Model Driven Software Engineering for Web Applications

Andreas Kraus

Dissertation

zur Erlangung des akademischen Grades
des Doktors der Naturwissenschaften
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

vorgelegt am 23.04.2007

Tag der Einreichung: 23.04.2007

Tag des Rigorosums: 04.07.2007

Berichterstatter

Prof. Dr. Rolf Hennicker

(Ludwig-Maximilians-Universität, München)

Prof. Dr. Antonio Vallecillo

(Universidad de Malaga, Spanien)

Summary

Model driven software engineering (MDSE) is becoming a widely accepted approach for developing complex applications and it is on its way to be one of the most promising paradigms in software engineering. MDSE advocates the use of models as the key artifacts in all phases of the development process, from analysis to design, implementation and testing. The most promising approach to model driven engineering is the Model Driven Architecture (MDA) defined by the Object Management Group (OMG). Applications are modeled at a platform independent level and are transformed to (possibly several) platform specific implementations. Model driven Web engineering (MDWE) is the application of model driven engineering to the domain of Web application development where it might be particularly helpful because of the continuous evolution of Web technologies and platforms.

However, most current approaches for MDWE provide only a partial application of the MDA pattern. Further, metamodels and transformations are not always made explicit and metamodels are often too general or do not contain sufficient information for the automatic code generation. Thus, the main goal of this work is the complete application of the MDA pattern to the Web application domain from analysis to the generated implementation, with transformations playing an important role at every stage of the development process. Explicit metamodels are defined for the platform independent analysis and design and for the platform specific implementation of dynamic Web applications. Explicit transformations allow the automatic generation of executable code for a broad range of technologies. For pursuing this goal, the following approach was chosen.

A metamodel is defined for the platform independent analysis and for the design of the content, navigation, process and presentation concerns of Web applications as a conservative extension of the UML (Unified Modeling Language) metamodel, together with a corresponding UML profile as notation. OCL constraints ensure the well-formedness of models and are checked by transformations. Transformations implement the systematic evolution of analysis and design models. A generic platform for Web applications built on an open-source Web platform and a generic runtime environment is proposed that represents a family of platforms supporting the combination of a broad range of technologies. The transformation to the platform specific models for this generic platform is decomposed along the concerns of Web applications to cope in a fine-grained way with technology changes. For each of the concerns a metamodel for the corresponding technology is defined together with the corresponding transformations from the platform independent design models. The resulting models are serialized to code by means of serialization transformations.

Zusammenfassung

Die modellgetriebene Softwareentwicklung (MDSE) entwickelt sich zu einem der vielversprechendsten Paradigmen für die Entwicklung komplexer Anwendungen. Modelle spielen dabei die zentrale Rolle in allen Phasen des Entwicklungsprozesses, von Analyse und Entwurf bis zur Implementierung. Die Model Driven Architecture (MDA) ist der derzeit erfolgversprechendste Ansatz zur modellgetriebenen Softwareentwicklung. Anwendungen werden dabei auf einer plattformunabhängigen Ebene modelliert und durch Transformationen in eine plattformspezifische Implementierung überführt. Die modellgetriebene Web-Anwendungsentwicklung (MDWE) wendet das Prinzip der modellgetriebenen Softwareentwicklung auf den Bereich der Web-Anwendungen an, wo sich dieser Ansatz als besonders nützlich erweist, gegeben durch die andauernde Weiterentwicklung von Web-Technologien und –Plattformen.

Die meisten aktuellen MDWE-Ansätze setzen den MDA-Ansatz allerdings nur teilweise um. Ferner werden die verwendeten Metamodelle und Transformationen oft nicht explizit definiert, und die Metamodelle sind oft zu allgemein oder enthalten nicht ausreichend Informationen zur automatischen Code-Generierung. Daher ist das Hauptziel dieser Dissertation die umfassende Übertragung des MDA-Ansatzes auf den Bereich der Web-Anwendungsentwicklung, von der Analyse bis zur Implementierung, wobei Transformationen eine entscheidende Rolle in jeder Phase des Entwicklungsprozesses spielen. Explizite Metamodelle werden definiert für die Analyse, den plattformunabhängigen Entwurf und die plattformspezifische Implementierung. Eindeutig definierte Transformationen ermöglichen die automatische Code-Generierung für ein Vielzahl von Web-Technologien. Um dieses Ziel zu erreichen wurde der folgende Ansatz gewählt.

Für die Analyse und für den plattformunabhängigen Entwurf der Inhalts-, Navigations-, Prozess- und Präsentationsebenen einer Web-Anwendung wird ein Metamodell als eine konservative Erweiterung des UML-Metamodells (Unified Modeling Language) definiert. Ein entsprechendes UML-Profil dient dabei als Notation. OCL-Constraints, die durch Transformationen überprüft werden, stellen die Wohlgeformtheit der Modelle sicher. Transformationen implementieren auch die systematische Entwicklung der Analyse- und Entwurfsmodelle. Eine generische Plattform ermöglicht eine Aufspaltung der Transformation plattformunabhängiger Modelle in einzelne Transformationen für die verschiedenen Ebenen einer Web-Anwendung. Für jede Ebene wird dazu ein Metamodell für die entsprechende Implementierungstechnologie und eine entsprechende Transformation definiert, wodurch eine Vielzahl von Technologien kombiniert werden kann. Die resultierenden Modelle werden dann durch Serialisierungstransformationen in Code umgewandelt.

Acknowledgments

First of all, I would like to thank my two supervisors Rolf Hennicker and Antonio Vallecillo, and Martin Wirsing. Then, thanks go to my colleagues, especially Nora Koch and Alexander Knapp for the fruitful discussions on Web engineering topics, and to Matthias Ludwig and Stephan Janisch for the time working together in the GLOWA-Danube project. Finally, I thank all of my students and all the people from abroad I have been working together over the last years.

My special thanks go to Sabine, our families and friends for their support and patience.

This work has been supported by the German BMBF-project GLOWA-Danube.

CONTENT

1 Introduction 13

1.1 Problem Statement 14

1.2 Approach 14

1.3 Introduction to the DANUBIA Case Study 19

1.4 Organization of the Work 20

2 Model Driven Software Engineering 21

2.1 Model Driven Architecture (MDA) 23

2.1.1 Model Types 24

2.1.1.1 Computation Independent Models (CIM) 25

2.1.1.2 Platform Independent Models (PIM) 25

2.1.1.3 Platform Specific Models (PSM) 25

2.1.1.4 Platform Models (PM) 26

2.1.2 Transformation Types 26

2.1.2.1 Model Type Transformations 26

2.1.2.2 Model Instance Transformations 27

2.2 Object Management Group Meta Architecture 27

2.2.1 Metamodel Layering 28

2.2.2 Meta Object Facility (MOF) 29

2.2.3 Unified Modeling Language (UML) 30

2.2.4 UML Extensions 31

2.3 Transformation Approaches 32

2.3.1 Classification 32

2.3.1.1 Hard-Coded Transformations 33

2.3.1.2 Model-To-Code Approaches 33

2.3.1.3 Direct-Manipulation Approaches 34

2.3.1.4 Relational Approaches 35

2.3.1.5 Graph-Transformation-Based Approaches 36

2.3.1.6 Structure-Driven Approaches 37

2.3.1.7 Hybrid Approaches 37

2.3.1.8 Other Model-To-Model Approaches 37

2.3.1.9 Discussion 38
2.3.2 Query/Views/Transformations (QVT) 38
2.3.2.1 Declarative Rules (Relations) 40
2.3.2.2 Imperative Rules (Operational Mappings) 42
2.3.2.3 Tools 43
2.3.3 Atlas Transformation Language (ATL) 44
2.3.3.1 Modules 45
2.3.3.2 Queries 49
2.3.3.3 Refining Mode 49
2.3.3.4 Tools 49
2.3.4 Transformation Modularization 50
2.3.5 Discussion 51

3 Model Driven Web Engineering 55

3.1 Elaborationist versus Translationist Approach 55

3.2 Separation of Concerns 58

3.3 Transformation Environment 60

3.4 Related Work 62
3.4.1 UML-based Web Engineering (UWE) 62
3.4.1.1 ArgoUWE 63
3.4.1.2 UWEXML 64
3.4.1.3 Transformation Techniques and Model Driven Process 66
3.4.2 WebSA 67
3.4.3 MIDAS 68
3.4.4 WebML 68
3.4.5 OOWS 69
3.4.6 HyperDE 70
3.4.7 Moreno et al. 71
3.4.8 Muller et al. 71
3.4.9 W2000 72

4 Platform Independent Analysis and Design 73

4.1 General Techniques 76
4.1.1 Checking Well-Formedness of Models 76
4.1.2 Transformation Traces 78
4.1.3 Expression Language 82

4.2 Requirements 84
4.2.1 Metamodel 85

4.2.2 Analysis Content: Example 88
4.2.3 Web Use Cases: Example 89

4.3 Content 90
4.3.1 Metamodel 91
4.3.2 Transformation Requirements2Content 92
4.3.3 Manual Refinement 94

4.4 Navigation 96
4.4.1 Metamodel 98
4.4.2 Navigation Space 103
4.4.2.1 Transformation RequirementsAndContent2Navigation 103
4.4.2.2 Manual Refinement 107
4.4.3 Addition of Indices 108
4.4.3.1 Transformation AddIndices 108
4.4.3.2 Manual Refinement 110
4.4.4 Addition of Menus 111
4.4.4.1 Transformation AddMenus 111
4.4.4.2 Manual Refinement 114

4.5 Process 114
4.5.1 Process Integration 116
4.5.1.1 Metamodel 116
4.5.1.2 Tranformation ProcessIntegration 118
4.5.1.3 Manual Refinement 120
4.5.2 Process Data and Flow 120
4.5.2.1 Metamodel 121
4.5.2.2 Transformation CreateProcessDataAndFlow 126
4.5.2.3 Manual Refinement 134

