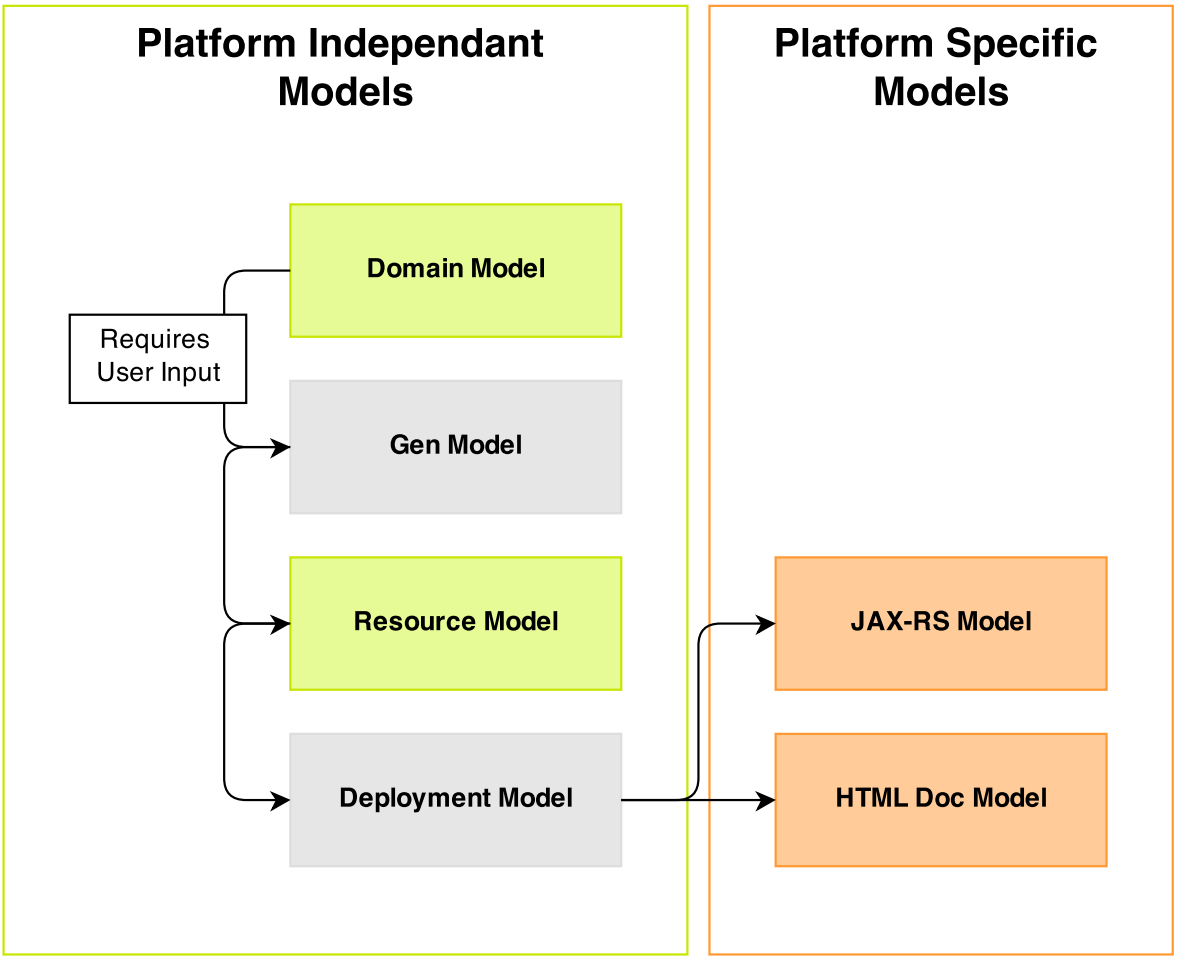

Figure 3.1.: Eclipse RESTModeling Prototype Models

3.1. Platform Independent Models

3.1.1. Domain Model

The Domain Model provides the first and initial possibility to graphically model objects of the domain the web service should be related to. Within this model the user can design object to object relations with an aggregation. The object itself can contain several attributes of a primitive type and methods (Figure A.1). Every modeling object except the aggregation has an optional author and description. At first appearance this looks quite similar to a UML class diagram, however it does not cover the same functionality. Hence it is more like a subset of

UML.
The resulting model acts as the basic application interface but does not include any REST characteristics yet. To get a closer approach to a REST web application some more user input is needed which intentionally is not covered by the Domain Model.

3.1.2. Gen Model

The term Gen Model is an abbreviation of Generic Resource Model since it is not the final version of a Resource Model. This model is an intermediate stage between the domain and the Resource Model. It is generated by certain choices a user makes during the transformation process of a Domain Model. Therefore it influences the transformation of the Domain Model into the Resource Model. To summarize, the Gen Model is a Domain Model enhanced with user input which will be transformed into the Resource model. So there is no direct transformation between domain and Resource model.

Currently there are two wizard-steps that have to be performed to transform a Domain Model into a Gen model. Due to the fact that POST is not an idempotent operation, the first step is to define what kind of CRUD strategy should be applied for every Domain Model object. Depending on this decision certain stubs for GET, POST, PUT and DELETE are created in the resulting Resource Model. The different choices are as follows:

Create Empty Resources, Update Them Afterwards If this option is selected and a client wants to create a new `Resource A` by requesting a superior `Resource B` two steps have to be performed (Figure 3.2). First of all the client sends a POST request to create a new resource to the location of `Resource B`. This request contains no special data input at all. The server creates a new empty `Resource A` and returns its location in a response header field with the HTTP status 201 (created). After receiving the location of the empty created `Resource A` the client requests this location via HTTP PUT and sends data for this resource. The web service updates the resource for the given location and returns in case of success the HTTP status 200.

This strategy should be used if the creation of a new element should be ensured and additional communication overhead is not crucial. In case the update of the method fails it can be repeated until it succeeds without creating new filled resources. Additionally empty, not updated resources can be deleted after a certain amount of time to reduce redundancy.

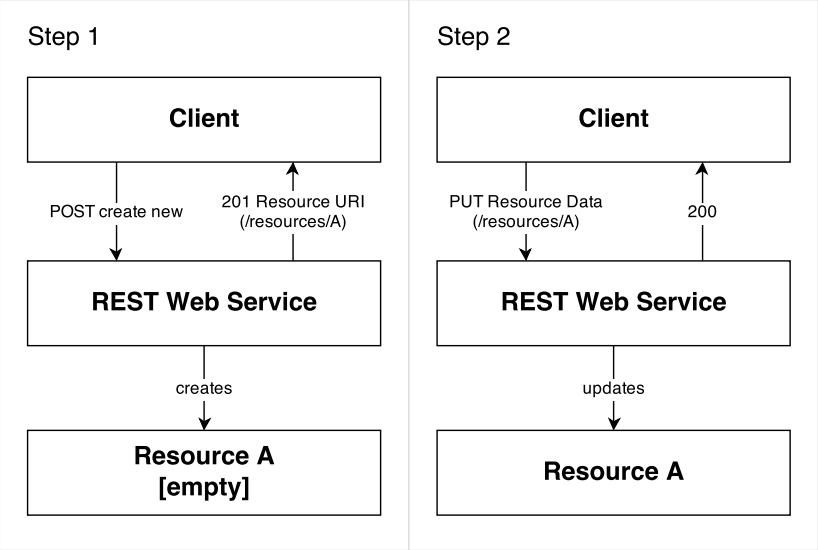

Figure 3.2.: CRUD Alternative I

Create Filled Resources Assuming the same precondition as for the first option (creating a new `Resource A` by addressing a superior `Resource B`) only one step has to be done for this type of strategy (Figure 3.3). The client sends a single POST request to the location of `Resource B` already containing data for the new `Resource A`. The web service tries to create the new resource with the provided data and returns the HTTP code 201 with the location to the newly created `Resource A`. In case the process fails before returning the location of the new resource the client does not receive a response. Due to a missing reaction from the server, the client would most likely retry its request and generate the new resource for the second time. In case this process fails again this might end up in a huge garbage of data at the server side after some time.

This strategy does not have the communication overhead of an additional request, but it might cause issues regarding data integrity and redundancy.

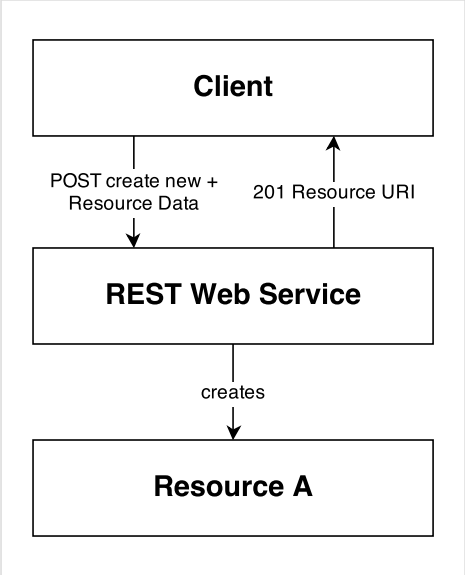

Figure 3.3.: CRUD Alternative II

No HTTP Methods In case of selecting this strategy resources will be generated within the resource model but they will not contain any HTTP methods at all.

The second step of the wizard allows to influence the creation of artificial `ListResources` to export the aggregation function from a Domain Model object to an additional object in the Resource Model. There are only two selectable options for this step, but are depicted in detail below.

Create Additional `ListResources` In case additional `ListResources` should be created, every aggregating relation of the Domain Model ends up in an artificial `ListResource` construct. The aggregating object (`Object A`) and the aggregated object (`Object B`) in Figure 3.4 are transformed directly into a `SimpleResource` element of the Resource Model. The aggregating link is transformed into a `ListResource` which is referenced by the resource object (`Resource A`) that was transformed out of the aggregating Domain Model object (`Object A`). Additionally it aggregates the the transformed `Resource B` and manages its instances. Requesting and creating objects of the type `Resource B` has to be done by addressing the `ListResource`.

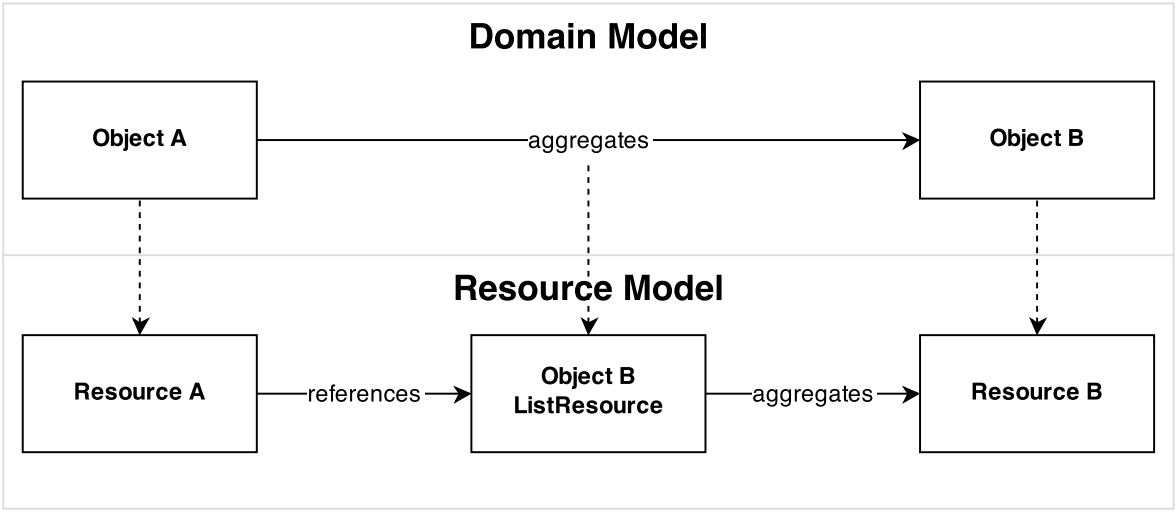

Figure 3.4.: Mapping A Domain Model Aggregation To The Resource Model

Do Not Create Additional `ListResources` Not creating additional `ListResources` still transforms both objects to `SimpleResources` but the aggregation is still done by the former `Object A` (now `Resource A`). Creating new resources and retrieving a list or a certain set of resources has to be done by addressing `Resource A`.

After applying all custom features the final Gen Model file is created and transformed into the Resource Model.

3.1.3. Resource Model

The root of a Resource Model instance is the ResourceDiagram element (Figure A.2). The Resource Model itself consists of `SimpleResources` and `ListResources` which are either connected by an `Aggregation` or a `Reference`. Every resource has a subset of methods whereby every method is identifiable by a type (`GET, POST, PUT, DELETE, OPTIONS, HEAD`). These methods have different lists of media types which they can consume and produce. In case a method needs a parameter that has to be passed to perform an action, a list of parameters can be set for it. `ListResources` can be used to handle a list of `SimpleResources` and already contain, in case they are auto-generated, HTTP methods with querying parameters to ease requests for subordinates of this list.
The Resource Model diagram editor allows to graphically add and alter HTTP methods for every resource which is equal to a domain object before the transformation, except any added list resources from the interim stage (Gen Model). By defining consumption and production types, different representations of a resource can be realized. After completing this model, the

applied mappings of the HTTP methods to resource elements or additional, new objects can be transformed into the Deployment Model without any further user input.

3.1.4. Deployment Model

The Deployment Model provides a generic basis for the URI structure of the application interface by mapping resources to a single user specific link. Dynamic parts in URLs are identified by curly brackets ({ }) and can be used to query for attributes related to the Domain Model.

3.2. Platform Specific Models

The platform specific models are needed for transformations of the application abstraction into a specific desired piece of code. Currently there are two different models that can be used to get a final "executable" result, the HTML Doc Model and the JAX-RS Model. Both of the models are used by a code generator to create a JAX-RS web service with empty stubs and the related documentation in HTML. The corresponding Ecore diagrams of the two platform specific models are kept simple, thereby it is not necessary to illustrate them in here.

4. Deficits Of The Prototype

The current state of the REST Modeling system is still in need of improvement. To state the deficits of the current system a show case is used, which should be possible to design by using this tool. The example describes several interactions that a customer can perform while interacting with a coffee shop. Figure 4.1 describes an abstract structure of this model. To maintain simplicity every order has a list of `Beverages` and respectively a single `Receipt`. In case a beverage needs a customized flavor, a set of different `Additions` should be optionally selectable. After adding all necessary beverages to the order it can be payed and the customer receives a receipt. This chapter covers the listing of issues that are in need of improvement by utilizing every view and model from top down which the REST Modeling tool provides.

4.1. Modeling The Domain Model

The current toolbar of the graphical domain editor provides three objects (`Object, Attribute, Method`) and a single link (`Aggregation-Link`) to connect them with each other. An `Object` contains `Attributes` and `Methods`, whereby a `Method` and an `Object` can contain `Attributes`. A detailed graphical structure is depicted in Figure A.1. The multipurpose usage of the `Attribute` modeling object (highlighted in Figure 4.2) leads to the first inconsistency. Actually a `Method` should contain proper parameter objects to distinguish between an `Object's Attribute` and a parameter which can be passed to the `Method`. Additionally not every property that applies to an `Attribute` is valid or reasonable for a method parameter (e.g. property: unique). As a result modeling the `pay()` method for an `Order` object for the given show case would be sloppy.

If two modeled objects have some sort of relation or connection the `Aggregation-Link` can be used to connect them. Since not all the object-to-object relations are aggregations (show case: `Order - Receipt`), a link type for associations is missing. Hence an accurate distinction between a real aggregation and a simple one-to-one relation is not possible its inevitable to use the `Aggregation-Link` to connect them. The example in Figure 4.2 forces an aggregation between the `coffeshop` and `Bar` objects instead of a simple reference or association.
Furthermore the concept of cardinalities was implemented by adding them to the property list of an `Aggregation-Link`. However they do not show up in the graphical editor yet.