4.6 Presentation 139
4.6.1 Metamodel 140
4.6.2 Transformation NavigationAndProcess2Presentation 145
4.6.3 Manual Refinement 150

4.7 Transition to the Platform Specific Implementation 152

5 Platform Specific Implementation 153

5.1 Generic Platform 154
5.1.1 Spring Framework 158
5.1.2 Runtime Environment 163
5.1.3 Configuration 167

5.1.3.1 XML Metamodel 170
5.1.3.2 Transformation Rules 171
5.1.3.3 Serialization to Code 176

5.2 Content via JavaBeans 176
5.2.1 Java Metamodel 178
5.2.2 Example 180
5.2.3 Transformation Content2JavaBeans 180
5.2.4 Serialization to Code 185

5.3 Content via RMI 186
5.3.1 Example 187
5.3.2 Transformation Content2RMIInterfaces 187

5.4 Navigation 190
5.4.1 Example 191
5.4.2 Transformation Navigation2Conf 192

5.5 Process 193
5.5.1 Process Runtime Environment: The Web Process Engine 194
5.5.2 Example 199
5.5.3 Transformation Process2Conf 200

5.6 Presentation 202
5.6.1 JSP Metamodel 203
5.6.2 Example 204
5.6.3 Transformation Presentation2JSP 205
5.6.4 Serialization to Code 208

6 Case Study 211

6.1 Platform Independent Analysis and Design 211
6.1.1 Requirements 211
6.1.1.1 Analysis Content 211
6.1.1.2 Web Use Cases 213
6.1.2 Content 215
6.1.2.1 Results of Transformation Requirements2Content 216
6.1.2.2 Manual Refinement 217
6.1.3 Navigation 219
6.1.3.1 Navigation Space 219
6.1.3.2 Addition of Indices 222
6.1.3.3 Addition of Menus 226
6.1.4 Process 228

6.1.4.1 Process Integration 228
6.1.4.2 Process Data and Flow 231
6.1.5 Presentation 240
6.1.5.1 Results of Transformation NavigationAndProcess2Presentation 240
6.1.5.2 Manual Refinement 244
6.2 Platform Specific Implementation 247
6.2.1 Content 247
6.2.1.1 Results of Transformation Content2JavaBeans 250
6.2.1.2 Manual Refinement 252
6.2.2 Navigation 252
6.2.2.1 Results of Transformation Navigation2Conf 253
6.2.2.2 Manual Refinement 255
6.2.3 Process 255
6.2.3.1 Results of Transformation Process2Conf 262
6.2.3.2 Manual Refinement 264
6.2.4 Presentation 264
6.2.4.1 Results of Transformation Presentation2JSP 265
6.2.4.2 Manual Refinement 267
6.3 Evaluation 269
7 Conclusion 271
7.1 Results 271
7.2 Limitations 272
7.3 Future Research 274
8 Table of Figures 277
9 References 283
A UML Profile 301
A.1 Tabular Overview 302
A.2 Trace 303
A.3 Requirements 304
A.4 Navigation 304
A.5 Process 305
A.6 Presentation 306
B ATL Transformations 309

B.1 Transformation Environment Setup 309

B.2 Metamodels 310

B.2.1 UWE Metamodel 311

B.2.1.1 KM3 Metamodel 311

B.2.1.2 Constraint Checking Query 315

B.2.2 Java Metamodel 324

B.2.2.1 KM3 Metamodel 324

B.2.2.2 Constraint Checking Query 326

B.2.2.3 Serialization Query 327

B.2.3 XML Metamodel 330

B.2.3.1 KM3 Metamodel 330

B.2.3.2 Constraint Checking Query 331

B.2.3.3 Serialization Query 331

B.2.4 JSP Metamodel 332

B.2.4.1 KM3 Metamodel 332

B.2.4.2 Constraint Checking Query 333

B.2.4.3 Serialization Query 333

B.3 PIM2PIM Transformations 334

B.3.1 Refinement Header 334

B.3.2 Trace Header 335

B.3.3 Transformation Requirements2Content 336

B.3.4 Transformation RequirementsAndContent2Navigation 337

B.3.5 Transformation AddIndices 343

B.3.6 Transformation AddMenus 344

B.3.7 Transformation ProcessIntegration 347

B.3.8 Transformation CreateProcessDataAndFlow 350

B.3.9 Transformation NavigationAndProcess2Presentation 360

B.4 PIM2PSM Transformations 369

B.4.1 Configuration Header 370

B.4.2 Transformation Content2JavaBeans 373

B.4.3 Transformation Content2RMIInterfaces 379

B.4.4 Transformation Navigation2Conf 384

B.4.5 Transformation Process2Conf 385

B.4.6 Transformation Presentation2JSP 390

1 INTRODUCTION

Recently, model driven software engineering (MDSE) is becoming a widely accepted approach for developing complex applications and it is on its way to be one of the most promising paradigms in software engineering. MDSE advocates the use of models as the key artifacts in all phases of the development process, from system specification and analysis to design and testing. Models are even replacing code as low-level artifacts. Developers are forced to focus on the problem space (models) and not on the (platform specific) solution space. Thus, the basic functionality of a system can be separated from its final implementation. Additionally, tool support for model driven engineering has continuously improved over the last years, from CASE tools with hard coded metamodels and hard coded code generation facilities to tools with flexible and/or extensible metamodels and model transformation facilities. The most promising approach to model driven engineering is the Model Driven Architecture (MDA) defined by the Object Management Group (OMG) [Miller03]. Applications are modeled at a platform independent level and are transformed by means of model transformations to (possibly several) platform specific implementations.

Web engineering is a relatively new direction of Software Engineering with focus on the development of Web-based systems [Kappel03a]. Several approaches for the development of Web applications have been proposed in the last years. Model driven Web engineering (MDWE) is the application of model driven engineering to the domain of Web application development where it might be particularly helpful because of the continuous evolution of Web technologies and platforms. Different concerns of Web applications are captured by using separate models, e.g. for the content, navigation, process and presentation concern. These models are then transformed to code, whereas code comprises web pages, configuration data for Web frameworks as well as traditional program code.

1.1 Problem Statement

Today, many MDWE approaches, such as W2000 [Baresi05], MIDAS [Cáceres04] or UWE [Koch06b], claim to be MDA compliant. However, most of them provide only a partial application of the MDA pattern, for example by not providing platform specific models. Further, almost each approach uses specific modeling elements for analysis and design with special elements for representing typical concepts of Web applications such as, for instance, navigation nodes and links, but only some of them define an explicit metamodel which is an essential prerequisite for applying model driven techniques. Additionally, these metamodels are often too general or do not contain sufficient information for the automatic code generation.

Additionally, although data-intensive Web applications are now handled well by most current approaches, there is still insufficient support for dynamic Web applications, i.e. Web applications supporting the execution of complex workflows, i.e. Web processes.

Further, many current approaches are not based on standards for metamodeling, notation and transformation, which complicates tool interoperability, reusability and extensibility. Often a proprietary graphical notation is used for the representation of the modeling elements, and proprietary tools are used for analysis, design and code generation. Model-to-model and model-to-code transformations are in many cases hard-coded and not made explicit.

Thus, the main goal of this work is the complete application of the MDA pattern to the Web application domain from the top to the bottom, i.e. from analysis to the generated implementation. Transformations play an important role at every stage of the development process. An explicit metamodel and a corresponding notation based on the UML standard is defined for the platform independent analysis and design of dynamic Web applications. On the other hand, metamodels representing technologies are defined for the platform specific implementation. Transformations support the systematic construction of platform independent models and allow the automatic generation of executable code for a broad range of technologies, based on a generic platform for dynamic Web applications.

1.2 Approach

An overview of the model driven development process of this approach is depicted in Figure 1. The main phases of the process are analysis, design and model-driven implemen-

tation. This corresponds to the computation independent models (CIM), the platform independent models (PIM) and the platform specific models (PSM) of the MDA approach. The aim of the analysis phase is to gather a stable set of requirements. The functional requirements are captured by means of specialized use cases and the content requirements are captured by a class model. The design phase consists of constructing a series of models for the content, navigation, process and presentation concerns at a platform independent level. Transformations implement the systematic construction of dependent models by generating default models which then can be manually refined by the designer. Information that is not available at a higher abstraction level has to be added by the developer, e.g. by introducing inheritance or adding additional features to modeling elements. The stereotypes «transformation» combined with «refinement» indicate that the transition from the requirements model to the design models consists of automatic transformations and manual refinement by the designer, whereas the transformation to the platform specific implementation model is carried out fully automatically, with exception of fine grained behavior as detailed below. Finally, the platform specific implementation model is serialized to code by model-to-code transformations.

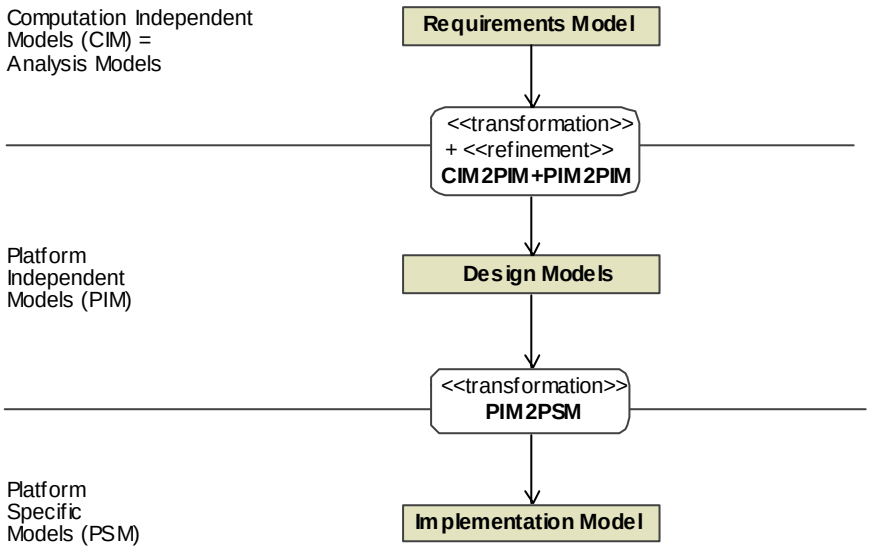


Figure 1. Development process overview

A major contribution is the definition of an explicit metamodel for the analysis and design of Web applications. A metamodel is a precise definition of the modeling elements and their relationships for a specific domain. The well-formedness of models is defined by constraints, specified in the Object Constraint Language (OCL), which are attached to the metamodel. Each transformation checks first the validity of the constraints for the respec-

tive input models. A first version of this metamodel was presented in [Kraus03a] and [Kraus03b] as a conservative extension of the UML 1.4 metamodel. Conservative means that the modeling elements of the UML metamodel are not modified. The metamodel for the platform independent analysis and design presented in this work is a refinement of this first version. It is adapted to the changes of UML 2 [OMG05a] and enhanced with constructs allowing the application of model transformations to support on the one hand the systematic design and on the other hand the transformation to platform specific models. The metamodel is structured along the concerns of Web applications which are addressed in this work: requirements, content, navigation, process and presentation. Actually, the metamodel is limited to modeling elements which are supported by the transformations presented in this approach. However, the metamodel and the transformations are designed to be easily extensible for adding further modeling constructs or whole new modeling aspects in the future. An additional trace model is used for handling incremental updates of the analysis and design models. A UML profile provides a notation for the metamodel, making use of all benefits and tools that support UML.

The complex workflow of a Web application is represented by a process model. This approach focuses on the modeling and transformation of “coarse grained” behavior. Thus, a process model expresses the composition of “fine grained” behavior by means of UML activities. The semantics of activities is based on control and data token flows, similar to Petri nets [Priese03]. Fine grained behavior is represented by UML operations which correspond to services. Thus, an operation call corresponds to a service call. This concept for the representation of the behavior of Web applications fits in the Service Oriented Architecture (SOA) approach [Dostal05] because the basic idea of the SOA approach is to see the realization of a business process as a composition of services. The service itself is assumed to be already predefined and implemented, thus the modeling and implementation of services themselves is not part of this approach.

Following the vision of MDA, the implementation platform is represented by a corresponding metamodel, and the platform independent design models are mapped by a transformation to the platform specific implementation model. A generic platform for Web applications is proposed which is built on the open-source Spring framework and includes a generic runtime environment that allows the execution of complex workflows. The Spring framework offers a high degree of flexibility for the combination of different technologies. It relies on the Model/View/Controller (MVC) pattern, where the concerns of a Web application correspond to the model (content), view (presentation) and controller (navigation and process) roles in the MVC pattern. This allows for a corresponding decomposition of the transformation to the platform specific models as depicted in Figure 2. For a concrete model technology (e.g. JavaBeans) or view technology (e.g. Java Server Pages) corre-

sponding metamodels and transformations have to be defined. A generic runtime environment plugged into the Spring framework takes the controller part of a Web application implementation. This controller has to be configured for a specific Web application by configuration data generated from the navigation and the process models. An abstraction technique for the communication between the model, view and controller parts allows to decouple the corresponding technologies and transformations. This is represented as inheritance relationships for the model and view technologies in Figure 2. In a final step, the platform specific implementation models are serialized to code.

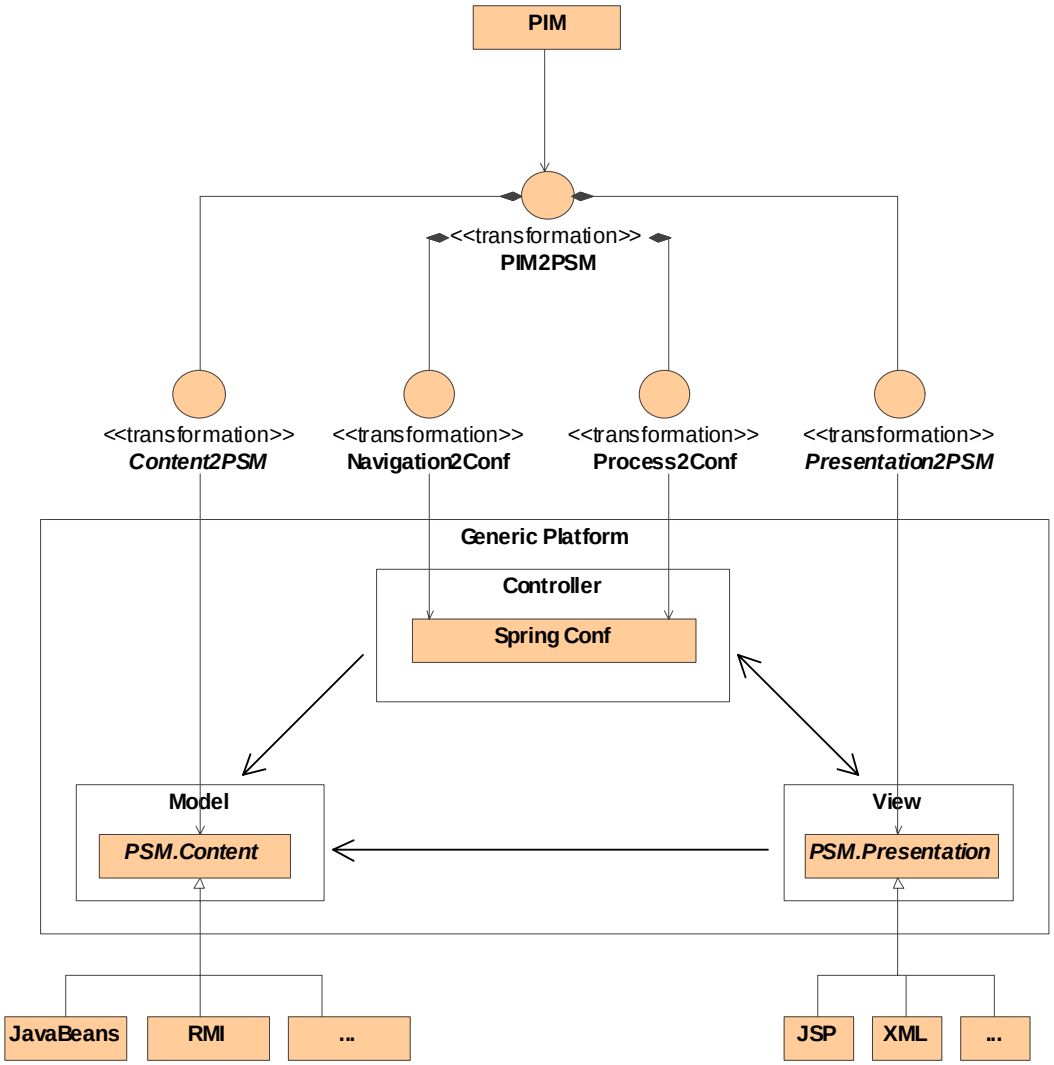


Figure 2. Platform specific implementation using a generic platform

As transformations are vital for the success of an MDA approach, this work comprises an evaluation of currently available transformation languages. The favored choices within the MDA meta architecture are the Atlas Transformation Language (ATL) [ATL06a] and QVT [OMG05b] which both originate from the Request For Proposal (RFP) for MOF 2.0 Query/Views/Transformations by the Object Management Group (OMG) [OMG02]. Both are hybrid transformation languages which combine declarative expressiveness and imperative constructs for those parts of a transformation which would be too cumbersome or even impossible to express with declarative constructs. The QVT standard is still in the finalization phase and sufficient tool support is not yet available. On the other hand, the tool support for ATL has already reached a stable state that is satisfactory for application to real world model driven engineering challenges, therefore ATL is used in this work for all kinds of transformations.

The context of this work is the UML-based Web Engineering approach (UWE), which is continuously evolved by the Web Engineering group of the Munich University LMU [UWE]. The contribution of this work to the further evolution of UWE is essentially the realization and elaboration of a transformational approach for the model driven development of dynamic Web applications supporting the fundamental principles of UWE. This comprises the:

- Addition of a process concern for supporting complex workflows
- Definition of a metamodel and a corresponding UML profile for the platform independent analysis and design
- Definition of transformations that implement the systematic evolution of the platform independent models
- Decomposition of the transformation to the platform specific models
- Development of a generic Web platform that supports the combination of a broad range of technologies, including a runtime environment that allows the execution of complex workflows
- Definition of platform specific metamodels and the corresponding transformations to generate the implementation of Web applications

An important guideline for this work is that no proprietary tools are used. Thus, there are no restrictions on the employed modeling tool as long as it supports UML 2 profiles and stores models in the standardized model interchange format. Further, the platform specific

part of this approach can be reused by other Web design approaches if their corresponding metamodels are made explicit and the transformations to the platform specific models are adapted accordingly. The metamodel for the platform independent part of this approach can be understood as a common metamodel for Web application analysis and design, which can be extended by the special features of other Web design approaches.

1.3 Introduction to the DANUBIA Case Study

The author is involved in the project GLOWA-Danube [GLOWA-Danube], and the Web user interface to be developed for the environmental simulation system with the name DANUBIA is used as a running example in this work. The GLOWA-Danube project is part of the GLOWA initiative, which has been established by the German Ministry of Education and Research, to address the various consequences of global change on regional water resources in a variety of catchments with different natural and cultural characteristics.