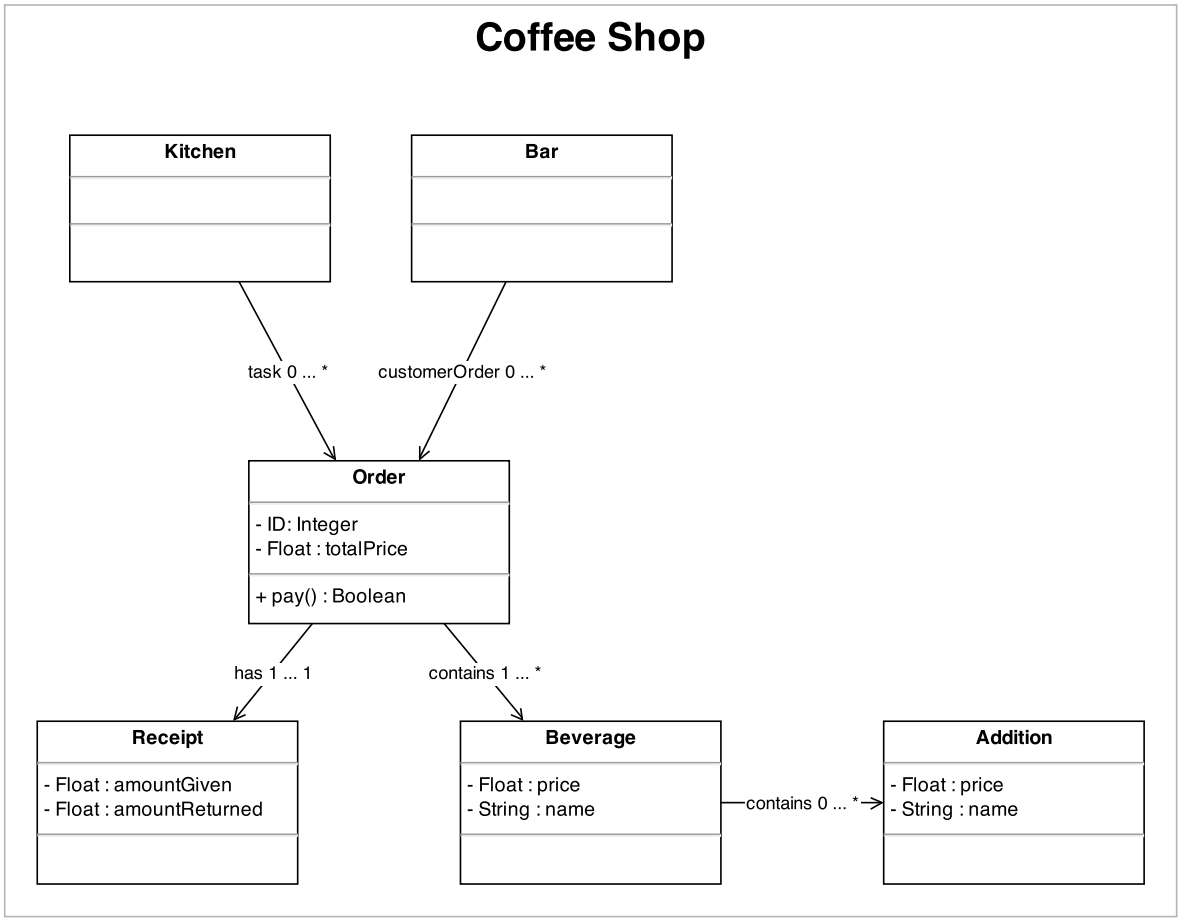

Figure 4.1.: Coffee Shop Show Case

4.2. Modeling The Resource Model

After transforming a potential Domain Model into a Resource Model every Domain Model object is mapped to a `SimpleResource`. The intermediate Gen Model allows creating `ListResources` for aggregated links, in case the user demands it. The graphical editor provides the two mentioned resource types, methods which act as HTTP methods for a resource and related parameters that can be used when invoking a HTTP method. Resources can either be connected via a `Reference` or `Aggregation` link.
Regarding the given show case, the resulting Resource Model of the transformation has a lot of inconsistencies. A real root resource is not generated automatically, whereby every object which has no incoming edge in the Domain Model graph becomes a root or “entry point” for the web service. Discovering a complex web service graph would be partly impossible if not

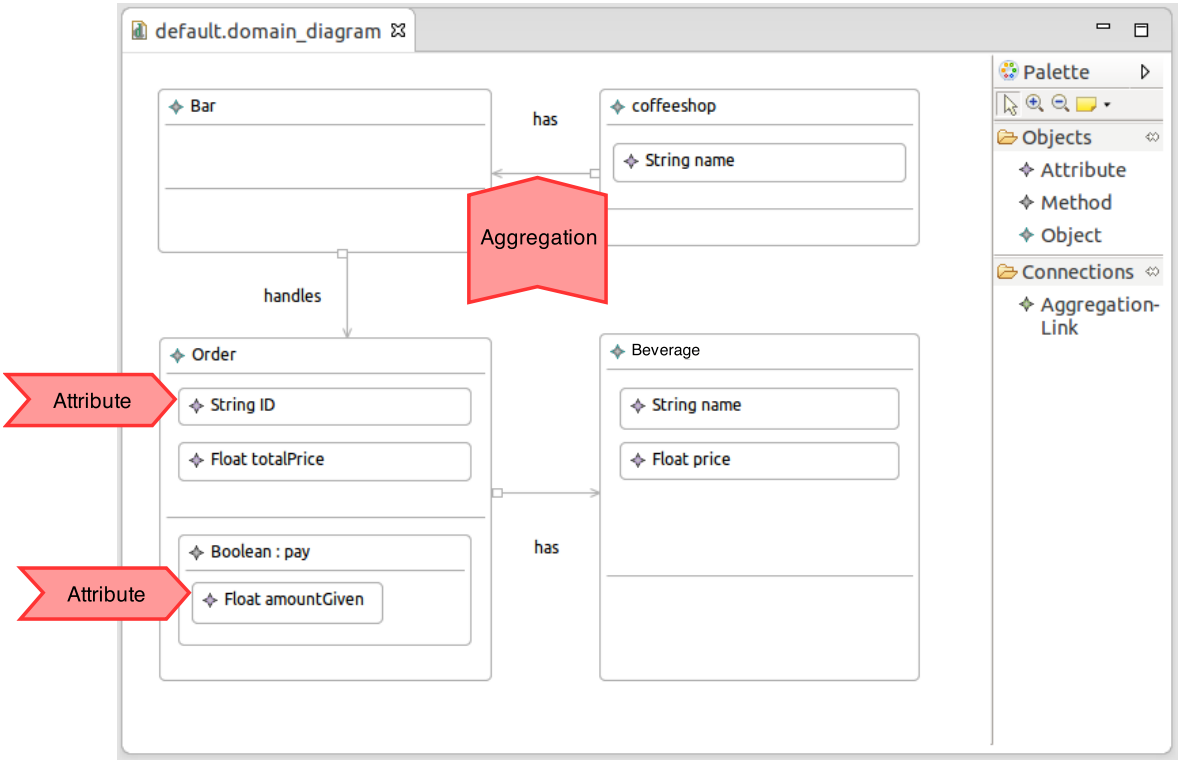

Figure 4.2.: Domain Model Deficits Illustrated By An Example.

every root resource is already known.
Even though the Resource Model provides two different ways of connecting resources, the transformation process is not capable of identifying a reference in the Domain Model from a real aggregation. This results in an additional `ListResource` for every newly transformed `SimpleResource`, in case the user specified wizard input demands it. The missing possibility of a simple one-to-one relation and the fact of a “forced” aggregation influences the outcome of the Domain Model transformation and requires a lot of manual modeling work to achieve it in a Resource Model.

The methods within a resource that represent HTTP methods are specified by a type which can be set in their properties. All methods have the same class specification, whereby every method has a certain subset of properties which do not fit to the method description specified by the Network Working Group ([FIa99]). A full property list contains produce and consume entries for requests and replies independent of their method types. So in case a method type does neither have a body part in the reply nor in the request it would be possible to set a consume or produce property. HTTP methods which are currently supported and match that

inconsistency are OPTIONS and HEAD. Thus a proper and precise modeling of performable interactions with a resource are not possible.

At a closer look the only obvious connection between the Domain Model and the generated Resource Model are the `SimpleResources`. `Attributes` and `Methods`, as well as related `Parameters` are ignored during the transformation process and not inserted into the Resource Model in any way. Any information from the Domain Model which is necessary to design the Resource Model has to be looked up in the domain diagram file. In case the `pay()` method of the `Order` domain object should be somehow realized within the Resource Model, several approaches can be used. The invocation of the domain method with its parameters can be mapped to any arbitrary resource method using the given resource parameter objects, but there is no distinct way. Due to these missing structural guidelines a lot of effort has to be done to build, parse and process generic parameters that are sent to a resource method (HTTP method) to achieve a functional method call.

4.3. Modeling The Deployment Model

The Deployment Model is used to specify URLs to the given resources from the Resource Model. The meta model structure of the Deployment Model combines a Resource Model element with a string or mapping element which acts as a local identifier for this resource. If a web service has a specific root (e.g. `/coffeeshop`) and a subsequent resource `BarResource` has the URL `/bar`, it would be possible to send requests to `/coffeeshop/`**`bar`**. In Figure 4.3 there are two ways to address the `OrderResource` resource (or, if desired, an overlying list). The first one as a subsequent resource from `KitchenResource` and the second one as a subsequent resource from `BarResource`. Since there is only one property field to specify the URL, it would not be possible to have an individual name for each incoming edge. This reveals a wrong fundamental approach of the Deployment Model. Instead of mapping resources to URLs, every linking element from the Resource Model should have a separate URL to provide individual access to the modeled resources.

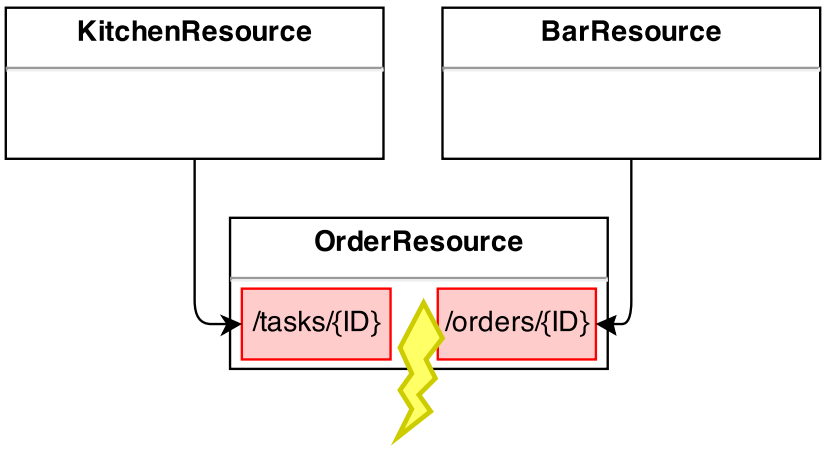

Figure 4.3.: Deployment Mapping Inconsistency

When filling the URL mapping parts for every resource (prospectively links) the user has to know all the Domain Model details for that resource to create dynamic URLs, if needed. The Deployment Model editor like every other model editor is not coupled or related directly to others. This results in a bad usability caused by an increased overhead for looking up relations or attributes while editing or setting up URL mappings.

4.4. HTML Documentation Generation

After finishing the URL mappings an HTML documentation can be generated by documenting elements of the resource and Domain Model. The documentation lists all resources of the Resource Model and their URL mappings specified in the Deployment Model. A detailed view of a single resource contains metadata (author, comment, ...), as well as the supported HTTP methods describing their consumption and production types. Every entry of a resource has a separately listed but linked domain object, in case it exists. A relation between a Domain Model element and the corresponding resource element is not visible.
Further the HTML documentation should be a reference document for a single resource's details and relations, hence an absolute URL for every given resource should not be needed. The web service itself should provide links to linked resources in the response to provide service exploration which is led by the server side or chosen by the client side. The additional Domain Model information should be combined and integrated into the given related resource information.

4.5. REST Compliance Of The Result

In case all models are completely modeled and defined the RESTModeling application allows creating a standalone REST web service. This web service provides several Java method stubs for the modeled HTTP methods to implement the missing logic for every resource. Although real content is missing the service should be usable in a “RESTful” manner. To verify what kind or level of “RESTfulness” the outcome of the current system matches, the features of the JAX-RS web service will be mapped to the requirements made by the Richardson Maturity Model from page 19. In the following every requirement of each level beginning at the initial level zero to level three is analyzed. A level is only valid if all conditions of every predecessor level and those from the level itself apply to the application.

Level 0 The initial Level requires the use of the HTTP transport with at least one of the two HTTP methods GET or POST. Since the current system does not support any other protocols besides HTTP it is obvious that this condition is fulfilled, even though it is theoretically possible to remove all the HTTP methods when designing the Resource Model of an application. In this specific case no interaction would be possible at all, due to missing stubs in generated code. Nevertheless for this assessment the responsible user or designer should always focus on a correct and runnable application. Intentionally bad usage of the system is not assumed.
The second requirement is a single service endpoint that can be addressed for requests. The transformation of the Domain Model into the Resource Model maps domain objects with no incoming edge to resource objects tagged as a root resource. The existence of multiple endpoints is possible, but there is at least one. Thereby the second requirement for this level is fulfilled and the outcome is valid for this level.

Level 1 The only additional condition to fulfill this level is that resources are accessible by different URIs. By using the given transformations from domain to resource and resource to Deployment Model every object of the Domain Model is at least mapped to a single resource or optionally to a list of resources managed by a superior list resource. After combining them into a corresponding Deployment Model every resource has its own URI sequence or subsequence.

Level 2 To fulfill level two of the maturity model multiple HTTP methods have to be supported by the web service. The assumption that the designer wants to build a REST compliant application applies here as well. Basically there are six available HTTP methods which can be used in the graphical Resource Model editor: POST, GET, PUT, DELETE, OPTIONS and HEAD. Since the required methods for the operations Create, Read, Update and Delete are the first of the four provided, the web service will have a guaranteed REST compliance for level two. The additional methods OPTIONS and HEAD are just optional features that can implemented in the service stubs and do not further influence the compliance.

Level 3 The provision of HATEOAS (Hypertext As The Engine Of Application State) is the final requirement. It ensures that links to other resources, alternatives or representations are sent as meta information when requesting the service. These links can be either wrapped in a compartment in the message body or returned to the client in the header field of the response. Using HATEOAS makes the discovery of a service more self explaining, just by following a link in a response. The current result of the application does not return any type of links at all, neither in the header nor wrapped in the message body. Consequently a currently generated service does not fulfill the final condition for level three.

As stated, the present outcome of the code generation is not completely REST compliant regarding the given model. Chapter seven of the original thesis ("Zusammenfassung und Ausblick") [Sch13] already lists the introduction of this feature as a required precondition to gain a "RESTful" application.

4.6. List Of Stated Issues

In the following, the issues that were mentioned above in this chapter are outlined to provide a rough overview. In chapter 5 all model enhancements, structural improvements and code changes to mend the deficits of the current system will be described in detail.

1. Domain Model
   a) Avoid Ambiguous `Attribute` Objects.
   b) `Aggregation-Link` as the only link type.
   c) Missing cardinalities.
2. Resource Model
   a) Missing single root resource, entry point.
   b) Every Domain Model link results in a `ListResource`.
   c) (HTTP) Methods contain redundant properties.
   d) Missing integration of Domain Model methods.
   e) Missing integration of Domain Model attributes.
   f) Missing visible connection in the graphical modeling editor with the Domain Model.
3. Deployment Model
   a) Multiple URL mappings of a single resource are not possible.
   b) Automatically generated URL properties.

4. HTML Documentation
    a) Missing integration of Domain Model information into a single resource's details.
    b) URLs are not needed in the documentation, references should be supported.
5. REST Compliance
    a) The generated outcome is not REST compliant according to Richardson Maturity Model level three.

5. Enhancements

As shown in chapter 4, the current state of the `REST Modeling` system is still in need of improvement. Major as well as smaller changes in model the structure and the code have to be made to provide a better usability and quality of the outcome when modeling a web service. This chapter covers all necessary steps to improve the listed deficits. To provide a structural approach of every enhancement the list of issues from section 4.6 will be processed in detail step by step. A solution to a deficit does not cover a code based description of the improvement. Most of the time they describe a general abstract approach which will be realized and implemented in chapter 7 (Implementation).

5.1. Domain Model

The strongest inconsistency of the Domain Model is the insufficient definition of its meta model. A lot of details are simply omitted which results in a bad usability and in a badly formed model structure.

Deficit 1a.) Avoid Ambigous Attribute Objects

To avoid the usage of an `Attribute` element as a member of an `Object` and a `Method`, a new domain modeling object `Parameter` is added to the Domain Model. The new Object has a type, a name as well as the additional comment and author properties of every object. This `Parameter` can only be added into `Method` objects.

Deficit 1b.) Aggregation-Link as the only link type

Basically there is the need of two different link types for different semantic use cases. The first type is semantically very close to a directed association from the UML notation. It is used to connect two objects with each other and to describe a one way “knowing” relationship. For example the `Order` and `Receipt` object from the show case could use a unidirectional association to navigate from an order to a receipt. This relationship is not necessarily a one-to-one relationship. By combining cardinalities with the association link multiple elements can be associated to a single superior object (e.g. `Bar` - `Order`).

The other type which is needed is some sort of the UML notation's composition. A composition represents a special dependency of two objects. It is used in case the linked object is required as a part of the whole and could not exist on its own. A perfect example for this is the relation between a `Beverage` and a set of its `Additions`.
An addition could not exist without a beverage but it is sometimes needed to refine it. Therefor an addition could be a shot of milk, some cream or a special additional taste.

So far the semantic differences for the domain modeling are made clear, the resulting changes for the Resource Model will be described in the upcoming section. Both of the new link types have all the `Aggregation-Link's` properties since there is no difference in their usage.

Deficit 1c.) Missing Cardinalities

Even though cardinalities were added to the model it was not possible to set or edit them in any way. To provide a correct syntactic and semantic cardinality feature the super class of every linking object (`Aggregation-Link, Association`) has to have a `minOccurence` and `maxOccurence` property and the cardinalities which are related to a link have to appear in the graph.
Using the UML notation regarding multiplicities, a connection between two single objects (e.g. `Order` and `Receipt`) would result in a link label displaying a simple 1 at each end of it. This is necessary due to the UML notation's support of bidirectional connections between objects.

Since the Domain Model only consists of unidirectional links it is reasonable to support a simplified concept of cardinalities between two Domain Model objects. The generic structure of a cardinality label is `[Link.Name] [Link.minOccurence] ... [Link.maxOccurence]` whereby the start of an edge is always assumed to be a single entity. The properties `Link.minOccurence` and `Link.maxOccurence` identify the minimum and maximum amount of entities of the target object.

The resulting possible interconnections for a Domain Model graph are depicted in Figure 5.1. The one-to-one association for `Order` and `Receipt` would be read as: An `Order` object has minimum one and maximum one (=exactly one) `Receipt` object. Accordingly the one-to-many Composition: A `Beverage` object has minimum none and maximum unlimited `Addition` objects.