This work is not about DANUBIA, thus only the parts relevant for the Web user interface are used in the examples. The inner structure of DANUBIA, called "core system", is described in [Ludwig07]. The user interface can be structured into:

- Project management, which serves to organize environmental simulations. A project represents a set of simulation runs for a common objective. Simulation runs may be additionally grouped into scenarios for representing specific assumptions.

- Component management, which serves to administrate simulation components and their metadata.

- Global data management, which is used for the management of data that is shared by all simulation projects, such as for example geographical data about the simulation area.

- Result data management, for the administration and processing of the results of simulation runs.

The project manager part serves as the case study in this work. For more detailed information on GLOWA-Danube and DANUBIA the reader is referred to [Ludwig02].

1.4 Organization of the Work

This work is organized as follows: Chapter 2 gives an introduction to model driven software engineering (MDSE) in general with focus on the Model Driven Architecture (MDA) and the meta architecture of the Object Management Group (OMG). Most important, approaches for model transformations are discussed and classified. In Chapter 3 the basic constituent parts of the approach of this work are presented. This comprises the application of the separation of concerns principle for metamodel and transformation decomposition and a presentation of the transformation environment. Further, an overview of the state of the art for model driven Web engineering is given. Chapter 4 comprises the platform independent part of the approach. For every concern of Web applications the corresponding part of the metamodel is presented together with transformation rules for the derivation of the corresponding models from other models. Chapter 5 addresses the platform specific part of the approach for a proposed generic platform. Thus the specific metamodels for the target platform are presented together with the corresponding transformation rules from the platform independent models. In Chapter 6 the results of the previous chapters are applied to the DANUBIA case study. Finally, Chapter 7 concludes this work with a discussion of the results, the limitations of the approach and remarks about proposed future research topics.

2 MODEL DRIVEN SOFTWARE ENGINEERING

Recently, model driven engineering (MDE) has attained considerable attention and it is on its way to be one of the most promising paradigms in software engineering. A good introduction to the general background of MDE is given in [Bézivin05]. MDE refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. Models are considered as first class entities, even replacing code as primary artifacts. Developers are forced to focus on the problem space (models) and not on the (platform specific) solution space. The complexity of platforms is handled better by model driven approaches than by third-generation programming languages [Schmidt06]. Also the tool support for model driven engineering has continually improved over the last years, from UML CASE tools with a hard coded metamodel and hard coded code generation facilities to tools with flexible and/or extendible metamodels and model transformation facilities.

The most promising approach to model driven engineering is the Model Driven Architecture (MDA) defined by the Object Management Group (OMG) [Miller03]. Applications are modeled at a platform independent level and are transformed by means of model transformations to (possibly several) platform specific implementations.

Other approaches are based on domain specific languages (DSL) [Fowler04b]. A domain specific language is targeted to a particular kind of problems in contrast to general purpose languages that are supposed to be applicable to a broad range of problems. A DSL can be either external or internal. External DSLs are written in a different language than the main programming language of the application and are processed using some form of compiler or interpreter. Configuration files or Unix mini languages such as *awk* are examples for external DSLs. Internal (or embedded) DSLs are embedded in the main programming language of the application. Dynamic programming languages such as Lisp, Smalltalk or Ruby are particularly suited for internal DSLs. There are some analogies between domain specific languages and the MDA approach. A DSL corresponds to a metamodel and the counterpart of the meta-metamodel is the grammar for specifying a DSL. In most cases a DSL can be represented as a MOF metamodel or even as a UML profile (cf. 2.2), thus DSL

approaches can be integrated or combined with MDA approaches. This is particularly useful because some Web frameworks use small domain specific languages, e.g. for configuration purposes. The most prominent example for an approach based on domains specific languages, also called language workbenches [Fowler05a], is the Software Factories approach propagated by Microsoft.

A Software Factory is defined as a configuration of languages (i.e. DSLs), patterns, frameworks and tools that can be used to rapidly produce a set of unique variants of an archetypical product [Greenfield04]. For a so-called software product line first a common software architecture is designed and a framework developed that supports this architecture. The construction of a new system then consists of assembling and configuring software components provided by the framework. Domain specific languages are used to specify the particular properties of this new system and they are used for the automatic generation of glue code and configuration data.

Each of the two approaches, MDA and Software Factories, has its advantages and disadvantages. Debates about which approach is better suited for model driven engineering are still going on, see for example [Bast04]. The important result is that neither of the two approaches is a clear winner. One misunderstanding often encountered by people defending Software Factories is that an MDA approach has to exclusively use the general purpose modeling language UML. This is not true, as arbitrary (MOF) metamodels can be used including small and problem tailored metamodels corresponding to small domain specific languages. One important difference between these approaches relevant for this work is the higher abstraction level of MDA which consists of a clear differentiation between the platform independent problem space and the platform specific solution space, whereas DSLs in the Software Factories approach often intermingle the platform specific and the platform independent aspect. Additionally, MDA focuses on using standards such as UML, MOF and XMI which results in better tool interoperability. The fact that mature MDA tools are already available in contrast to tools for Software Factories is an important argument for MDA, although this may change in the future as Software Factories are strongly promoted by Microsoft. On the other hand an important ingredient of the Software Factories approach is the development of frameworks for a product line while the MDA approach does not give a direction about how to actually generate code from the platform independent models. For a deeper comparison of these two approaches see [Muñoz05].

Although the approach of this work is mainly based on the MDA approach, some important features of the Software Factories approach are adopted for the platform specific part of this approach. It is assumed that the platform for a Web application generated by this approach comprises a Web framework tailored for the transformation of the platform spe-

cific models to code. Such a Web framework itself is assembled from different stable Web and non-Web frameworks such as for example the Apache Tomcat Web framework and the Spring framework. These frameworks are customized by a particular configuration and by adding additional elements for the specific approach. The configuration for the resulting Web framework is generated from platform specific models representing the configuration. More details are presented in Chapter 5.

The following sections start with an introduction to the Model Driven Architecture. Then the meta architecture of the OMG and some relevant standards, such as the Unified Modeling Language (UML) are discussed. Subsequently, transformation approaches, which are vital for the success of the MDA, are presented and classified.

2.1 Model Driven Architecture (MDA)

The Model Driven Architecture (MDA) is a specialized approach for model driven engineering. It evolved from the former Object Management Architecture (OMA), which provided a framework for distributed systems [Soley97]. The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns [Miller03]. Therefore, a system should be specified independently from the platform that supports it. Based on platform specifications and the choice of a specific platform the system specification should be transformed into the specific platform. The so called MDA pattern is depicted in Figure 3: a platform independent model (PIM) is transformed to a platform specific model (PSM) by means of a transformation that may get some additional input as illustrated by the empty box on the upper right of the figure. In general, transformations can be between any type of models, e.g. PIM to PIM, and also from models to code. When PIMs are based on a virtual machine, a so called abstract platform [Almeida04], not only the PIMs have to be transformed to the specific platform but also the virtual machine.

The MDA approach is further based on a set of other OMG standards such as the Meta Object Facility (MOF) and the Unified Modeling Language (UML) which are presented in 2.2. Even though the MDA approach is general in respect to metamodels the OMG propagates the use of UML as modeling language and UML profiles, see 2.2.3 and 2.2.4.

Although transformations are a key factor for the success of MDA, the approach is not based on a specific transformation language. Transformation approaches for MDA are presented and discussed in 2.3, with particular emphasis on transformation approaches that resulted from the Request for Proposal (RFP) for MOF 2.0 Query/Views/Transformations [OMG02] which was issued because of the need for a future standardized transformation

language. The transformation language QVT is currently in the finalization phase of becoming the future standard for transformations in the scope of MDA, but stable tool support is not yet available, see 2.3.2. For this work the Atlas Transformation Language (ATL) is used, see 2.3.3. Although the ATL proposal did not succeed in getting accepted as future QVT standard it fulfills large parts of the requirements from the RFP and has a technological lead over QVT implementations as the first running implementation was already available in mid 2005. Further, ATL transformations could be mapped to QVT transformation, thus the result of this work could be migrated to QVT when a stable tool support is available, see also 2.3.5.

In the following sections some general concepts of the MDA such as model types and transformation types are presented.

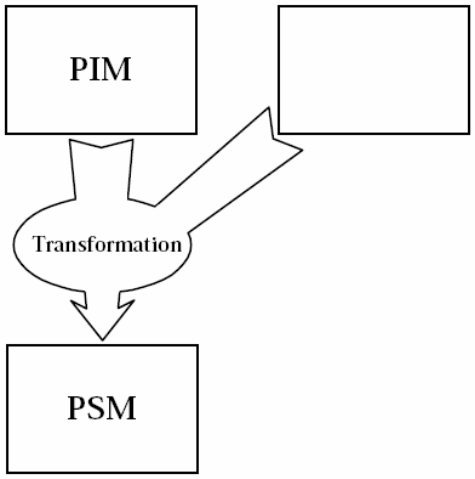


Figure 3. MDA Pattern, from [Miller03]

2.1.1 Model Types

In the context of the MDA the following model types are distinguished: computation independent model (CIM), platform independent model (PIM), platform specific model (PSM) and platform model (PM). The former three models represent views of the system from different viewpoints and abstraction levels corresponding to the analysis, design and implementation views in conventional (non-MDA) software engineering.

2.1.1.1 Computation Independent Models (CIM)

The MDA Guide states that a computation independent model is a view of a system from the "computation independent viewpoint" [Miller03]. The intended meaning is less on abstracting from computation but on details about the structure of a system. Other synonyms for computation independent model are analysis model, domain model or business model, depending on the context of the adapted MDA approach. The CIM plays an important role in bridging the gap between those that are experts about the domain and its requirements on the one hand, and those that are experts of the design and construction of the artifacts that together satisfy the domain requirements, on the other [Miller03].