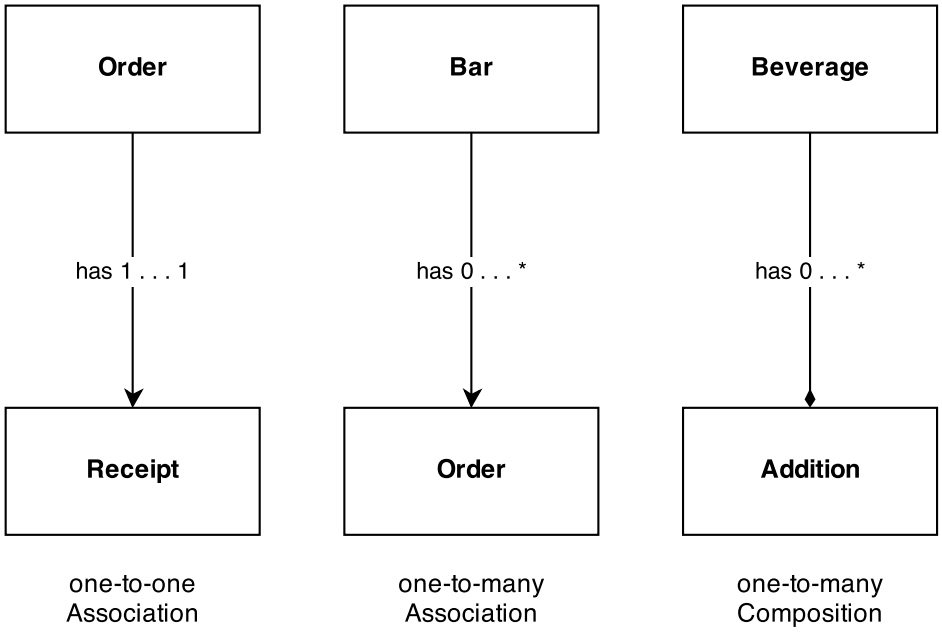

Figure 5.1.: Interconnection Possibilities In The Domain Model.

5.2. Resource Model

Since there are a lot of deficits regarding the modeling process with the Resource Model the following enhancements are one of the key aspects that have to be improved.

Deficit 2a.) Missing single root resource, entry point

Currently every Domain Model object with no incoming edge is transformed into a root resource element. This may result in a set of resource elements which all act as a root node. The need of an absolute superior root node is necessary to provide a proper discovery of the REST web service.

Inserting a root node into the Resource Model can be forgotten by a developer. Therefore an inevitable way is needed so the root will be created automatically. An appropriate environment to set details for a root node is the Domain Model Diagram root. The root node of a Domain Model diagram is a separate object which is not visible in the toolbar but is automatically available as soon as the model is created. Besides the current properties author and comment, a new property (`application-context`) is added to define a context for the elements of a Domain Model. It needs a default value in case the user does not set it. This

property is used to create a single and absolute root node in the Resource Diagram. By altering the transformation process from the domain to the Resource Model, a root resource has to be generated using the application-context. The additional new root resource has to link to every former root of the Resource Model to provide its precedence.

The artificial root resource is required for the enhanced Deployment Model structure which is described below in this chapter. Due to the fact that the new Deployment Model assigns URLs to linking elements instead of resource elements the root resource is used to create an incoming link for every transformed “root” object of the Domain Model. The Resource Model root element is omitted when creating the platform specific JAX-RS Model. It is used to populate the base URL property of the web service.

The introduction of the application context property results in a change for every model up until the final generated web service code.

Domain Model The Domain Model is used to specify the application context as a property of its absolute root element (`DomainDiagram`).

Gen Model The Gen Model stores the `DomainDiagram` property in its absolute root due to its participation in the web service generation.

Resource Model When transforming the Gen Model into the Resource Model the application context property is used to create a single superior resource element.

Deployment Model The application context property is not represented in the Deployment Model at all. Only the links from the Resource Model are added, which have its corresponding resource element as a source.

JAX-RS Model The generation of the JAX-RS Model uses the value from the Gen Model as the `baseURL` property for the web service web.xml file (url-pattern).

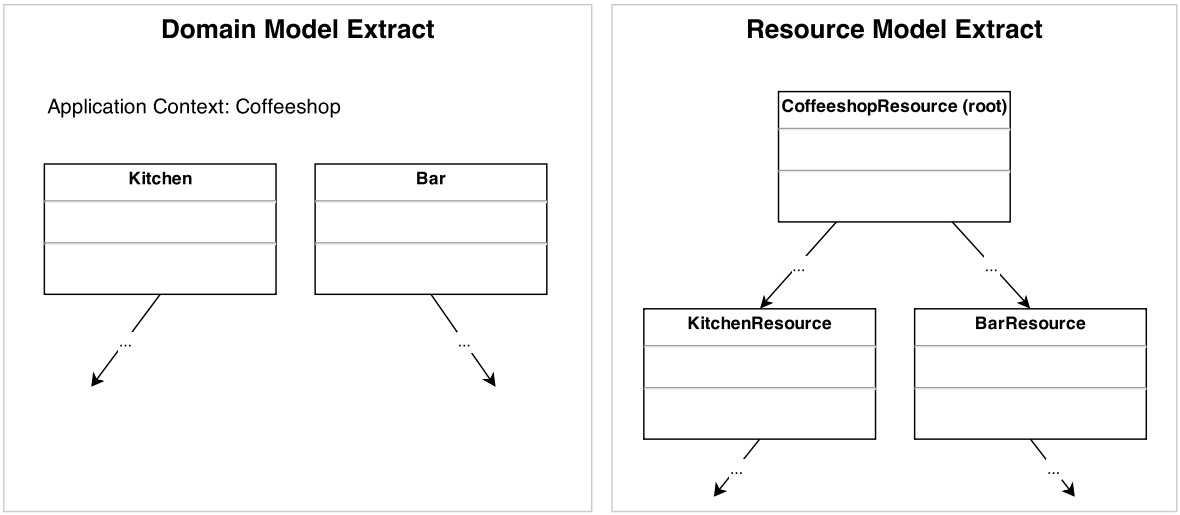

Figure 5.2.: Addition Of The Resource Model Root Resource.

As an example Figure 5.2 depicts this scenario for the show case of this thesis. The two objects `Kitchen` and `Bar` are simply translated into resource elements. The additional root resource is named after the application context of the Domain Model. The additional links to the former Object Model roots are the actual enhancement. The resulting Deployment Model would provide URLs for the following list of links. Without this enhancement the first two links would not be definable.

- CoffeshopResource (root)-To-KitchenResource
- CoffeshopResource (root)-To-BarResource
- KitchenResource-To-[SuccessorResourceA]
- BarResource-To-[SuccessorResourceB]

Deficit 2b.) Every Domain Model link results in a ListResource

The enhanced set of links and the newly introduced cardinalities build the major prerequirement for this improvement. The transformation into the intermediate `Gen Model` has to differentiate between a Domain Model `Composition` and `Association` regarding their given cardinalities. The additional user selection of an additional `ListResources` influences the transformation of the links significantly.

The Resource Model is a completely different view designed on different semantic assumptions

and rules. As a consequence the Resource Model does not support any cardinalities for links because it would not be reasonable. Links are only used to visualize the existence of a connection between two resources. The type of link (`References` or `Aggregations`) is only needed to make a visible sematic difference between them. It is irrelevant for further transformations (e.g. Deployment Model, HTML PSM Model).

Due to the different semantics of the domain and Resource Model specific rules have to apply for a transformation process depending on the user selection. There are two main identifiers for a transformation of Domain Model objects and their interconnection. The first one is the user selection which determines whether to create an artificial `ListResource` in between two transformed resources (former domain objects) or not. The second one is `maxOccurence` property of a cardinality which influences the amount of resources that are linked. Combining them results in four possible cases for each type of Domain Model connection (one-to-one Association, one-to-many Association, one-to-many Composition and for the sake of completeness, one-to-one Composition, although the use of this construct is very unlikely). Since the type of link does not influence the transformation process Figure 5.3 only depicts transformation rules for an `Association`. Every rule can be applied to a `Composition` in the exact same way.

In case the user wants to create additional `ListResources` a simple one-to-one association (or composition) between objects is transformed into a one-to-one reference between resources. In contrast, a one-to-many association or composition is transformed into a custom construct. The source of the connection will be transformed into a `SimpleResource (BarResource)`. The target is transformed into a `ListResource (OrderListResource)` which has an incoming reference from the transformed source resource (`BarResource`) and an outgoing aggregation link to another `SimpleResource (OrderResource)` which represents a single transformed target resource itself. The `ListResource (OrderListResource)` can be used to get a certain or even filtered list of `OrderResources` and to create new `OrderResources`.

If no additional `ListResources` are required the one-to-one link transformation for Domain Model links obviously remain the same. The one-to-many connection will now result in a simple aggregation between two `SimpleResources (BarResource, OrderResource)` whereby new instances of an `OrderResource` are created via accessing the aggregating parent (`BarResource`).

Deficit 2c.) (HTTP) Methods contain redundant properties

To improve the deficit that there is no strict separation between the different model types, lots of additional changes have to be made in the modeling structure. Every HTTP method in the Resource Model can be queried with parameters, but actually not all the HTTP methods need parameters or provide a request body consumption or response body production. So as an approach to specify each HTTP method properly it is necessary to introduce a certain

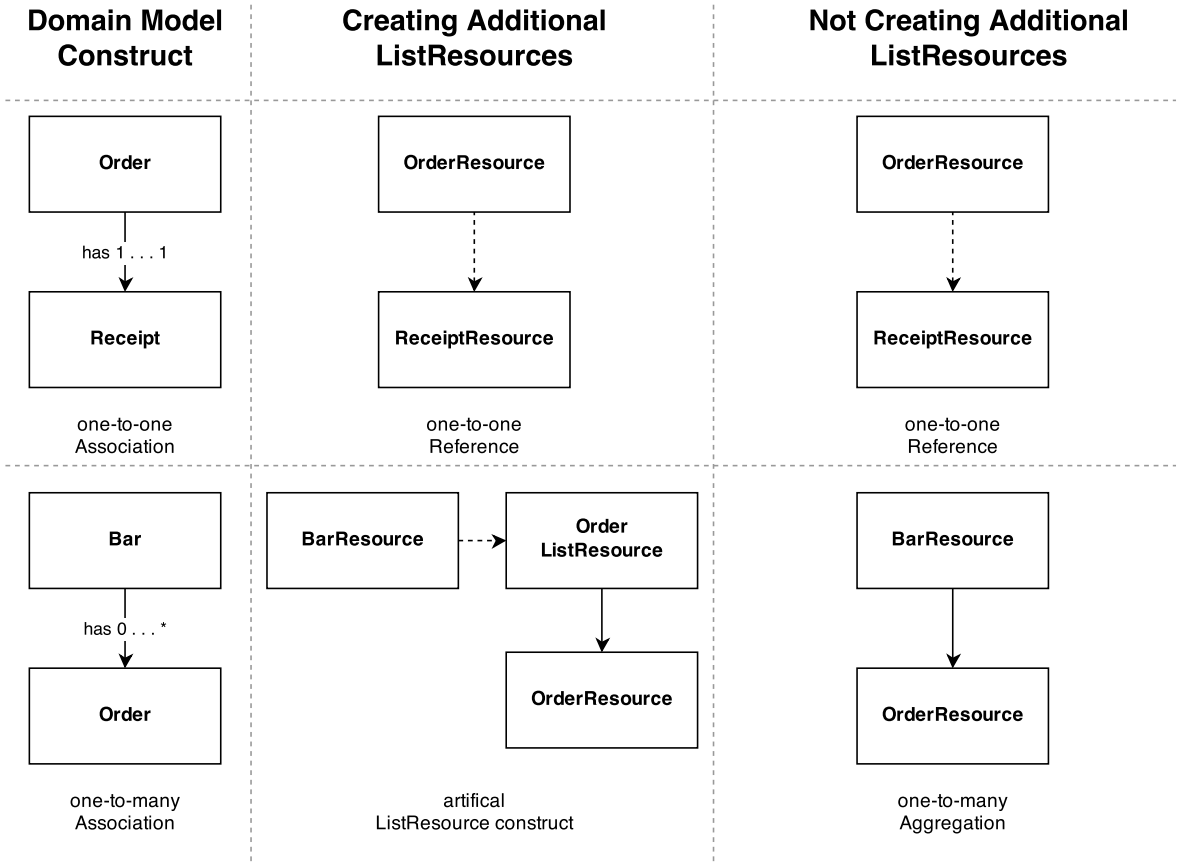

Figure 5.3.: Domain-To-Resource Link Transformation Rules

resource object for every type. The previous element `Method` which combines and represents all types is replaced by a `GetMethod, PostMethod, PutMethod, DeleteMethod, OptionsMethod` and `HeadMethod`. All of them need to be specified with different attributes and different child elements.

The current structure of a Resource Method object is defined as follows:

The properties `Consumes` and `Produces` in Figure 5.4 both have a list of Mutlipurpose Internet Mailextension (MIME) types. These MIME types which refer to the body of the request or response are assigned individually. The method type is replaced by the new resource type itself. Since the return code property depends on the processing of a request it can not be set in general for a `HTTP` method type.
In the following a reasonable set of properties and child elements (parameters) are mapped to

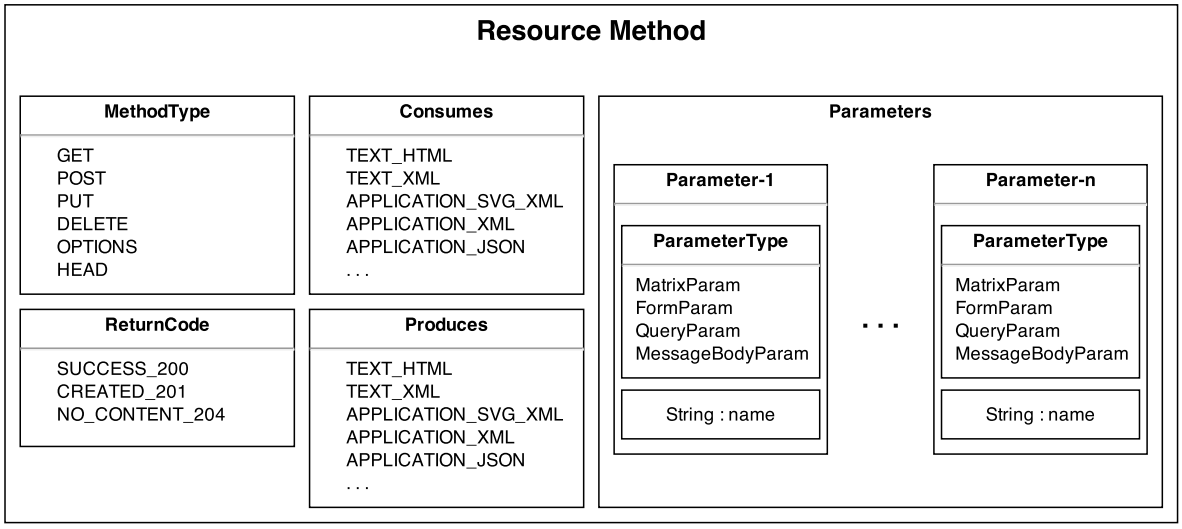

Figure 5.4.: Resource Method Structure Of The Current System

each method, regarding the stated characteristics of the Network Working Group ([FIa99]) and assumptions about the modeling process using the RESTModeling tool.

GET A `HTTP GET` method needs parameters that can be passed in a request. Resources which require a parameterized `GET` method are for instance `ListResources` with query parameters to return a specific subset of its subordinated list. Further this type of method needs no property of a MIME type list for incoming input due to a missing body in an HTML `GET` request.

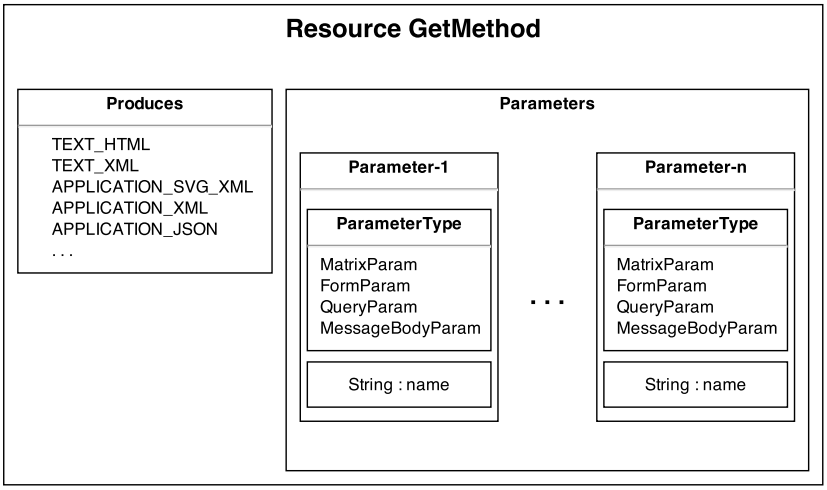

Figure 5.5.: Prospective Resource GetMethod Structure.

PUT In case a `PUT` method updates a resource and returns it in a desired format to the user it may consume and produce data for and of this resource in different MIME types. Consequently it is necessary to provide parameters and the possibility to set consuming and producing mime types for the request and response body.