2.1.1.2 Platform Independent Models (PIM)

A platform-independent model is a model that is independent of the features of a platform of any particular type. Platform independence is a matter of degree, so that even a model for a very general type of platform may be considered platform independent. PIMs can be targeted for a technology-neutral virtual machine, a general kind of platform or abstract platform, cf. [Almeida04].

2.1.1.3 Platform Specific Models (PSM)

A platform specific model is targeted for a specific platform. It is derived from a platform independent model by a transformation, thereby combining the platform independent specification with platform specific details. Depending on its purpose, a PSM can provide more or less detail. If it comprises all the details needed for automatically generating an implementation from the model then it represents a platform specific model of the implementation. The resulting code is then obtained by serializing the model. On the other hand, a PSM may require further automatic or manual refinement before obtaining a platform specific implementation model. In this work, PSMs represent implementation models. In Figure 4 (left) the platform specific implementation model for a Java class *Project* with a field *title* of type *String* is depicted using the notation of UML object diagrams. This model corresponds directly to Java code depicted in Figure 4 (right) which is derived by serializing the Java model.

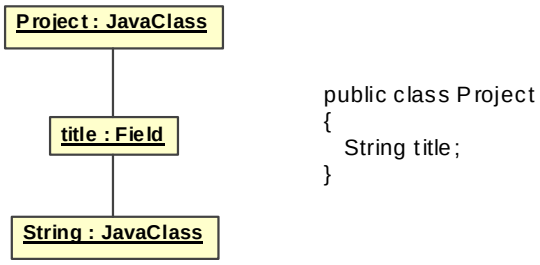


Figure 4. Example for a platform specific model and the corresponding code

2.1.1.4 Platform Models (PM)

The concept of a platform model in the MDA guide [Miller03] is ambiguous. On the one hand a platform model provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform, i.e. a platform model represents a model of a platform in a general platform metamodel. On the other hand it also provides, for use in a platform specific model, concepts representing the different kinds of elements available for a platform, i.e. a platform model provides a metamodel for the platform specific model. In [Wagelaar05] the concept of a platform model based on description logics is presented. It can be used to automatically select and configure a number of reusable model transformations for a concrete platform.

2.1.2 Transformation Types

The MDA guide [Miller03] distinguished two different types of transformations, model type transformations and model instance transformations.

2.1.2.1 Model Type Transformations

Model type transformations map instances from a source metamodel (defining the source types) to instances of a target metamodel (defining the target types). Figure 5 illustrates the relationships between models, metamodels and transformations. An important aspect is that transformations themselves are models, i.e. instances of a transformation metamodel. This allows for higher order transformations, i.e. transformations that generate transformations, c.f. 2.2.4. All metamodels share the same meta-metamodel (MOF), see 2.2.1. Note that transformations can also be multi directional, but nevertheless most implementations will only support unidirectional transformations.

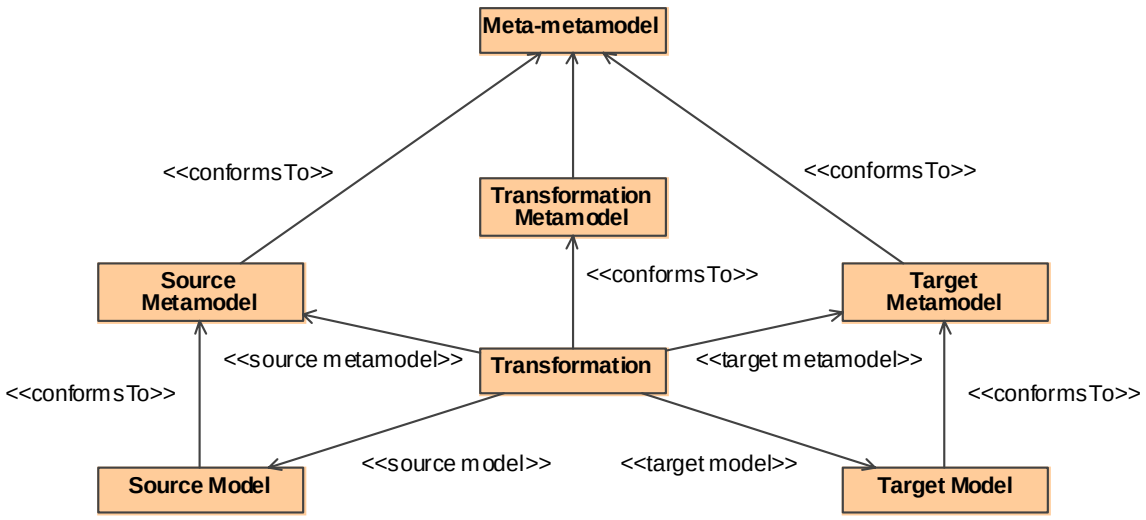


Figure 5. Pattern for model type transformations

2.1.2.2 Model Instance Transformations

A model instance transformation is a special kind of model type transformation where additional marks are used for selecting model elements from the source model. Thus, these marks drive the transformation. Marking can be done directly in the source model if the source metamodel supports an appropriate concept for marking, e.g. UML stereotypes and tagged values can be used as marks. Alternatively, a separate marking model can be used that assigns marks to model elements from the source model. Another more abstract kind of mark is the selection of patterns in the source model. Often, a set of marks is specific for a particular platform (often defined as UML profile) and marking the platform independent source model corresponds to the selection of the mapping to a specific platform dependent concept, e.g. the stereotype *«EJB»* triggers a transformation to create platform specific EJB modeling elements.

2.2 Object Management Group Meta Architecture

MDA is not a standalone standard but rather built on a set of different OMG standards. The first subsection explains the underlying metamodel hierarchy for the MDA approach. Then follows a description of the meta-metamodel which is the root of the metamodel hierarchy. Afterwards a short introduction to the Unified Modeling Language (UML) is given together with its extension facilities because the metamodel used in this work for platform independent modeling will be an extended UML metamodel.

2.2.1 Metamodel Layering

When dealing with models one has to distinguish between the concepts metamodel, model and model instance. A metamodel defines a language for specifying models and a model is an instance of a metamodel. Thus the "instance of" relationship spans a metamodeling hierarchy and because a metamodel is a model itself this hierarchy may be of infinite depth. The term "model" is used mostly in context of a specific layer M_i in the metamodeling hierarchy with its metamodel residing at layer M_{i+1}.

The OMG defines a four-layer metamodel hierarchy as foundation of its standards [OMG05a]. The root of the metamodeling hierarchy at layer M3 is the meta-metamodel. This meta-metamodel is called Meta Object Facility (MOF) and defines the language for specifying metamodels (see 2.2.2. and [OMG06a]). MOF is reflective, i.e. it can be used to define itself, thus there is no need for additional meta-layers above MOF. UML and the OMG common Warehouse Metamodel (CWM) are examples for metamodels at layer M2, i.e. languages for specifying models at layer M1. The metamodel hierarchy bottoms out at M0, containing the run-time instances of model elements defined at M1.

Summarized, the four layers of the OMG metamodeling hierarchy are:

M3: meta-metamodel = metamodel language specification = MOF

M2: metamodel = model language specification

M1: model

M0: model instance

An example of how these metamodel layers are related to each other is given in Figure 6 for UML "Class" and "Instance" model elements. A peculiarity of the UML is that at M1 constrained versions of "runtime" instances at M0 can be modeled, so called snapshots, by using instances of the metaclass *InstanceSpecification*. The implications of this so called "loose metamodeling problem", which is still present in UML 2, are discussed in [Atkinson01].

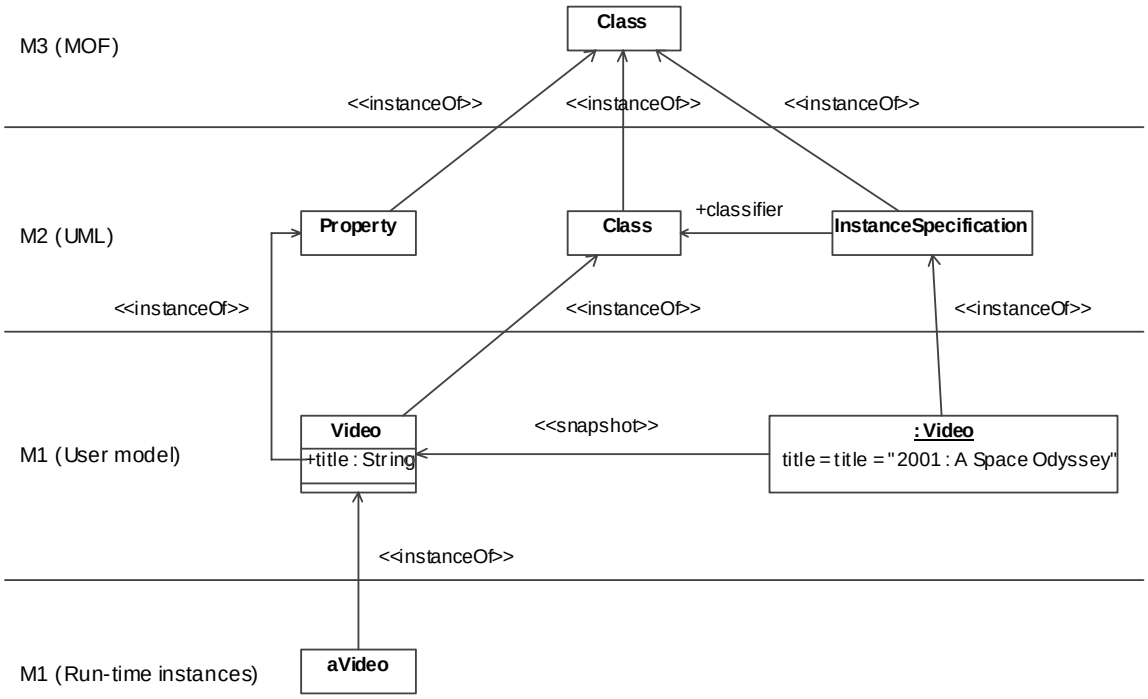


Figure 6. Metamodeling hierarchy example (adapted from [OMG05a])

2.2.2 Meta Object Facility (MOF)

The Meta Object Facility (MOF) serves as the metadata management foundation for MDA [OMG06a]. It emerged from the need for a standard for data exchange between applications, especially tools. Metadata, i.e. data about data, is essential for the specification of structure and meaning of data. MOF provides a standard for specifying metamodels, i.e. a meta-metamodel, which is the root of a metamodeling hierarchy, as presented in the previous section. MOF is reflective meaning that MOF itself is defined with MOF.

MOF mappings are important for metadata-driven interchange and metadata manipulation and therefore vital for the success of the MDA approach. For tool interoperability, especially UML tools, the mapping to the XML Metadata Interchange (XMI) format is of essential importance. The details about the mapping to XML Document Type Definitions (DTD) as well as to XML Schema definitions are described in [OMG05c]. Other mappings are available, for example to the Java Metadata Interface (JMI) [JMI], which allows for direct manipulation of models from within the Java programming language, see also 2.3.1.3.

The common modeling concepts of MOF Version 2 as well as UML Version 2 have been refactored to the common UML Version 2 Infrastructure library, which is reused in both

specifications [OMG05a], thus MOF modeling corresponds to a subset of UML class modeling and even its graphical notation can be reused for MOF modeling.

The MOF specification is split up into two packages, Essential MOF (EMOF) and Complete MOF (CMOF), corresponding to different conformance levels of tool interoperability. EMOF comprises the kernel metamodeling capabilities of MOF and closely corresponds to the facilities found in object oriented programming languages. The primary goal of EMOF is to allow simple metamodels to be defined with simple concepts while CMOF metamodels can be expressed as EMOF metamodels using EMOF extension mechanisms (similar to using UML profiles).

2.2.3 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a widely recognized general purpose language for modeling software systems as well as non-software systems. It is extendable through its profile mechanism for customization of the modeling language, see also the next section. The abstract syntax of the UML is specified as a MOF metamodel and for the concrete syntax a graphical notation is defined [OMG05a]. UML is suitable for modeling the static as well the dynamic aspects of a system. A UML model is usually edited with a UML tool by creating UML diagrams, which in turn are views of a UML model.

The first major version of the UML standard was developed in the late 90ties by the “three amigos” Grady Booch, Ivar Jacobson and James Rumbaugh, who joined efforts to unify earlier approaches for object-oriented modeling languages. The second major version UML 2 is the result of an initiative of the OMG, which issued three Requests for Proposals (RFP) for the three parts of the specification of the actual version of the UML. The final specifications were published in 2006. The biggest changes between UML 1 and UML 2 relevant for this work are the improved extension mechanisms by UML profiles (see also the next section) and the improved support for activity modeling. The former is needed for mapping the platform independent metamodel for Web applications to a UML profile, and the latter is needed for modeling Web processes.

The specification of UML 2 is structured in three parts. The infrastructure specification [OMG05a] defines the core static concepts of the UML, such as classes and associations. It is reused for the specification of the meta-metamodel MOF (cf. 2.2.2) as well as for the superstructure specification [OMG05a], which adds concepts for enhanced features of the UML, such as use cases or activities. Another part of the specification deals with the Object Constraint Language (OCL) [OMG06b]. The superstructure specification is organized in language units, such as for example actions, activities or classes. Additionally, each language unit is divided into different compliance levels for supporting different levels of

complexity of a language unit. A complete description of the UML is out of the scope of this work, for more details see [OMG05a] or [Hitz05].

2.2.4 UML Extensions

Extensions of the UML can be either heavyweight or lightweight. Heavyweight extensions are based an a modified UML metamodel with the implication that the original semantics of modeling elements is changed and that the externalized form is no longer compatible with UML tools. Lightweight extensions are called UML profiles and are based on the extension mechanisms of the UML[1]. A profile consists of a number of stereotypes, which in turn represent extensions of UML metaclasses. Although stereotypes themselves are specializations of classes and thus may contain attributes (formerly known as tagged values), it is not possible to have an association between two stereotypes or between a stereotype and a metaclass. The effect of new (meta)associations within profiles can be achieved in limited ways either by adding new constraints within a profile that specialize the usage of some associations of the reference metamodel, or by extending the *Dependency* metaclass with a stereotype and defining specific constraints on this stereotype [OMG05a]. Unfortunately, the capabilities of UML modeling tools have to be taken into account when defining a UML profile. For instance, it is not possible to define dependency relationships between arbitrary kinds of model elements, as for example between properties that are members of an association. Therefore, another more pragmatic solution is to use the full qualified names of model elements for referencing meta association ends.

A special case of heavyweight extensions are “profileable” extensions [Baresi02], which means that it is possible to map the metamodel to a UML profile. Then standard UML CASE-tools with support for UML profiles, i.e. stereotypes, tagged values and OCL constraints can be used to create the models for the extended metamodel. In addition, it is possible to define meta transformations, which transform back and forth from the profile definition to the heavyweight metamodel definition, see Figure 7. Also a higher order transformation can be defined that transforms from a profile to transformations (!) that transform back and forth between heavyweight metamodel instances and UML models with the corresponding profile applied. In [Abouzahra05] these transformations are realized using ATL, cf. 2.3.3.

[1] In general profiles are not restricted to UML but can be defined for any MOF metamodel

This work uses “profileable” heavyweight extensions of the UML and a UML profile as notation. This allows the definition of the particular modeling elements and transformations for Web application development at the right level of abstraction.

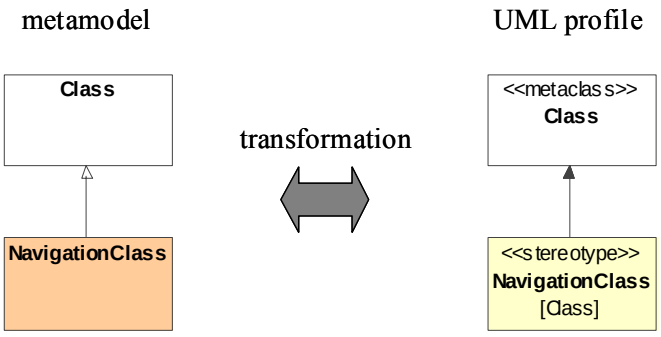


Figure 7. Transformation between metamodel and UML profile

2.3 Transformation Approaches

Transformations are vital for the success of the MDA approach. Expressed exaggeratedly, transformations lift the purpose of models from documentation to first class artifacts of the development process. In this section different transformation approaches are presented and classified. The two approaches QVT and ATL are explained in more detail in separate sections. Finally, transformation modularization and the reasons for choosing ATL as transformation language are discussed.

2.3.1 Classification

Czarnecki et al. propose a possible taxonomy for model transformation approaches [Czarnecki03]. The taxonomy is described with a feature model [Czarnecki98] for making design choices for model transformations explicit. The following sections represent a classification to fit current (and future) model transformation approaches. A specific approach may fit in several classes. Although the same classes as presented in [Czarnecki03] are used in the following, an updated selection of transformation approaches is discussed with the focus more on the overall impact of a specific transformation approach within the context of this work than on its detailed features.

2.3.1.1 Hard-Coded Transformations

In contrast to generic transformation approaches, in hard-coded transformations the metamodels and the transformations rules are implemented in a specific programming language. Thus they do not allow an easy adaptation to changes in the metamodels or transformation rules. An example for this class of transformation approaches is UWEXML, which is presented in 3.4.1.2.

2.3.1.2 Model-To-Code Approaches

Model-to-code transformations can be seen as specialized model-to-(platform specific) model transformations, i.e. a model-to-code transformation is equivalent to a transformation to the platform specific metamodel with subsequent serialization to code. Pure model-to-code approaches are therefore restricted in their application to simple PIM-to-PSM transformations.

Visitor-based approaches are based on the Visitor design pattern [Gamma95]. A visitor mechanism is provided to traverse the internal representation of a model and write code to a text stream. An example for this approach is Jamda [Jamda], an object-oriented Java framework providing a set of classes to represent UML models, an API for manipulating models, and a visitor mechanism to generate code. It does not support the MOF standard to define new metamodels, but new model element types can be introduced by subclassing predefined Java classes.

The majority of currently available MDA tools are *template-based* model-to-code approaches, e.g. AndroMDA [AndroMDA], OptimalJ [OptimalJ], XDE [XDE] and ArcStyler [ArcStyler]. AndroMDA is based on the template engines Velocity and XDoclet. A template is a piece of target code that contains meta tags to access the content of the source model and for iterative code expansion, see [Cleaveland01] for an introduction to template based approaches. In comparison to visitor-based approaches templates are more similar to the generated code.