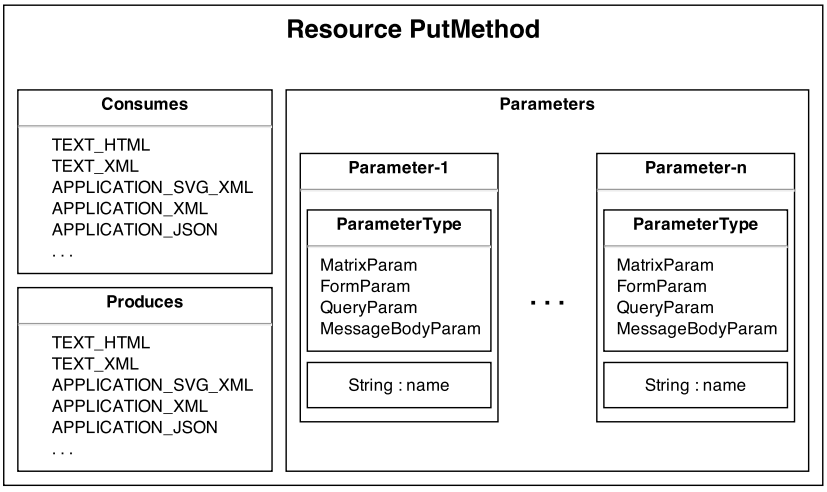

Figure 5.6.: Prospective Resource PutMethod Structure.

DELETE According to [FIa99] a `DELETE` method may return an entity in its response body. For the context of this thesis we determine that this is not necessary. Hence the produce

MIME type can be omitted for this method. In contrast a property list of MIME types for incoming request parameters is required.

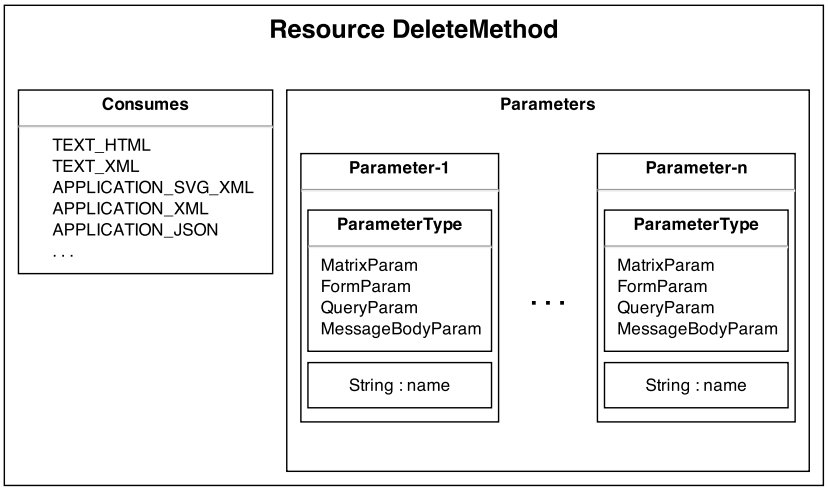

Figure 5.7.: Prospective Resource DeleteMethod Structure.

HEAD, OPTIONS The `HEAD` method is used to return metadata of a entity which is available when addressing a certain URL. All the information is stored in the header whereby a consume or produce property as well as parameters are not necessary. The `OPTIONS` method represents an information interface for navigating further starting at the requested URL. It can, but should not contain a message body containing this information. For the purpose of the modeling tool `HEAD` and `OPTIONS` do have the same properties - none.

POST The exception is the `POST` method which will be described in detail in the next subsection.

2d.) Missing integration of Domain Model methods

Integrating Domain Model methods and Domain Model object attributes are approached separately. Yet, there was no possibility to integrate any existent method from the Domain Model into the Resource Model. The previously omitted structure of a `PostMethod` is used to integrate them. Every method which is defined in the Domain Model will result in a separate

interaction in a resource object embedded in a `PostMethod` object. Since a HTTP `POST` provides the possibility of a request and response body it can be used for the input and output of a method.
A related example is the `pay()` method within the `Order` domain object. It is mapped to a single `OrderResource` as an `Interaction` object. Assuming that there are multiple methods for a Domain Model object, every object will be transformed in a single `Interaction`. As stated, every interaction acts as a service end point. Thus, it is necessary that the user can define a list of consuming and producing MIME types. Parameters of a Domain Model method are not necessary for the graphical modeling of the Resource Diagram. They do not influence any structural coherences but are required to perform proper requests with a `PostMethod` resource.

The `Interaction` object is the only object where the previously removed attribute `Return Code` (HTTP return code) is applicable. HTTP codes are used by default for every interaction with a HTTP operation. The codes are returned to identify the state of the request which was sent to the server. By providing a return code in case of a successful method execution the service developer has the choice to customize a HTTP response. Thereby it is possible to individualize the behavior of a web service for every modeled interaction. Since the return code can only be modeled for a successful method execution, other HTTP operations (GET, PUT, DELETE, ...) do not have this property. They have to be implemented manually in the web service code.

Compared to other invocable HTTP methods an `Interaction` needs no separate parameter in the modeling toolbar. Information and data is passed via the request body. The different consumable MIME types have to be supported by the corresponding code in the JAX-RS web service. In every request addressing this code, the body has to contain the name of the interaction, a list of method parameters (specified in the Domain Model) and the value for each parameter.
The resulting structure for a resource can be seen in Figure 5.8. It contains a subset of methods including a `PostMethod` embedding `Interactions`.

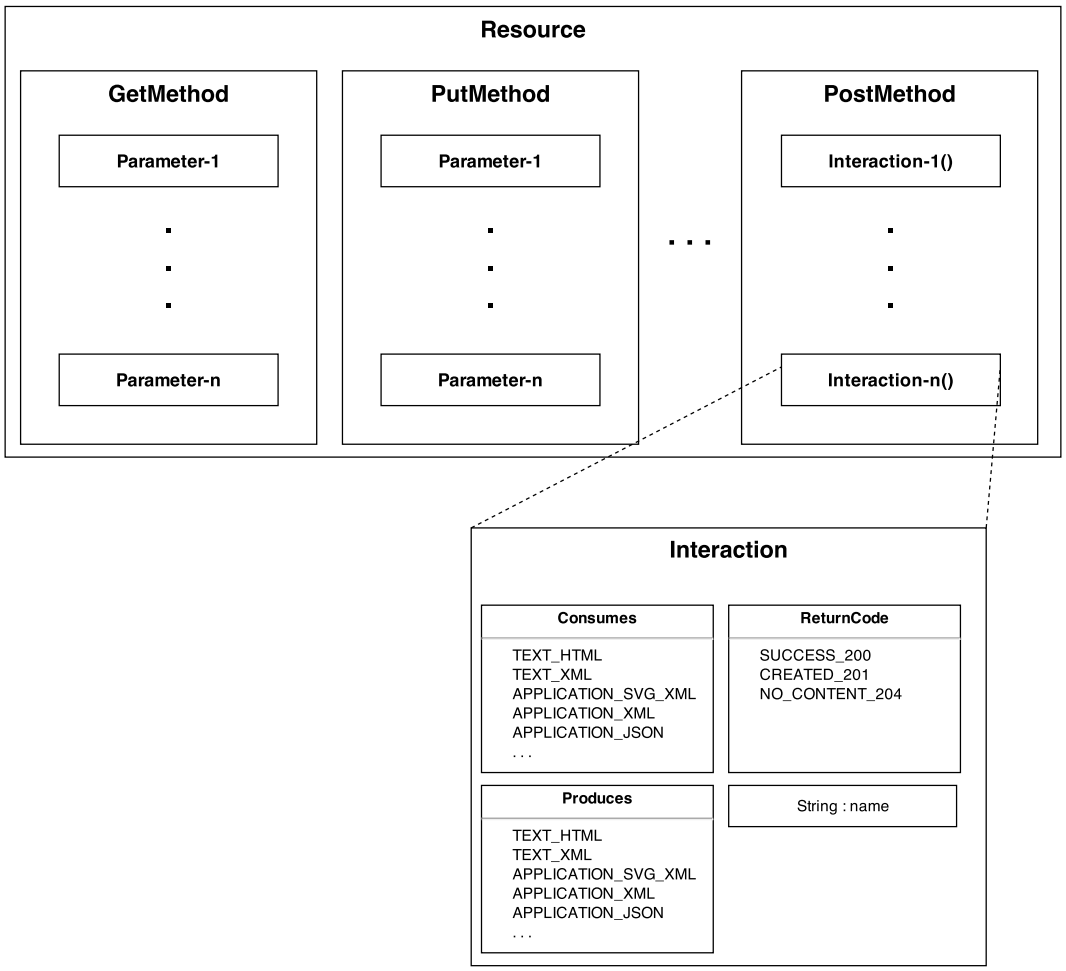

Figure 5.8.: Improved Structure Of A Resource With A Detailed View On Interactions.

Figure 5.9 represents an example for an invocation of the `pay()` method from the coffee shop show case. The highlighted parts colored in green represent the `Interaction` itself (former Domain Model method). The `Consumes` attribute (highlighted gray) of the interaction predefines a set of accepted MIME types. In this particular case the example request uses an XML structure. The root element of the XML request is the interaction containing a subset of parameters. These parameters are only modeled and defined in the domain method (orange) due to their irrelevance for the Resource Model. Using the given domain and Resource Model extract a request can be built very easy when addressing the `OrderResource`.

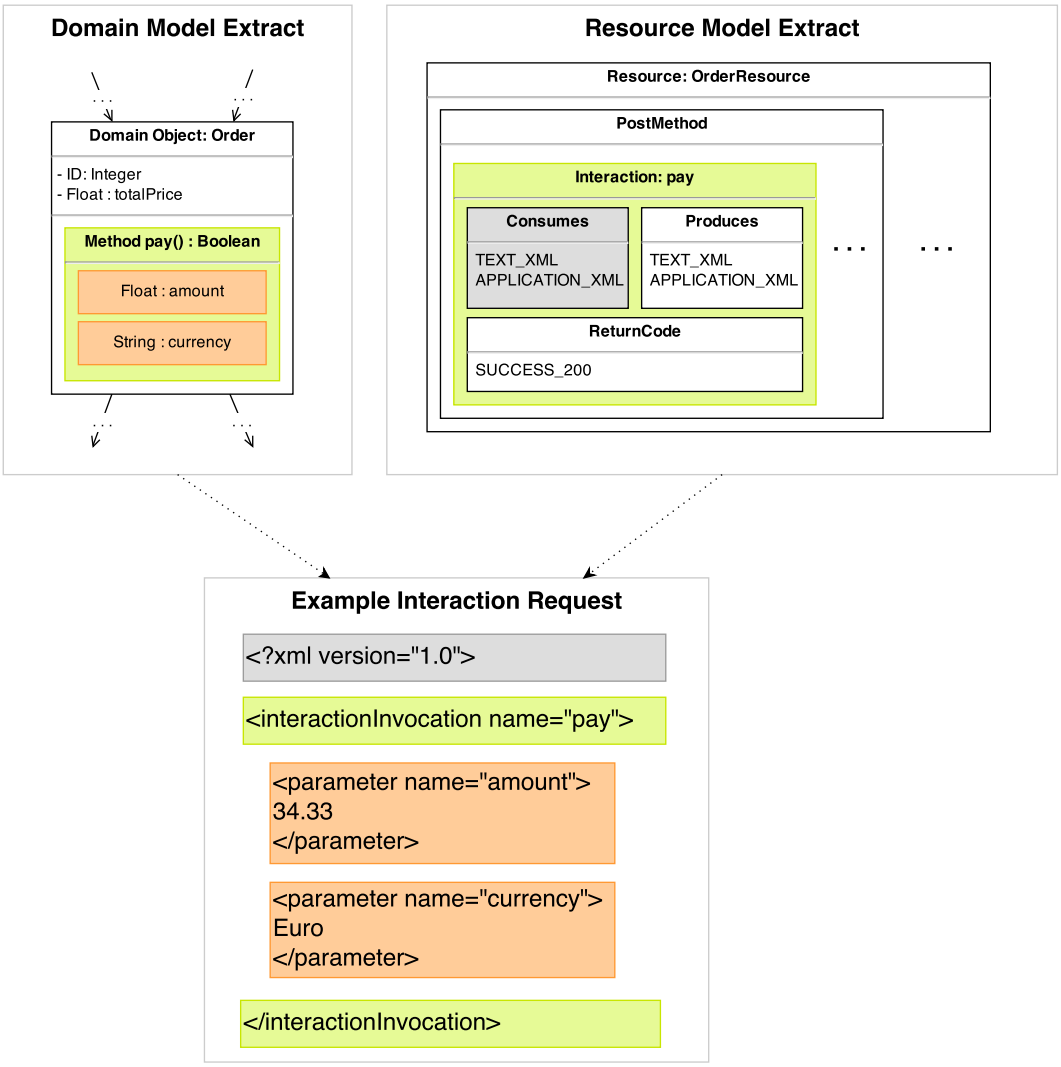

Figure 5.9.: Interaction Request Structure Demonstrated With An Example.

2e.) Missing integration of Domain Model attributes

Since the Domain Model methods are mapped to Resource Model `Interactions`, the attributes of domain objects still need to be included. Processing attributes of a domain object in its corresponding resource object causes no real benefit or improvement and there is no need for a `SimpleResource` to own Domain Model attributes.
An advantage of transforming attributes from the Domain Model into the Resource Model is gained when using them within a `ListResource`. Precondition for the integration of Domain Model attributes is that a simple object from the Domain Model is transformed into a

`SimpleResource` (e.g. `Order` object) and the superior `ListResource` during an intermediate step. This `ListResource` links and manages every `SimpleResource` object that was transformed. When addressing this list a set of `SimpleResources` is returned. The set can be modified during the request by start and stop parameters (Integer). These parameters are similar to the LIMIT function of the MySQL syntax for a SELECT query [Gie12]. The start parameter indicates the index of the first element, the stop parameter indicates the last index of a resource element in the whole list. In case of wrongly set index bounds the returned set will contain no entries at all.

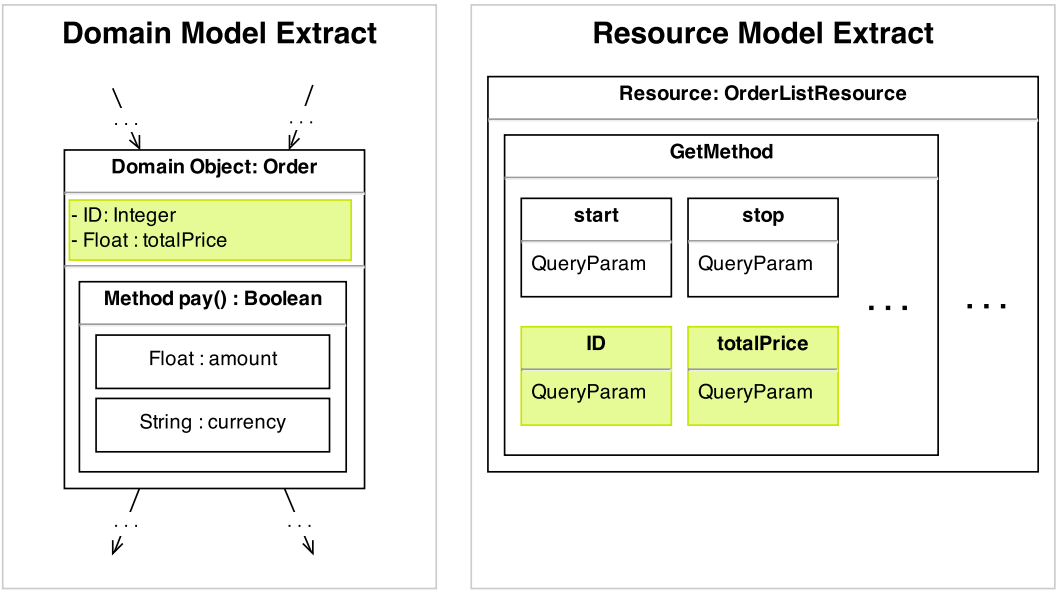

Figure 5.10.: Domain Attribute To ListResource Parameters Mapping.

By enhancing the transformation process, attributes of the related Domain Model object are transformed into query parameters of its related list (Figure 5.11). Hence it is possible to not only influence the number of results that are returned, the list can be filtered for certain entries by querying with values for the added parameters (attributes). The compulsory precondition for this feature is the addition of `ListResources`.

2f.) Missing visible connection in the graphical modeling editor with the Domain Model

As stated above a connection between the Resource Model and the Domain Model is essential to design a proper web service. The auto-generated graphical modeling editor for Resource Models has no real connection to the Domain Model editor, so any relation between a resource

and a domain object is not obvious. The corresponding ecore models are treated as two single and separate models by the EuGENia [1] engine. Even though it is possible to import the models into each others meta models a graphical relation between them is missing. Only ETL scripts which transform a domain into a Resource Model links them, by producing a graphically independent result (Resource Model) from a source (Domain Model) that can be used for further modeling steps. Consequently the Resource Model is an enriched and altered version of the Domain Model and provides, based on the mentioned enhancements in this chapter, already a few integrated and transformed Domain Model elements (`Interactions`, `Attributes`). Especially for the mapping of Domain Model `Methods` to `Interactions` it is important to know what kind of parameters the `Interaction` needs. The example invocation shown in Figure 5.9 is only possible having all the parameter information from the Domain Model and the name and MIME type information from the Resource Model.

Due to the auto generated editor there are only a few possibilities to integrate a decent visual connection into Eclipse. To display the related Domain Model information the property tabs are extended by an additional one. This new tab displays information which is retrieved from the Domain Model by focusing a certain resource graph element. Since not every object within a Resource Model graph has a related object in a Domain Model, information will be displayed in the following cases:

- The resource object has a related Domain Model object (e.g. `SimpleResource - Object`).
- The resource object is an `Interaction` and has a related Domain Model method.

In cases of a missing connection a simple message is displayed that indicates an absent counterpart in the Domain Model. Figure 5.11 depicts screenshots of the property add-on. The parts highlighted with a green outline represent the information that is displayed when focusing an interaction. Additionally the view provides sample data for a certain set of consumable MIME types to ease queries addressing the superior `PostMethod`. Accordingly, parts highlighted in orange are displayed when focusing a resource element.

[1] Eclipse EuGENia is a tool that automatically generates all models for a graphical editor from an Ecore meta model.

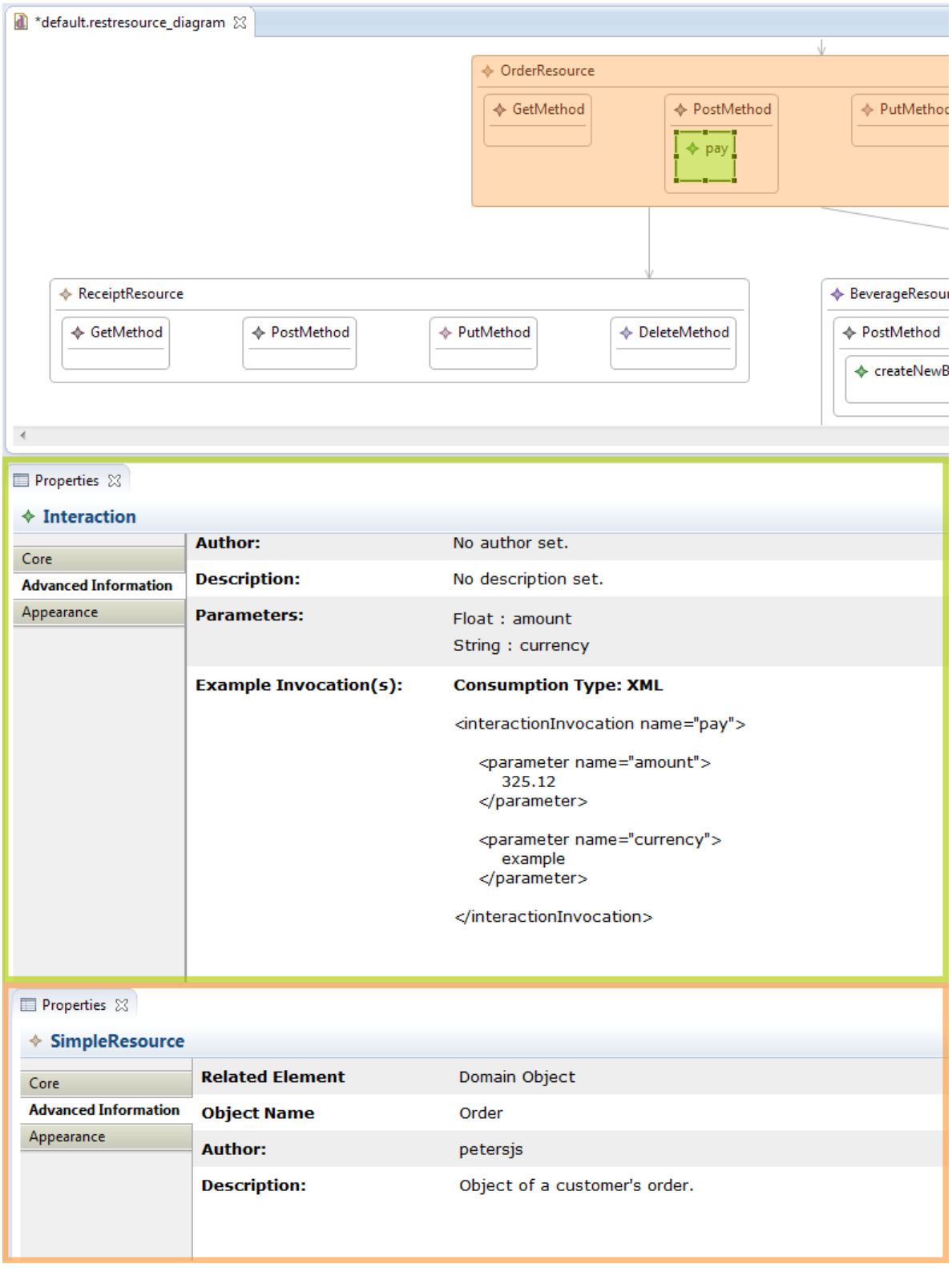

Figure 5.11.: Property-Addon In The Resource Diagram.

5.3. Deployment Model

3a.) Multiple URL mappings of a single resource are not possible

The fact that the Deployment Model rests on a wrong assumption concerning URL mappings leads to a complete restructuring of the model itself. The previous model which connects a resource object (`SimpleResource, ListResource`) to a user defined URL will be altered into a model that connects a resource link (`Reference, Aggregation`) to a user defined URL. Thereby multiple mappings for a single resource are feasible (Figure 5.12) and the stated issue in Figure 4.3 can be now specified properly.

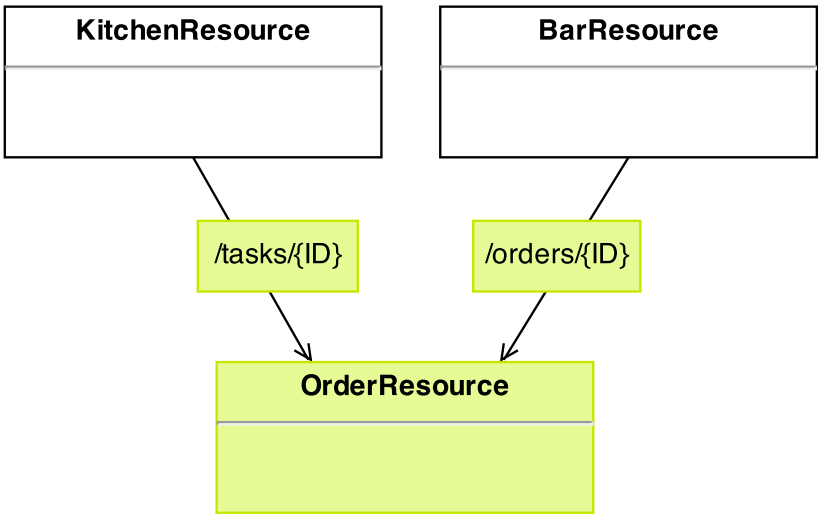

Figure 5.12.: New Deployment Mapping - URL To Link.

3b.) Automatically generated URL properties.

To provide a better usability and to improve the efficiency of the modeling process some small adjustments can be applied to the creation of the Deployment Model. The current way of defining a URL is done by a simple text input field. Manually added curly brackets indicate that the enclosed URL part is an unique attribute of the domain object which makes the URL dynamic. Likewise it is possible to omit any dynamic attributes and define a static URL part to a resource. Since the usability is not very user-friendly a conducted way of setting a dynamic or static URL to a resource is required. By automatically populating a static URL and a dynamic URL part, if possible, the usability is perceptibly improved. The behavior of the auto generation has to follow certain rules. In case a resource object has a related Domain Model object containing an unique attribute it can be used as a dynamic URL part. To illustrate the auto

generation of a URL the coffee shop example is used. Assuming that the `ID` attribute of the `Order` object is unique the resulting URL would be `/order/{ID}`. If `ID` is not unique or there is no unique attribute at all a generated URL would be `/order/`. Consequently a lot of effort can be saved by defining a proper Domain Model and automatically populate the corresponding URL into the Deployment Model. Nevertheless, the created URLs can be individually altered by the user. Applying any URL patterns the use of plurals for nouns within the URL have to be made by the user since the nouns are simply created by using Domain Model object names.

5.4. HTML Documentation

4a.) Missing integration of Domain Model information into a single resource's details.

Due to new POST interactions with a resource a lot more information is needed to perform correct requests. The current structure of a single documented resource is very simple. In case it has a linked Domain Model element it is just added on the bottom of the resource description. Building a correct request requires some lookup effort to browse through the unstructured documentation.

To increase the information content and its appearance every interaction from the Resource Model should be linked to its corresponding method in the Domain Model. Metadata as well as required parameters and example requests for certain content types support the user to understand coherences and request details. As interactions are the only interface where details are specified by the Domain and Resource Model it is currently the only reasonable part to provide detailed examples for a request.

4b.) URLs are not needed in the documentation, references should be supported.

Since the HTML documentation is only used to give details about a resource element the web service discovery is not part of it. Therefore any given URLs are from the Deployment Model not listed in it. Instead, unidirectional references between resources should be obvious and easy to follow. The root resource has to be tagged in any way so that the structure is understandable by following linked resources beginning at the root.

By now the HTML documentation has to document the following for every resource:

- Metadata (author, comment, name)
- Parameters for each of the HTTP methods (except POST)
- Interactions mapped to HTTP POST

- Examples for Interactions
- Relations or connections to other resources

5.5. REST Compliance

Improving the REST compliance of the resulting JAX-RS web service is the last major issue that has to be solved during the enhancement process. Currently HATEOAS (Hypermedia As The Engine Of Application State) are missing whereby the service discovery is impossible due to missing link elements in response headers or bodies. Links enable discoverability for a web service. Several servers providing that service can respond to any client at almost any stage because the client carries its state within its requests. The state can easily be mapped to a graph which is local respectively relative from a single resource's point of view. Combining all of these local resource graphs into a bigger graph, the structure of the web service can be ascertained.

An extract from a possible resulting Resource Model dependent on the show case is visible in Figure 5.13. To serve as an appropriate example the graph has an additional URL mapping for the given resource connections.

Listing 5.1 HATEOAS Example: OrderResource text/xml

```
HTTP/1.1 200 OK
Content-Type: text/xml

<?xml version="1.0"?>
<orderresource ID="2389" totalPrice="3.45" isPaid="true">
  <link rel="self" href="/orders/2389"/>
  <link rel="receipt" href="/orders/2389/receipt"/>
  <link rel="baverages" href="/orders/2389/beverages"/>

  ...

</orderresource>
```

A current request to the `OrderResource` via HTTP `GET /latestOrder` would return the resource with no further navigation possibilities. A connection to the `BaverageListResource` and the `ReceiptResource` would not be obvious. A required functionality would be that for every requested URL (resource) a set of links (if possible) are returned to the client. Since a resource can have multiple MIME types it is necessary to return a type conform link. Using a text/xml content type the links could be returned in the return message's body as described in Listing 5.1.

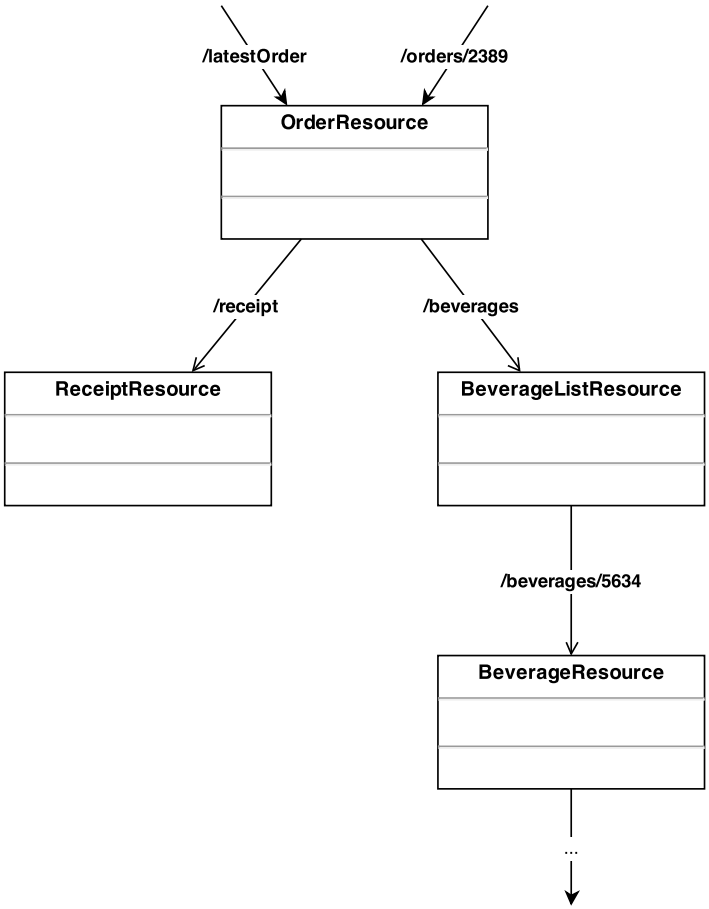

Figure 5.13.: Fully Populated Deployment Model

The provision of such a navigation forces changes in the generation of the web service source code. In exactly the same way as the resources are linked for the HTML documentation generation it is necessary to find all outgoing connections to other resources to provide the directed navigation for every node of a resource graph. In case a logical process influences the provision of certain links (e.g. `/receipt` is available after the order is paid) it has to be implemented in the corresponding code parts manually.

So far the link provisioning is limited to links resulting from the Resource respectively Deployment Model. A further feature of the HATEOAS enhancement is the pagination for lists. It allows navigating through a list of resources by providing links for a predecessor page and (if available) a successor page.
In case a `ListResource` is requested with query parameters (start and stop) to limit the results links can be provided by analyzing the amount of resources in this list by using the query offset.

This concept rests upon the assumption that the lowest bound of a start parameter is zero. Thus, the entries per page (page size) are calculated as follows:

$$\textbf{page size} = \text{Request Parameter } \textbf{stop} - \text{Request Parameter } \textbf{start} + 1$$

Using the page size, a generic way of building predecessor and successor links is listed in Table 5.1. For the sake of completeness the reference to the actual requested page itself is added as well. Every row in the table contains the link name and the start and stop parameters used for addressing this specific page.

Link To	**Start Parameter**	**Stop Parameter**
previous	Current **start** Parameter - **page size**	Current **start** Parameter - 1
self	Current **start** Parameter	Current **stop** Parameter
next	Current **stop** Parameter + 1	Current **stop** Parameter + **page size**

Table 5.1.: General Pagination Rules For Lists

Example `OrderResourceList`

Assuming that the `OrderResourceList` has at least 15 entries the web service's response to the request `/orders?start=5&stop=9` contains the links (previous, self, next) of Listing 5.2 to provide the desired pagination.

Listing 5.2 HATEOAS Pagination Example: OrderResourceList text/xml

```
HTTP/1.1 200 OK
Content-Type: text/xml

<?xml version="1.0"?>
<orderlistresource>
  <link rel="previous" href="/orders?start=0&stop=4"/>
  <link rel="self" href="/orders?start=5&stop=9"/>
  <link rel="next" href="/orders?start=10&stop=14"/>
  ...

</orderlistresource>
```

6. Technologies

6.1. Eclipse Epsilon

The whole system is based on Eclipse Epsilon[1] which is a special release of the regularly published Eclipse IDE. It has already pre-installed plugins to ease model driven development. It is a family of languages and tools based on the core of Epsilon, the Epsilon Object Language (EOL) which is a combination of JavaScript and OCL (Object Constraint Language). Using Epsilon provides simplification in code generation, model-to-model transformation, model validation, comparison, migration and refactoring. The overview in Figure 6.1 (based on the Eclipse Epsilon documentation) clarifies the composition of the two main parts EOL and ECM (Eclipse Model Connectivity). By combining them, languages for specific tasks can be extended from EOL. A full list of all available languages provided by Epsilon can be found on the documentary website [Eps14]. The REST Modeling plugins only use the Epsilon Transformation Language (ETL) and the Epsilon Validation Language (EVL) which will be described in the following.

6.1.1. Emfatic and EuGENia

Emfactic is a language to create EMF Ecore models in a textual form. An Ecore model file represents the meta model description of a specific model (e.g. Domain Model, Resource Model, ...). In case elements from other meta models (ecore files) are required, they can be imported within each other. The syntax is related to Java and provides a set of annotations for a better customization of the resulting GMF Editor. By describing and designing the editor model, the complexity is massively reduced. All previously described models (Domain, Resource, Deployment, ...) are described by such an EMF file, although not everyone of them provides a graphical modeling editor. EuGENia allows to easily transform those models within the context menu into Java source code. The code reflects at least the model structure of the Emfatic file and additionally the source files for the graphical GMF or plain textual editor, in case the model contains any GMF annotations. In Figure 6.2 all files and intermediate models are listed which are required to get a graphical editor out of the source file. Every single model can be created step by step by transforming predecessor models. Several custom changes can

[1] Eclipse Epsilon - http://www.Eclipse.org/epsilon/

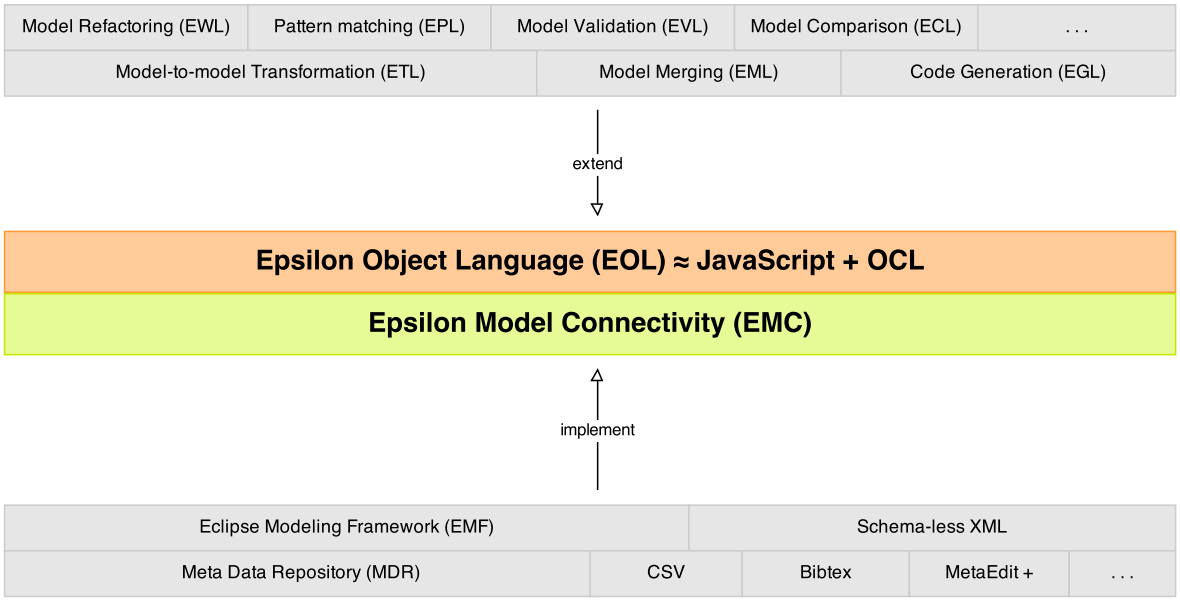

Figure 6.1.: Eclipse Epsilon Architecture Overview

be made in the Ecore and Genmodel model to have some further influence on the resulting editor. Alternatively, if no specific changes have to be made, the Emfatic file can be transformed directly into an editor and intermediate models are generated automatically.