The *openArchitectureWare* (oAW) tool platform [OAW] describes itself as “a suite of tools and components assisting with model driven software development built upon a modular MDA/MDD generator framework implemented in Java supporting arbitrary import (design) formats, meta models, and output (code) formats”. In fact the oAW platform is a powerful tool suite for straightforward transforming models (or arbitrary input data) to code. The main differences, e.g. to the more general QVT approach, are: the concept of models and metamodels only holds on the input side of a transformation. The output of a transformation is plain and unstructured text, i.e. code. Although the output can represent a

model at low level, such as XMI, and from this a model instance could be instantiated, this approach is considered more a hack than a reasonable use of the oAW framework. Transformations are expressed as transformation templates in a template language called XPand. A transformation template is defined for a specific metaclass and executed on all instances of it. Transformations can be composed and inherited. The output of a transformation template is a concatenation of literal code and properties of the model element (i.e. instance of the metaclass). One of the main practical benefits of using the oAW platform is the large number of UML tools that is supported for constructing the input model. By customizing the so called instantiators, new dialects or metamodel extensions can easily be adopted. A basic UML metamodel represented with Java classes can be used as is or be extended by custom metamodel classes either manually or by means of a metamodel generator. Also handling the UML extension mechanisms such as stereotypes and tagged values is easily accomplished. Additionally, aspects are supported as well in the metamodel as in the transformation templates. The framework integrates into the Ant deployment tool and into the Eclipse development platform.

2.3.1.3 Direct-Manipulation Approaches

Direct-manipulation approaches can access an internal model representation via an Application Programming Interface (API) for a particular programming language, such as Java. Thus, direct-manipulation approaches use hard-coded transformation rules, but they rely on standardized interfaces or frameworks for manipulating models.

The Java Metadata Interface (JMI) Specification [JMI] is the result of a Java Community Process for the implementation of a dynamic, platform-neutral infrastructure that enables the creation, storage, access, discovery, and exchange of metadata within the Java environment. JMI is based on the Meta Object Facility (MOF) specification [OMG06a], which defines a generic programming mechanism (using IDL) that allows for the discovery, query, access, and manipulation of metamodel instances, either at design time or at runtime. The semantics of any modeled system can be completely discovered and manipulated with standard Java interfaces through the modeling components defined by JMI, i.e. platform-independent discovery and access of metadata is possible. These interfaces are generated automatically from MOF models, for any kind of metamodel. Additionally, metamodel and metadata interchange via XML is enabled by JMI's use of the XML Metadata Interchange (XMI) [OMG05c] specification. Applications using JMI can use the metadata to dynamically interpret the meaning of information, take action on that information and automate transactions across disparate systems and data sources. [Jamda] for example is based on JMI.

The Novosoft Metadata Framework (NSMDF) [NSMDF] is based on the JMI specification. Generated classes conform to the JMI specification and also additional services like event notification, undo/redo support are provided. The metadata repository is an in-memory implementation. The framework API itself is generated from the MOF specification, where the shipped (binary) distribution was generated for MOF version 1.3. Libraries for handling UML 1.3 und 1.4 models in the version 1.1 of the XMI interchange format are available in binary form for the direct usage to process UML models. Unfortunately, this version of NSMDF is not integrated in the actual version of the open source CASE tool ArgoUML [ArgoUML]. Instead, it uses the former version of the framework called NSUML with support for UML 1.3 and the version 1.0 of the XMI interchange format. Due to huge discrepancies between the UML 1.3 and 1.4 metamodel an easy transformation between the different interchange formats is not trivial. The tool ArgoUWE [Knapp03] for development of Web application for the methodology UWE is based on ArgoUML, hence it is an example for a direct-manipulation approach using NSMDF, see also 3.4.1.1.

NetBeans [NetBeans] is another open source project sponsored by Sun Microsystems. It hosts the NetBeans integrated development environment (IDE) and the NetBeans platform, a software development framework. NetBeans is based on a modular architecture and one such module is the Metadata Repository (MDR). It can be used either within the development environment or separately. The MDR subproject is another JMI implementation. Metamodels have to be defined using the version 1.4 of MOF, although a transformation from MOF 1.3 models is done transparently. The metadata repository is a file based implementation. The automatization of metadata tasks is encouraged by the definition of Ant tasks. Furthermore, with the UML2MOF tool it is possible to transform UML models using the UML Profile for MOF [OMG06a] to MOF models, i.e. a regular UML CASE tool can be used to define MOF models.

2.3.1.4 Relational Approaches

Relational approaches are declarative approaches based on mathematical relations. Basically, a relation is specified by defining constraints over the source and target elements of a transformation. Without special constructs relations cannot be executed and have no direction. Relational approaches with executable semantics can be implemented with logic programming using unification-based matching, search, and backtracking, see [Gerber02]. QVT (see 2.3.2) and ATL (see 2.3.3) support the relational approach and additionally provide imperative constructs, i.e. they are hybrid approaches.

2.3.1.5 Graph-Transformation-Based Approaches

Graph-transformation-based approaches are declarative approaches based on the theoretical work on graph transformations. Typed, attributed, labeled graphs [Andries96] are particularly suitable to represent UML-like models.

When applying a graph transformation rule a left-hand-side graph pattern is matched and replaced by a right-hand-side graph pattern. In addition to the left-hand-side graph pattern matching conditions can be defined, e.g. negative conditions. Additional logic is needed to compute target attribute values. In most approaches, rule scheduling is done externally including non-deterministic selection, explicit condition, and iteration (including fixpoint iterations). Fixpoint iterations can be used for computing transitive closures. An introduction to the concept of graph transformations is given in [Heckel05].

The Attributed Graph Grammar System (AGG) is a rule based visual language supporting an algebraic approach to graph transformation [AGG]. It aims at specifying and rapid prototyping applications with complex, graph structured data. AGG may be (re)used (without GUI) as a general purpose graph transformation engine in high level Java applications using graph transformation methods. The special characteristics of AGG are [AGG]: complex data structures are modeled as graphs which may be typed by a type graph. Graphs may be attributed by Java objects and types. Basic data types as well as object classes already available in Java class libraries may be used and in addition, user-defined Java classes can be used. Graph rules may be attributed by Java expressions, which are evaluated during rule applications. Additionally, rules may have attribute conditions being boolean Java expressions. The formal semantics of rule application is given in terms of category theory, by a single categorical construction known as a pushout in an appropriate category of attributed graphs with partial morphisms. This approach is also named single-pushout approach (SPO). For more details about the formal background of AGG see [Ehrig06]. The AGG tool environment provides graphical editors for graphs and supports visual interpretation and validation. The Tiger project [Ehrig05] extends AGG by a concrete visual syntax definition for flexible means for visual model representation, including the generation of visual editors as plug-ins for the Eclipse framework.

Another promising approach based on graph transformations is VIATRA which currently serves as the underlying model transformation technology of several ongoing European projects mainly in the field of dependable systems [VIATRA]. The main objective of VIATRA is the specification of model transformations in a mathematically precise way. In addition to graph transformations, a rule-based specification formalism for abstract state machines [Börger03] is provided, which allows the use of imperative constructs in addition to the declarative constructs provided by graph transformations. Graph transformation

rules are assembled to complex model transformations by abstract state machine rules, which provide a set of common control structures with precise semantics frequently used in imperative or functional languages.

Other graph-transformation approaches for model transformation include MOFLON [Amelunxen06], UMLX [Willink03], GreAT [Agrawal03], ATOM [Lara02], PROGRES [Schürr89] and BOTL [Marschall03].

2.3.1.6 Structure-Driven Approaches

The idea behind structure driven approaches is that a transformation first creates a hierarchical structure of the target model. Then attributes and references are set in the target model. OptimalJ is an example for this approach [OptimalJ]. So called incremental copiers that have to be specialized by the user copy elements from the source to the target model. Then the target model can be changed by the transformation implementation. The mapping of specific source types is implemented by defining a corresponding Java method with the source type matching the input parameter type of the method.

2.3.1.7 Hybrid Approaches

Hybrid approaches combine declarative and imperative constructs. QVT (see 2.3.2), ATL (see 2.3.3) and VIATRA (see 2.3.1.5) are hybrid approaches. Another approach is XDE [XDE], which is based on The-Gang-Of-Four design pattern [Gamma95]. Thus, the basic concepts are parameterized collaborations or UML collaboration diagrams to model design patterns. Patterns can be associated to so called Scriptlets, which are similar to JSPs and responsible for model-to-code transformations.

2.3.1.8 Other Model-To-Model Approaches

Some transformation approaches do not fit into any of the classes presented in the previous sections, for example XSLT [XSLT], a declarative and functional transformation language for transforming XML documents. As MOF compliant models can be represented in the XML Metadata Interchange (XMI) format (cf. 2.2.2), XSLT could in principle be used for model-to-model transformations. But this approach has severe problems with scalability, thus XSLT is not suited for more complex transformations

2.3.1.9 Discussion

In the previous sections different classes of transformation approaches have been presented. This section discusses the advantages and disadvantages of each transformation class regarding the use of it within an MDA approach.

Hard-coded transformations and direct-manipulation approaches are too low-level because transformations have to be implemented directly by the user in some programming language. Model-to-code approaches could only be used for the last step in an MDA process when models have to be serialized to code. Structure-driven approaches are too specialized because they can only be used for certain kinds of correspondences between source and target elements. Relational approaches are a good compromise between flexibility and declarative expressiveness.

From the academic viewpoint approaches based on graph transformations may seem the best choice because of their high expressiveness and declarative nature. The theoretical foundation of graph transformations eases solving formal questions regarding properties of transformations. But also the complexity stemming from the non-determinism in the rule scheduling and application strategy hinders the practical applicability of this approach. For example the termination of the transformation process has to be studied carefully.

Hybrid approaches combine declarative expressiveness and imperative constructs for those parts of a transformation which would be too cumbersome or even impossible to express with declarative constructs.

2.3.2 Query/Views/Transformations (QVT)

QVT originates from the OMG Request for Proposals (RFP) for MOF 2.0 Query/Views/Transformations [OMG02]. There have been eight submissions with the two most promising being QVT-P [QVTP03] and ATL (see 2.3.3). Some of the submitters (including those of QVT-P and ATL) joined to form the QVT-Merge group. The first version of the future QVT standard is currently in the finalization phase [OMG05b]. QVT is a hybrid transformation language, i.e. it has declarative and imperative constructs. The declarative constructs stem from the original QVT-P proposal. Transformations can be unidirectional or multi-directional, the latter implies the specification of an inverse transformation for imperative transformations. A special use of transformations is to check models, i.e. without modification of the participating models. QVT also allows incremental updates.