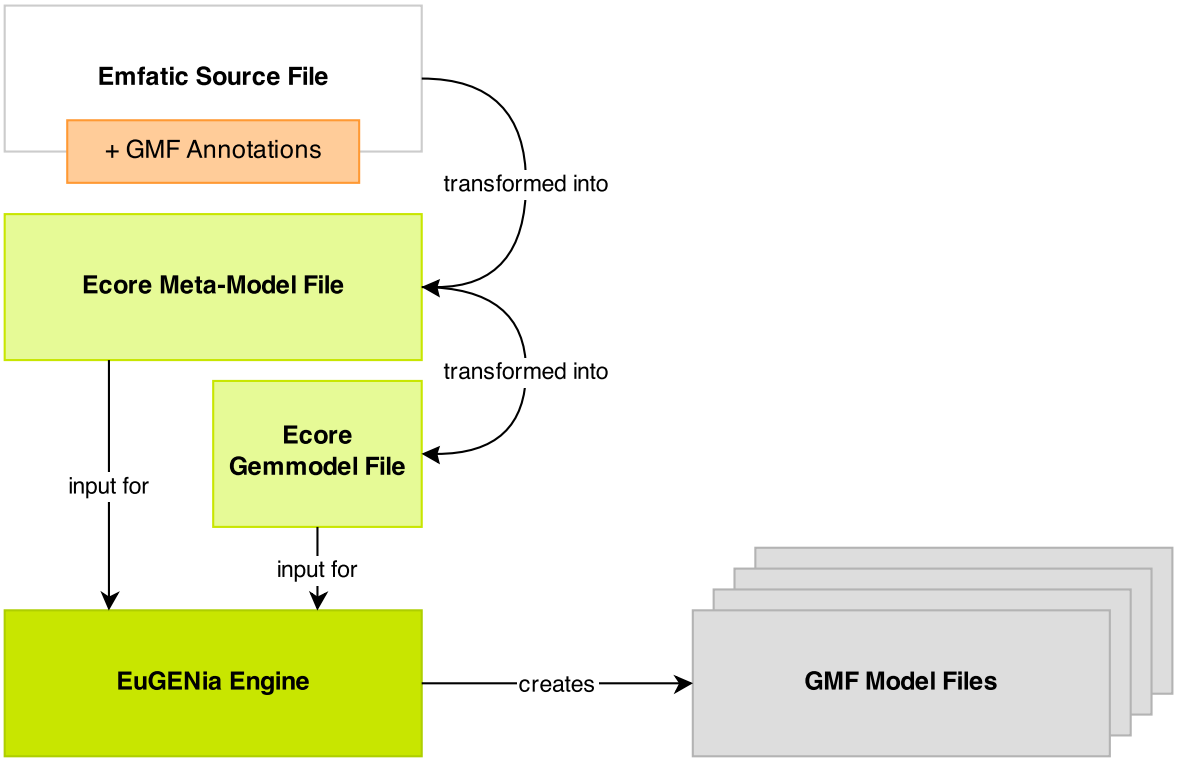

Figure 6.2.: EuGENia-Emfatic Model Stack

Listing A.1 is a simple example of a graphical model editor specified in an EMF file. In general it can be compared to a Java class with additional information for the GMF editor. It describes classes, abstract classes, enums as well as primitive and complex data types. Using the gmf.diagram() annotation special properties of the editor for that specific type of diagram can be set. In this example case the only defined property is the file extension for the model (model_extension), the actual data of the diagram, and the extension of the diagram itself (diagram_extension). To maintain simplicity there are only two classes of nodes (gmf.node(...)) and one type of connection (gmf.link(...)). Without the GMF annotations it would only be possible to generate an XML representation from the model. Generating the modeling tool out of this, it would be possible to have objects in the diagram linked with each other through connections and containing multiple attributes which are type of the given enumeration `AttributeType`. A graphical representation of a modeled example within this editor is Figure 6.3 and its related XML representation Listing A.2.

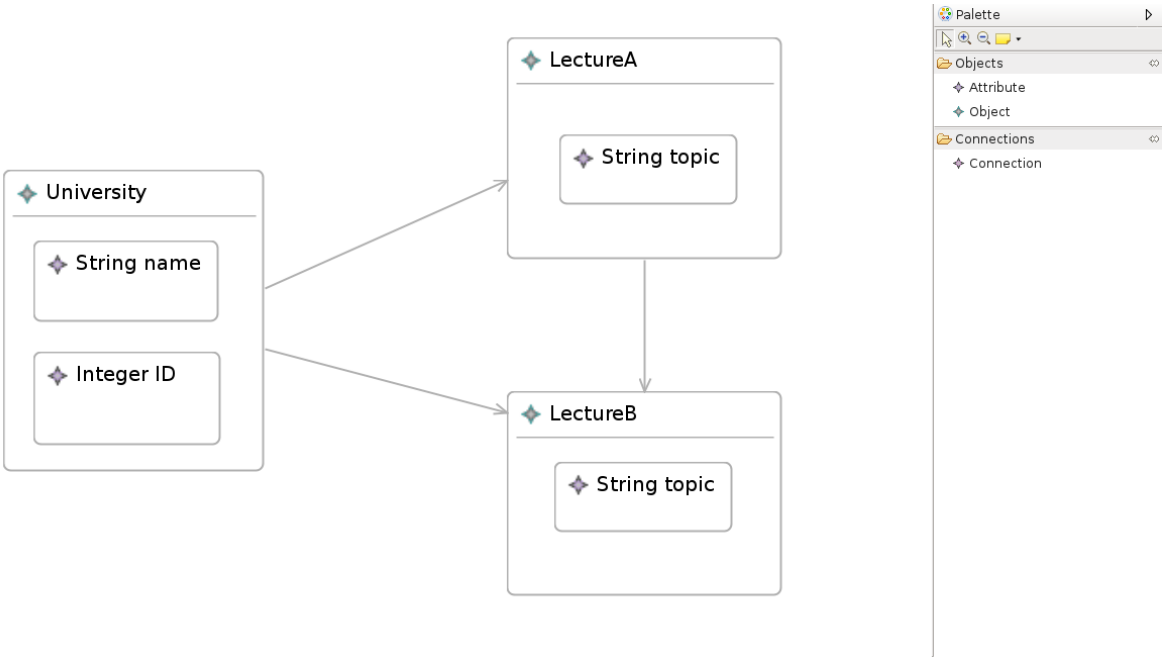

Figure 6.3.: Example GMF Editor Generated From An Example EMF Source File.

6.1.2. ETL - Epsilon Transformation Language

To transform the set of models from one to another, the ETL language is used. ETL combines declarative and imperative model transformation languages into a hybrid language. It is based on the abstract EOL language with certain singularities for the transformation process. A transformation file contains a set of rules and operations that have to be applied to the elements of the source and target model. The scripts can be invoked by the Epsilon-API or Ant-Script[2] targets. Currently all model to model transformations are invoked by a Java class which uses the necessary ETL files. As an example Listing A.3 describes a transformation of a SourceElement from a SourceModel into a TargetElement of a TargetModel. All rules in an ETL script will be processed in the same order they are listed. Operations can be invoked by those rules to perform additional actions. A rule consists of a source which has to be transformed into a target element and optionally a guard statement which allows ignoring a rule in case the guard is not active. Almost every transformation between the models of the REST Modeling tool is performed by these scripts: Domain-to-Resource, Resource-to-Deployment and Deployment-

[2]Ant-Script - http://antscript.com

to-Plattform-Specific-Models. The Gen Model (subsection 3.1) is the exception. It is build by Java code by combining the wizard input of the user with the designed Domain Model.

6.1.3. EVL - Epsilon Validation Language

To ensure a valid designed model, the Epsilon Validition Language is used, which is another task specific language (6.1) that extends the Epsilon Object Language. It allows validating instances of meta models in the Ecore Model Editor and in the GMF editor. Certain rules (constraints) define for a specific context what kind of properties or requirements have to be fulfilled to gain a valid model. The EVL file is connected to both types of editors by using Eclipse Extension Points. A simple example of such a validation file which validates an instance of the resulting model from Listing A.1 is outlined in Listing A.4.

6.2. JET - Java Emitter Templates

Generators for platform specific models or code have to be written individually. The usage of the Java Emitter Templates simplifies code generation to a very user-friendly level. JET allows creating templates for individual needs which can be easily filled with certain content to get a complete output file. The JAX-RS web service stubs and the HTML documentation are both created using JET.

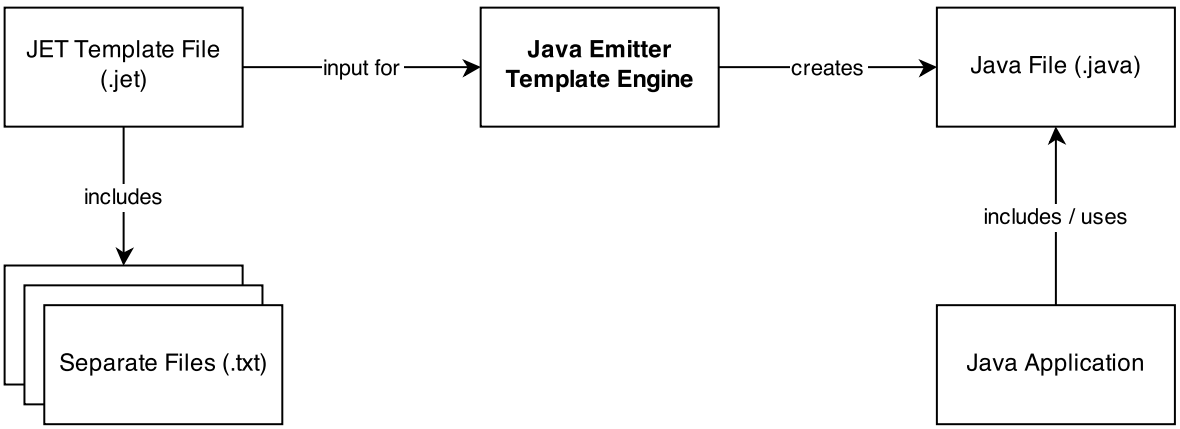

Figure 6.4.: JET Process Of A Single Template.

Figure 6.4 depicts a process for a single JET template. The template itself has to have the JET file extension (.jet). Within this file a class name has to be specified which will be used to name the resulting Java class after the creation process. To provide a structured development other files containing JET code can be included without any special statements. They are treated like regular code in the JET file but they need a different file extension (e.g. .txt) otherwise the Java Emitter Template Engine will cause an error. The JET files are automatically transformed into Java files which can be imported and used a Java application. Listing A.5 shows an JET file example of the HTML documentation's index page.

7. Implementation

This chapter describes the new architecture of the RESTModeling system as well as some individual aspects of the implementation. Further a comparison is drawn between experienced advantages and disadvantages of describing Meta models with the Eclipse Modeling Framework and transforming it into a Graphical Modeling Framework editor with EuGENia.

7.1. Overall Architecture

The performed enhancements described in chapter 5 affected some parts of the architecture of the system. Most of the modifications are part of already existing models or components. No further models were added during the process, only the existing ones were modified to enrich their information content and improve their meta model structure. Figure 7.1 shows the different components highlighted gray, used models light green, model-to-model transformations light orange and the resulting outcomes strong green.
The newly added property plugin is integrated into the Resource Diagram editor and receives its input from the code generator. Further details on this plugin are outline in section 7.4.
The `HTML-PSM-Model` is now created right out of the Resource Model and can be transformed into the documentation by the `Code Generator` afterwards. Apart from that, the architectural structure is unaltered from the original system.

7.2. Altered Model Structure

Most of the meta model structures are kept the same way as they were in the prototype. The Domain Model and Deployment Model have new elements or simply altered elements. However, the small enhancement (mapping URLs to links instead of resources) in the Deployment Model caused huge inevitable modifications to the Deployment Model and JAX-RS Model generation. The structure of the Gen Model was not influenced by this thesis at all and remains in the same state it was before.

The most structural changes happened within the Resource Model's meta model 7.2. The previously element `Method` is now a super class of the `PostMethod, GetMethod, PutMethod,`

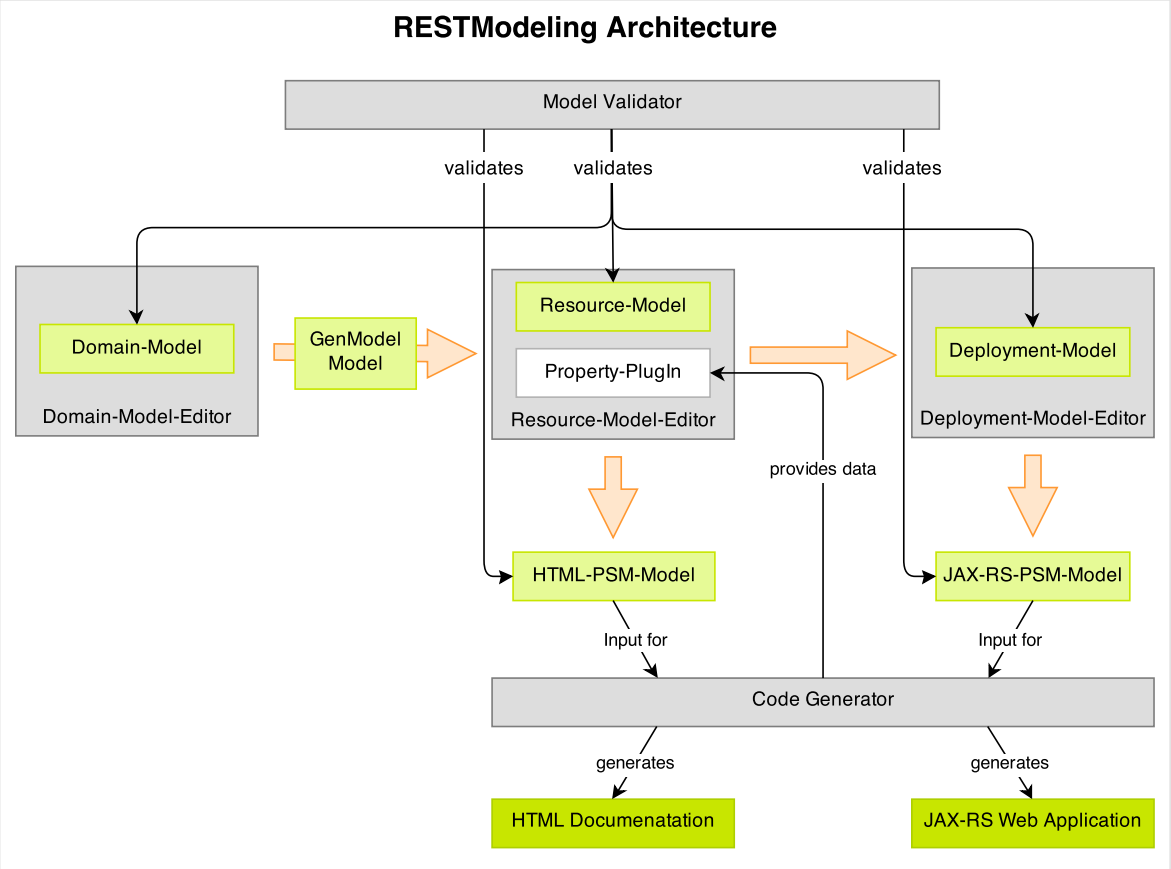

Figure 7.1.: New Architecture Of The RESTModeling Tool

`DeleteMethod`, `OptionsMethod` and `HeadMethod`. The individual specified properties and containment relations from section 5.2 are depicted in the bottom part of the meta model diagram. The newly introcuded object `Interaction` can be embedded in arbitrary quantity into the `PostMethod`.
The resulting effects of this structural modification are spread all over the system. Transformations from and into the Resource Model were altered as well as the generation of the documentation and the JAX-RS web service.

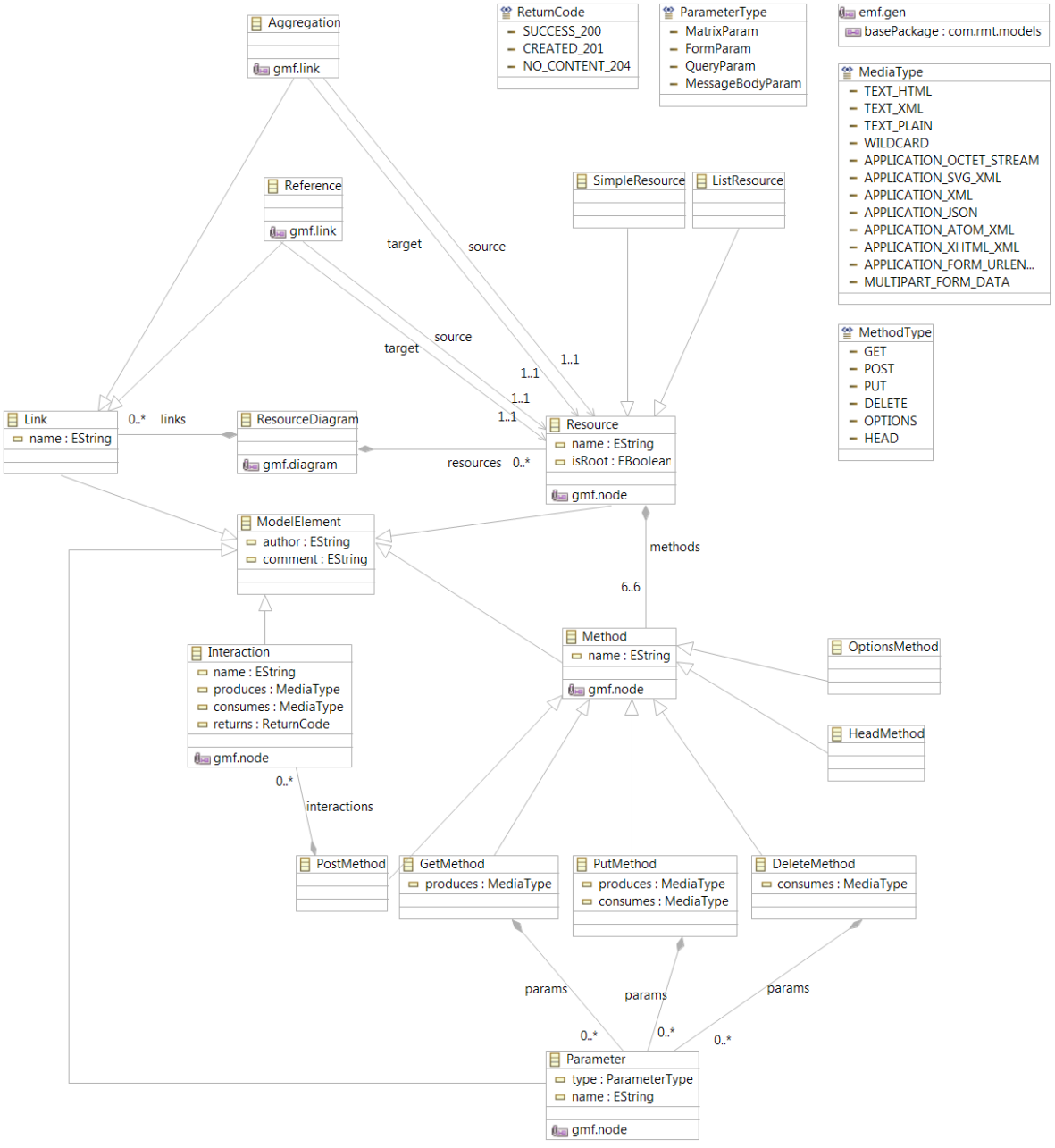

Figure 7.2.: New Structure Of The Resource Model

7.3. Code Generator - Example Invocations

The example invocations which are used within the Resource Diagram editor's property view and in the HTML documentation are completely based on the code generator. The generator project provides independent Java classes which can be imported and invoked from other Java projects or components to get a list of example invocations for consumption types of a Resource Model Interaction. Currently there are several classes for the generation of the JAX-RS web service, the HTML documentation and the property add-on for the Resource Diagram. They are created by the `Java Emitter Template (JET) Engine` by parsing their origin source file (.jet). Each of these jet files may include a set of template part files (.txt) which contribute to the whole template.

To generate example content into an already existing template file, the provided main file in the /templates/examples/ directory has to be included. This part results a simple HTML snippet containing the examples for a specific Resource Model Interaction (com.rmt.models.restresource.Interaction) related to a Domain Model Method (com.rmt.models.domain.Method). These two objects have to be passed to the superior JET template and forwarded to this part.

Currently there are only examples available for the media types APPLICATION_XML and TEXT_XML. In case the component is invoked by using a not supported media type the return will contain a list of types for which examples are available. Every set of media types e.g. XML, JSON or HTML is located in its own file that is included in the main part.

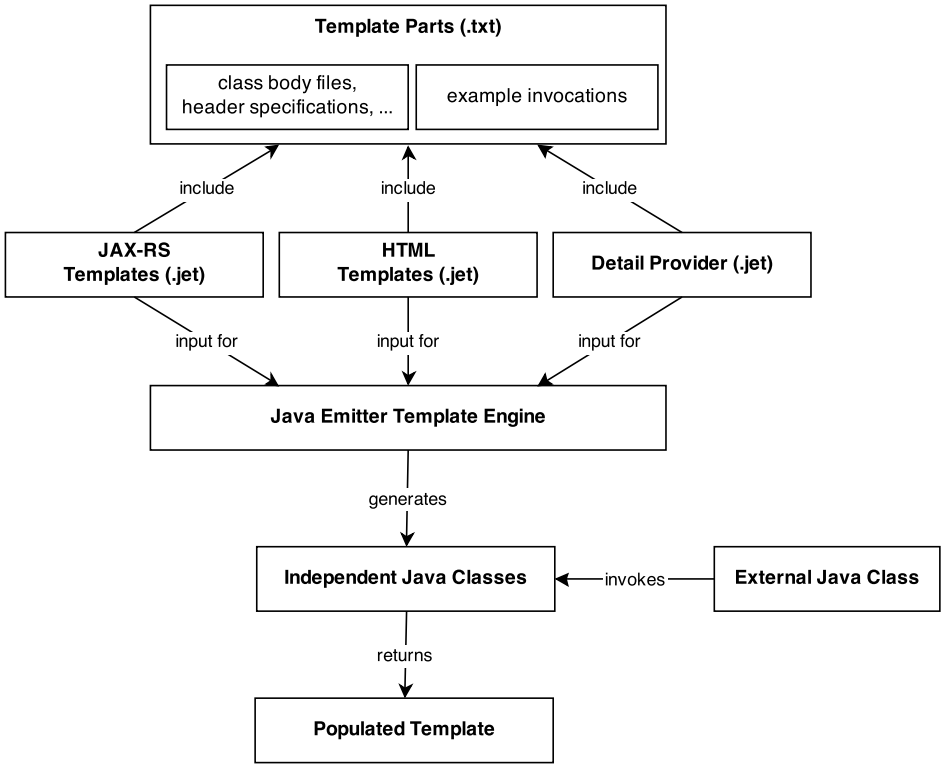

Figure 7.3.: Structure Of The Code Generator Component.

7.4. Resource Property Add-On

The Resource Property Addon builds a completely separate project. It uses the TabbedPropertView extension of the automatically generated Resource Diagram project and adds a new tab to the property view. The view's Standard Widget Toolkit (SWT) Pane has a simple child element - a Java browser object. This browser is used to display data from the Code Generator.

Every focus of an element within the Resource Diagram triggers an event in the Java class which is related to the property view. The add-on initially parses the Gen Model of the currently edited resource diagram file. As soon as a supported element (listed in 5.2) is focused it searches for the related Domain Model object within the Gen Model and uses both of them to invoke the Code Generator. The generator is enhanced by a new JET file (Figure 7.3) for a `Detail Provider`. The resulting Java class provides, if queried with correct parameters, HTML content with combined information about the interaction and the Domain Model object. The previously mentioned example invocations are included into the Detail Provider's JET file as well to provide example requests in case their interaction consumption types are supported.

Instead of writing the Detail Providers content into a file it is directly retrieved as a String and set as the content for the browser object in the view pane.

Using this method new resource element types can be supported easily by simply adding new content for the Detail Provider's JET file. The Java browser object uses the default Eclipse browser to render its data. This browser can be changed within the Eclipse properties to other installed browsers.

7.5. HTML Generation

Due to the fact that the URLs to a resource are not part of the HTML documentation no Deployment Model information is needed to create it. By altering the structure of the HTML Doc Model and its creation process several changes in the Code Generator were inevitable. Now the Code Generator creates not only a single file containing all the resources, it is split up into different parts. The main file contains a list of linked resources which provides to navigate to a single resource. Every HTML resource content is now stored its own file inside a subdirectory. Additionally necessary JavaScript content and libraries as well as the Cascading Style Sheets (CSS) are stored in separate files. To provide invocation examples within the HTML documentation the necessary template parts are added to the main JET file.

7.6. Version Management

Since the version management of auto-generated content is not simple to handle, the prototype had all files under version control. Versioning every file of this system causes a lot of traffic and especially a lot of required disk space on the server side. Committing altered files selectively results in a lot of detail work.

Since most of the Java Classes for the models and their editors can be created out of the EMF file all projects solely containing auto-generated code are deleted (e.g. domain.edit, domain.editor, domain.diagram). Additionally all auto-generated code within a heterogeneous project (auto-generated and manually written code) is deleted as well. Thus in the different model projects only remain three files. The EMF file (initial file), the Ecore file and the Gen Model file. This Gen Model file has no relation to the mentioned Gen Model, it exists in every project. All remaining projects with manually written code are properly under version control. Consequently the repository size (amount of files) has been reduced significantly and every model, editor or diagram can be created within seconds by using EuGENia with the model files.

7.7. Eclipse Epsilon Evaluation

While developing with Eclipse Epsilon and all its connected technologies (Emfatic, EuGENia, ...) several pros and cons came up that either supported the implementation or required additional effort. In the following some facts about using the Epsilon are compared to scenarios not using any of these technologies.

Pros

Easy Syntax And Sematics The creation of a EMF file is pretty simple due to the Java related syntax and logic. Several EMF classes with attributes and references can be used to build a meta model definition using concepts of object oriented programming (e.g. inheritance).

Single File Simple editors only need the single EMF file to become generated. In case the editor has to be altered modifications have only to be made in the definition file instead of many files.

Quick Results As soon as meta model definition exists an editor (graphical or textual) can be created by EuGENia within seconds. Necessary views and projects are created automatically during that process. Thus, compared to a manual implementation, there almost no effort to get an executable editor.

Solid Model No assumptions or concepts are needed for the structure for an instance of a meta model. The engine automatically generates an XML based model which is easy to parse and use by ETL transformations.

Cons

Missing Flexibility The amount of supported EMF constructs is high, but still not everything is realizable with them (e.g. single model editable by multiple editors). Using a manual way of implementing an editor provides a lot more flexibility. A single editor could be used to create an absolute RESTModling model which combines all the models covered by this thesis. A single editor may simply provide multiple views and different tools to alter it.

Enhancements Integration The integration of manually developed enhancements can be complex and work-intensive due to the automatically overriding when regenerating an editor.

Code Deletion Of course the use of EuGENia results in the already mentioned disadvantage of code deletion (MDSD) as well.

Transformation Effort Since there are many models for the different editors a lot of effort goes into the creation of their transformations. Changing one model (e.g. Resource Model) results in an inevitable change of four ETL transformations (Gen Model-To-Resource, Resourece-To-Deployment, Resource-To-HTML, Deployment-To-JAX-RS).

Under Development Many unsolved issues were discovered while modeling meta models, especially when using GMF annotations.

In summary, it can be stated that even though a lot of effort goes into the transformation processes of the models, the realization of this exploratory work would have consumed much more time using common procedures, even though the customization opportunities would have been much better.

8. Conclusion

This thesis faces the conceptional issues when modeling and generating a REST web service with the existing RESTModeling approach from [Sch13]. The issues can be grouped into two major parts. The first one addresses the modeling process itself by using the different existent layers of models and their graphical editor. The second part is the code outcome of the system which covers an HTML documentation and several JAX-RS stubs that can be used as a basis for a web service. From each of these parts certain conclusions can be drawn that are stated in this chapter.

The added elements and classes to the graphical domain and resource editors provide a better and more accurate way of modeling them. By using the `Association` link type and the newly introduced cardinalities within a Domain Model only a certain set of combinations result in a list in the resource model after the transformation. Due to separate HTTP methods and the possibility to map Domain Model methods to specific resource objects a better and especially more accurate Resource Model can be designed.

The usage of Emfatic and EuGENia does not allow multiple editors or views for a single model whereby it is impossible to store the information in one model that can be edited in different views using that technology. Splitting up the model into separate models each addressing a different aspect of the modeling process is a reasonable approach using Emfatic. By visually relating and integrating them into each other the information contained in a single model is increased. The visual integration of the Domain Model information within Resource Modeling editor and the simplified Deployment Model editing reduces or even removes the effort to look information up in other models.
The integration of model information within other models allows that certain views (Domain, Resource, Deployment) can be modeled by different technical groups since no detailed knowledge about the related model is necessary.

An HTML documentation to a Resource Model provides a much more detailed view on resources and their related Domain Model elements than before the modification. Related or linked resources can be found more easily and sample requests are outlined for every listed interaction. Further it is not necessary to populate the Deployment Model to get a proper documentation.

Due to the new structure of the Deployment Model's meta model a resource can be addressed via multiple links. Thereby modeling more complex web services is now possible in the first place. Different client types requesting the service can each have their individualized links to a resource (e.g. accessing the `OrderList` of the show case via `/customerorders` as a `Bar`-client or `/tasks` as a `Kitchen`-client).

One of the major aspects about this thesis is the improvement of the REST compliance of the outcome. The integration of automatically generated links enhances the REST compliance of the generated web service by fulfilling the third and last level of the Richardson Maturity Model [Fow10]. For this purpose no special user input is required since the transformation processes and the code generation take care of the HATEOAS provision. This yields in the initially desired condition, obtaining a higher level of REST compliance with less or even no additional effort.

The prerequirement is that every model which is necessary to generate the web service is entirely and properly filled. Using an incomplete or malformed model may result in a web service not fulfilling desired REST compliance. Hence a correct semantic and syntactic validation would additionally ensure the quality of the outcome by defining strict rules and supporting the user during the modeling process.

9. Perspectives

The enhancements and structural changes that were made within this thesis lead to the improved REST compliance of the outcome, the HTML documentation and the usability of the different model editors (graphical and textual). Nevertheless some additions can be implemented that improve the state of the modeling process and the generated code. This chapter lists some potential suggestions for the RESTModeling system.

Semantic Dependencies And Workflows

The current system does not support any semantic dependencies neither between elements of the domain nor of the Resource Model. In case a special workflow has to be followed it is not possible to have a semantically influenced link generation for the web service navigation. An example is depicted in Figure 9.1 for the used show case from chapter 4. If the workflow would be applied to the link generation of the web service the requesting client can be navigated by following semantic dependencies. In the current state it is possible to give up an order that can immediately be paid. It does not contain any beverages but there would be a “valid” link to a receipt (Figure 5.13) which is semantically sloppy. A possible approach to integrate this would be an enhanced or specially annotated Resource Model. A related issue which does not use any MDSD but gives a first thought to workflow related REST services is described by Jim Webber, Savas Parastatidis and Ian Robinson in “How to GET a Cup of Coffee” [WPR].

Long Running Request Realization

Sometimes HTTP POST or DELETE requests may take more time than a client is able to wait. To take care of those types of request some additional features are required. In case of a long running operation (LRO) a separate task resource may be added in the Resource Model for every tagged method from the Domain Model. This task resource is created and returned to the client as soon as a LRO is invoked. The temporary resource provides some sort of identification about the progress of the request.

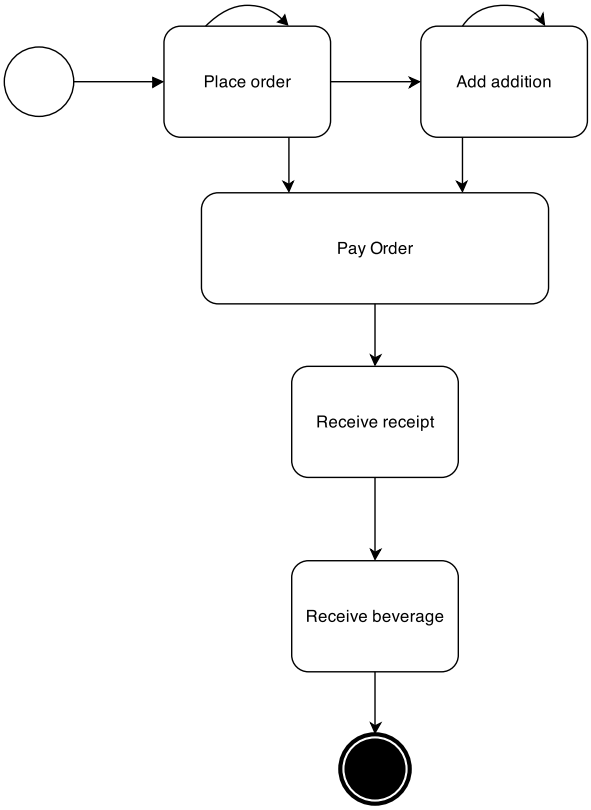

Figure 9.1.: Workflow Coffe Shop Show Case

Improved HTML Documentation

The HTML documentation provides an alphabetically ascending sorted list of resources and their information (metadata, request properties, linked elements, ..). An order with breadth-first search applied to the Resource Model may result in a nicer resource structure discovery. Another additional feature is a graphical representation of every single resource interconnected with each other in a graph within the HTML documentation. The model for this graph would be the Resource Model itself.