The declarative part is based on the concept of relations and has a two-level architecture: Relations at the higher level and Core at the lower level. The Relations language allows the

declarative specification of the relationships between models. It supports complex object pattern matching and has a visual notation similar to UML object diagrams. A Relations model can be mapped to Core, the lower level declarative language of QVT for execution on an engine implementing the Core semantics. The Core language only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models. All model elements of source and target are treated symmetrically. Because of its simplicity it is easier to define semantics for the Core language, see [OMG05b]. As Relations models can be mapped to Core models, Core is mainly used for implementing the declarative part of QVT. As in ATL (see 2.3.3) there is an analogy to Java and the Java Virtual Machine: Relations corresponds to the language Java and Core to the Java Virtual Machine that actually executes the transformation or program respectively. The transformation that maps Relations models to Core models plays the role of the Java Compiler.

There are two mechanisms for invoking imperative implementations of transformations from Relations or Core: one standard language, called Operational Mappings using OCL with side effects, as well as non-standard black-box implementations of MOF operations. Further there is a one-to-one mapping between operations of imperative implementations to relations in Core or Relations, meaning that even if only the imperative part of QVT is used, there is always an implicit declarative specification that is refined by the imperative implementation.

The relationship between the different parts of QVT is depicted in Figure 8.

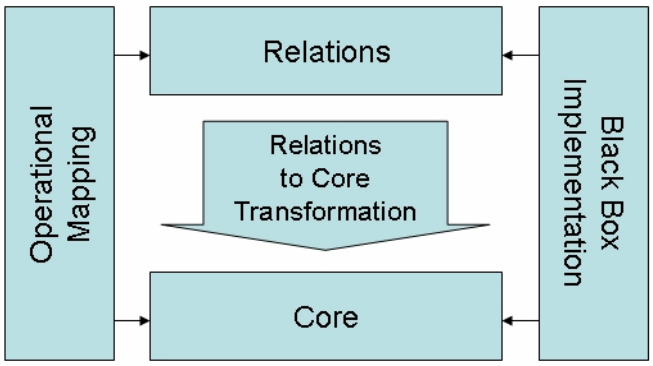


Figure 8. Relationships between QVT parts [OMG05b]

2.3.2.1 Declarative Rules (Relations)

In the Relations language a transformation is specified as a set of relations that must hold for the participating models. A basic relation definition for an example transformation that maps UML classes to Java classes looks like this:

```
relation Class2JavaClass
{
    domain uml c : Class { name = cn }
    domain java jc : JavaClass { name = cn }
}
```

A relation definition comprises a list of domain pattern definitions. A domain is a distinguished typed variable that can be matched in a model of a given type. In the example there are two domains for UML and Java models, respectively. A domain pattern can be considered as a template for objects and their properties that must be located, modified, or created in a candidate model to satisfy the relation, see [OMG05b] for the definition of syntax and semantics. In the example given above, the property *name* in both patterns is bound to the same variable implying that they should have the same value. When executing the transformation the relation above does not have any direction nor will the participating models be modified, but inconsistencies are reported. Thus, this represents a mere model checking transformation.

Domains can be marked as *checkonly* or *enforced*. Checkonly domains are merely checked if there exists a valid match that satisfies the relationship. Enforced domains are first checked and when the checking fails, the target model is modified so that the relation holds. For more details about the semantics see [OMG05b]. The direction of the transformation is from checkonly domains to enforced domains. The example with enforcement of creating the Java classes (if they don't exist yet) looks like this:

```
relation Class2JavaClass
{
    checkonly domain uml c : Class { name = cn }
    enforce domain java jc : JavaClass { name = cn }
}
```

A very important aspect for modeling transformations is how transformation rules can be composed. This allows giving a structure to transformation rules and facilitates reuse by decomposing complex transformations into many small ones. In QVT transformation rules can be composed by using either the *when* part or the *where* part of a rule. The *when* part

defines the guard condition of a rule. It can be used to express which rules have to be executed before executing a rule. The *where* part on the other hand can be used to trigger other transformations after a rule is executed. The following example uses the when part to compose two rules for mapping packages and their contained classes to Java:

```
relation Package2Package
{
    checkonly domain uml p : Package
    {
        name = pn
    }
    enforce domain java jp : Package
    {
        name = pn,
        isImported = false
    }
}

relation Class2JavaClass
{
    checkonly domain uml c : Class
    {
        name = cn,
        package = p : Package {}
    }
    enforce domain java jc : JavaClass
    {
        name = cn,
        package = jp : Package {}
    }
    when
    {
        Package2Package( p, jp );
    }
}
```

The other variant using the where part (and the first version of *Class2JavaClass*) looks like this:

```
relation Package2Package
{
    checkonly domain uml p : Package
    {
        name = pn,
```

```
        ownedType = c : Class {}
    }
    enforce domain java jp : Package
    {
        name = pn,
        isImported = false,
        classes = jc : JavaClass {}
    }
    where
    {
        Class2JavaClass( c, jc );
    }
}
```

It case of using *when* the referenced rule is triggered before, in case of *where* after the referencing rule, thus it depends on the transformation design, which solution is more appropriate.

In addition to the textual notation there is a graphical notation for the Relations language that is similar to UML object diagrams. The example relation *Class2JavaClass* in the graphical notation is depicted in Figure 9. The strength of the graphical notation is the visualization of domain patterns in a intuitive way, hence facilitating the acceptance of QVT among users. For more information about the graphical notation see [OMG05b].

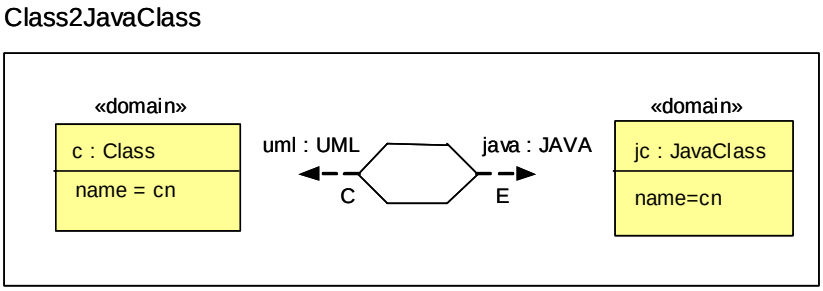


Figure 9. Graphical notation of QVT Relations

2.3.2.2 Imperative Rules (Operational Mappings)

The Operational Mappings language can either be used for a complete imperative approach or for complementing declarative relations with an imperative implementation (hybrid approach). A transformation in the Operational Mappings language comprises an entry operation called *main* and a set of mapping operations. A mapping operation is syntactically described by a signature of a source element type, a guard (a when clause), a mapping body

and a postcondition (a where clause). Even if it is not explicitly notated in the concrete syntax, a mapping operation is always a refinement of a relation, which is the owner of the when and where clauses. The method body of a mapping operation comprises imperative expressions and object expressions. Imperative expressions are a marriage between OCL expressions and typical imperative expressions as found for example in Java. Object expressions provide a high-level construct for creating and/or updating model elements. For more details about syntax and semantics of Operational Mappings see [OMG05b].

The following example shows the imperative realization of the transformation rules in the previous section. Operations are always called explicitly by using the *map* operator. The entry point of the transformation is the entry operation *main*.

```
main()
{
    uml.objectsOfType( Package )->map package2Package();
}

mapping Package::package2Package() : Package
{
    name := self.name;
    classes := self.ownedType->map class2JavaClass();
}

mapping Class::class2JavaClass() : JavaClass
{
    name := self.name;
}
```

2.3.2.3 Tools

As already stated in the introduction, the specification of QVT is currently in the finalizations phase, and therefore fully compliant tool support is not yet available, although initial efforts have been made. For example, the current version of the modeling tool Together Architect 2006 for Eclipse[2] provides a partial implementation of the imperative part of QVT, i.e. an implementation for operational mappings. An alternative could be the (possibly bi-directional) mapping from the relational part of QVT to a graph-based transformation approach such as AGG or VIATRA but, as already stated in 2.3.1.9, this would be in-

[2] Together Architect product homepage http://www.borland.com/de/products/together/index.html

sufficient without a mapping of the imperative part of QVT. The lack of tool support at the time of writing was one of the major reasons for choosing ATL over QVT, but this decision was not made without interoperability between the approaches in mind, see also 2.3.5.

2.3.3 Atlas Transformation Language (ATL)

The Atlas Transformation Language (ATL) [ATL06a] originated as proposal from the ATLAS INRIA & LINA research group[3] to the Request For Proposal (RFP) document for MOF 2.0 Query/Views/Transformations by the Object Management Group (OMG) [OMG02]. Although the ATL proposal did not succeed in getting accepted as future QVT standard the ATLAS INRIA & LINA research group joined the QVT-Merge group, which is actually finalizing the first version of the future QVT standard, see 2.3.2. Nonetheless, ATL fulfills large parts of the requirements from the RFP and has a technological lead over QVT implementations as the first running implementation was already available in mid 2005. A large and growing user community together with the active members of the research group ensures that ATL keeps evolving. Right now, it has already reached a stable state that is satisfactory for application to real world model driven engineering challenges. Some of the shortcomings of the early versions of ATL in comparison to QVT, such as missing M:N transformations and deficiencies for composing transformations, are eliminated with the version *ATL 2006*, see [Jouault06b].

The model transformation language of ATL is specified as a metamodel (abstract syntax) and as a concrete textual syntax. ATL transformations are unidirectional, operating on a number of read-only source models and producing a number of