Enhanced Model Validation

The current validation which is done by using the ETL (Eclipse Transformation Language) is not covered by this thesis at all. There are already validations for every platform independent and platform specific model. These validations are kept very simple and they only state basic constraints like empty names and correct link targeting. Besides the simple property validation the rules may be extended to validate structural parts (unreachable resources) or even correct relations to other models.
Since the validations have to be triggered manually a validation handler could be integrated which could be located between the transformation of a model into another model. In case the transformation fails it would highlight non-compliant elements, labels or constructs that violate the constraints. To provide the easy usability, the validation could be automatically triggered if a user wants to transform from one model to another.

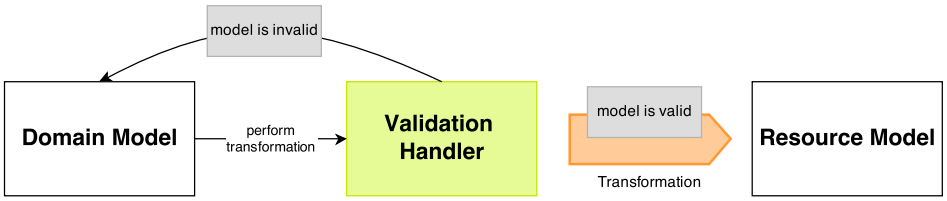

Figure 9.2.: Validation Handler Between Domain And Resource Model

A. Appendix

A.1. Eclipse Epsilon Listings

Listing A.1 shows an example for a EMF meta model. A graphical GMF editor can be created out of this source file.

Listing A.2 is the resulting model code in XML of the modeled diagram in 6.3.

Listing A.3 outlines an example transformation for a source to a target element using Eclipse Transformation Language (ETL).

Listing A.4 uses Eclipse Validation Language to validate instances of the given model in Listing A.1.

Listing A.5 shows the template for the HTML documentation index page. This code will be transformed into Java code regarding the given annotations.

Listing A.1 Example EMF Source File For A Graphical Model Editor.

```
@gmf
package examplemodeling;

@gmf.diagram(
 model_extension="examplemodel",
 diagram_extension="exampldiagram")
class ExampleDiagram extends DocumentedElement {
  val Object[*] objects;
  val Connection[*] connections;
}

abstract class DocumentedElement {
  attr String name;
}

@gmf.node(label="name")
class Object extends DocumentedElement {
  @gmf.compartment(style="list")
  val Attribute[*] attributes;
}

enum AttributeType {
  String;
  Integer;
  Float;
  Character;
  Boolean;
}

@gmf.node(label="name, type", label.pattern="{1} {0}")
class Attribute extends DocumentedElement {
  attr AttributeType type;
}

@gmf.link(
      source="source",
      target="target",
      source.decoration="none",
      target.decoration="arrow",
      tool.name="Connection")
class Connection extends Object {
      ref Object source;
      ref Object target;
}
```

Listing A.2 Model XML Code Of Figure 6.3 (Model).

```
<?xml version="1.0" encoding="UTF-8"?>
<example:ExampleDiagram xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:example="http://a.b.c/x/y/Z">
 <objects name="University">
  <attributes name="name"/>
  <attributes name="ID" type="Integer"/>
 </objects>
 <objects name="LectureA">
  <attributes name="topic"/>
 </objects>
 <objects name="LectureB">
  <attributes name="topic"/>
 </objects>
 <connections source="//@objects.0" target="//@objects.1"/>
 <connections source="//@objects.0" target="//@objects.2"/>
 <connections source="//@objects.1" target="//@objects.2"/>
</example:ExampleDiagram>
```

Listing A.3 Example ETL Transformation Script.

```
rule SourceElemenToTargetElement
 transform source : SourceModel!SoureElement
 to      target : TargetModel!TargetElement
 {
  guard : source.transformable == true

  ("Transforming: " + source.name).println();
  target.name = source.name;
  target.type = source.target;

  if(source.addComment) {
   addComment(target);
  }

  ("Transformed: " + source.name).println();
 }

operation addComment(element : TargetModel!TargetElement) {
 element.comment = "Auto generated element.";
}
```

Listing A.4 Example EVL Script To Validate Instances Of Model A.1.

```
context Object {
  constraint hasID {
    check: self.attributes.exists(a|a.‘type‘==Integer)
    message: "Object should have an identifier attribute"
  }

  constraint hasName {
    check: self.name <> ""
    message: "Object should have a name"
  }
}

context Connection {
  constraint NoSelfLink {
    check: self.target <> self.source
    message: "Object can’t reference to itself"
  }
}
```

Listing A.5 Documentation Index Page Built With A Java Emitter Template (JET).

```
<%@ jet
package="com.rmt.generator.templates.htmldoc"
class="IndexPage"
imports="
 com.rmt.generator.util.WebHelper

      com.rmt.models.psm.htmldoc.HtmlDocumentation
      com.rmt.models.psm.htmldoc.ResourceElement
      "
%>
<%
      HtmlDocumentation doc = (HtmlDocumentation)argument;
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
      <title>Documentation: <%=doc.getProjectName()%></title>
      <meta http-equiv="content-type" content="text/html; charset=UTF-8">
      <link rel="stylesheet" media="screen" href="css/htmlDoc.css">
 ...
</head>
<body>
 <h1><%=doc.getProjectName()%> HTML Documentation</h1>
 <p class="info-box">
  This documentation ...
 </p>

 <ul class="navi-list">
      <%    WebHelper.sortResourceObjects(doc.getResource().getElements());
            for (ResourceElement element : doc.getResource().getElements()) {
            if (element.isGenerateDocumentation()) {%>
            <a
                 href="content/<%=WebHelper.normalizeResourceName(element.getResource().getName())%>.html">
                  <li>
                        <%=element.getResource().getName()%>
                  </li>
            </a>
            <% } %>
      <% } %>
 </ul>
</body>
</html>
```

A.2. Ecore Meta Model Graphs

The following graphs can be retrieved from an Ecore meta model (e.g. Domain Meta Model) by using Eclipse EuGENia. They demonstrate the structure of a meta model graphically.

Figure A.1 depicts a graph of the prototype's Domain Model meta model.

Figure A.2 depicts a graph of the prototype's Resource Model meta model.

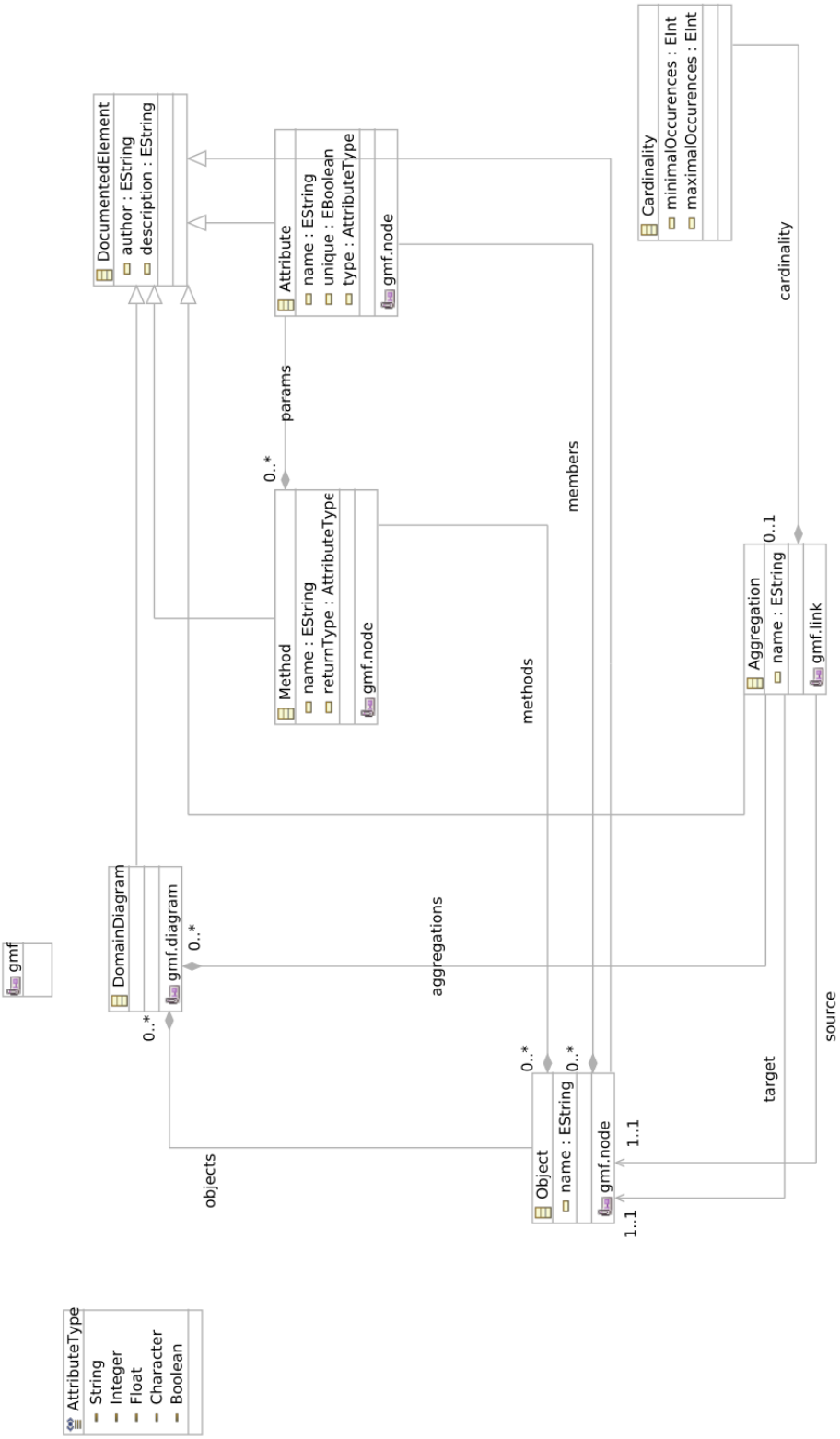

Figure A.1.: Diagram Graph Of The Meta Domain Model

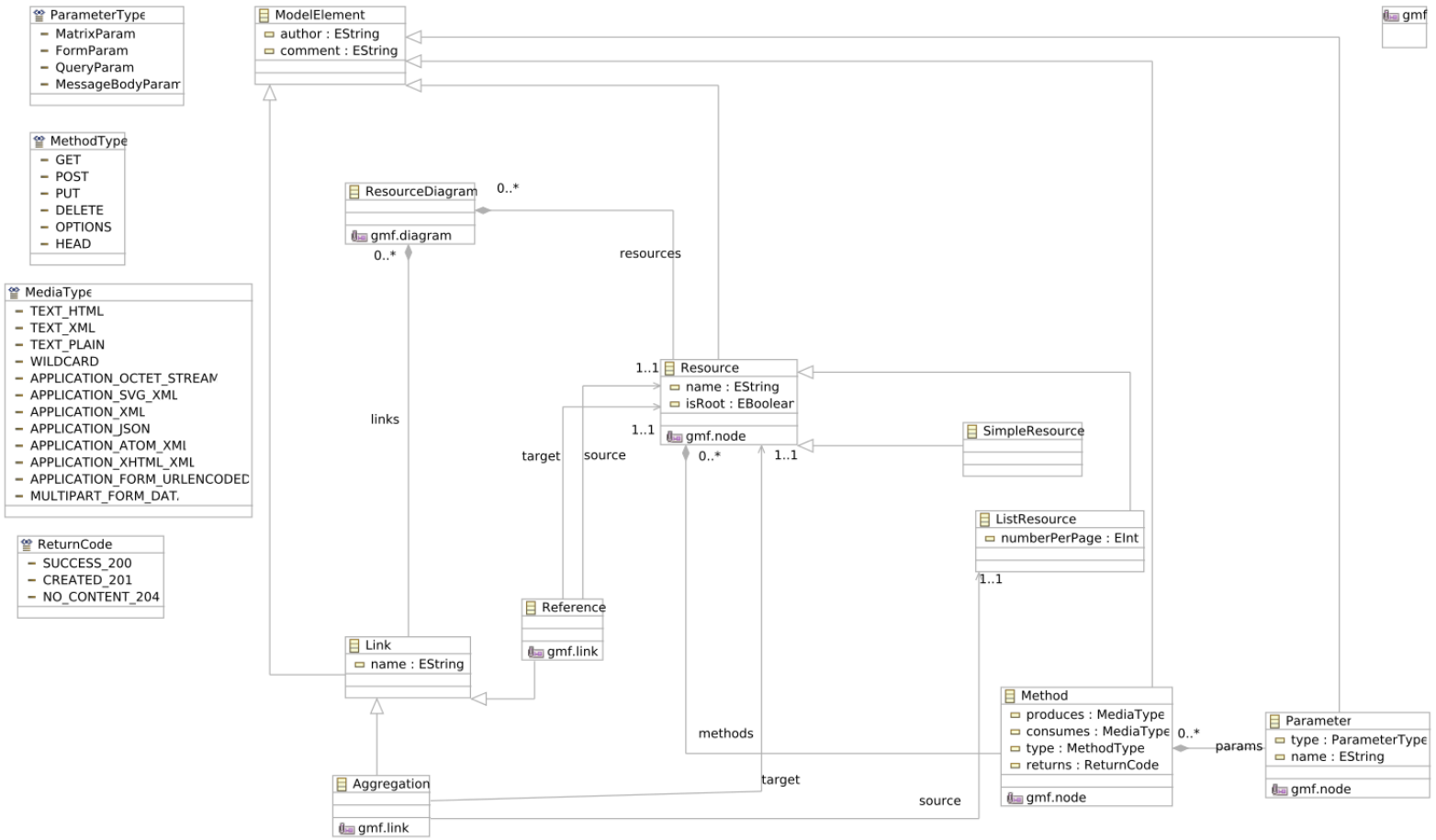

Figure A.2.: Diagram Graph Of The Meta Resource Model

Bibliography

[DH09] J. Den Haan. 8 reasons why Model-Driven Development is dangerous, 2009. Http://www.theenterprisearchitect.eu/blog/2009/06/25/8-reasons-why-model-driven-development-is-dangerous/. (Cited on page 28)

[DuV10] A. DuVander. REST vs. SOAP: Simplicity wins again, 2010. Http://www.programmableweb.com/news/new-job-requirement-experience-building-restful-apis/2010/06/09. (Cited on pages 9 and 12)

[Eps14] Epsilon. Epsilon Documentation, 2014. Version: 2014. (Cited on page 69)

[FIa99] R. Fielding, U. Irvine, et al. Hypertext Transfer Protocol - HTTP 1.1, 1999. Http://www.w3.org/Protocols/rfc2616/rfc2616.html. (Cited on pages 20, 41, 54 and 55)

[Fie00] R. Fielding. Dissertation Chapter 5: Representational State Transfer (REST). Internet, 2000. (Cited on pages 11 and 18)

[Fow10] M. Fowler. Richardson Maturity Model, 2010. Http://martinfowler.com/articles/richardsonMaturityModel.html. (Cited on page 84)

[Gie12] R. Giesler. MySQL Select Syntax, 2012. Http://dev.mysql.com/doc/refman/5.0/en/. (Cited on page 60)

[Jau14] S. Jauker. 10 Best Practices for Better RESTful API, 2014. Http://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/. (Cited on page 20)

[Mei05] O. Meimberg. Möglichkeiten und Potentiale der Formalisierung in der Softwareentwicklung. In *Schriften zum Software-Qualitätsmanagement*. Logos Verlag Berlin, 2005. (Cited on page 23)

[MSGR09] N. Moebius, K. Stenzel, H. Grandy, W. Reif. SecureMDD - A Model-Driven Development Method for Secure Smart Card Applications. In *International Conference on Availability, Reliability and Security*. IEEE, 2009. (Cited on page 26)

[PB12] A. Parada, L. Brisolara. A model driven approach for Android applications development, 2012. (Cited on page 25)

[Sch13] B. Schroth. *Entwurf und Realisierung von REST-Anwendungen nach den Prinzipien der modellgetriebenen Softwareentwicklung*. Master's thesis, University of Stuttgart, 2013. (Cited on pages 13, 31, 45 and 83)

[SV06] T. Stahl, M. Völter. *Model Driven Software Development*. WILEY, 2006. (Cited on page 27)

[SVEH07] T. Stahl, M. Völter, S. Efftinge, A. Haase. *Modellgetrieben Softwareentwicklung*. dpunkt.verlag GmbH, 2007. (Cited on page 23)

[Til11] S. Tilkov. *REST und HTTP*. dpunkt.verlag GmbH, 2011. (Cited on page 18)

[Vit10] T. Vitvar. API Anti-Patterns: How to Avoid Common REST Mistakes, 2010. Http://www.programmableweb.com/news/api-anti-patterns-how-to-avoid-common-rest-mistakes/2010/08/13. (Cited on page 11)

[VSK06] M. Voelter, C. Salzmann, M. Kircher. Model Driven Software Development in the Context of Embedded Component Infrastructures. In *Component-Based Software Development for Embedded Systems*. Springer, 2006. (Cited on page 26)

[WPR] J. Webber, S. Parastatidis, I. Robinson. How to GET a CUP of Coffee. 2008. Http://www.infoq.com/articles/webber-rest-workflow. (Cited on page 85)

[WPR10] J. Webber, S. Parastatidis, I. Robinson. *REST in Practice: Hypermadia and Systems Architecture*. O'Reilly, 2010. (Cited on pages 11 and 18)

All links were last followed on June 10, 2014.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature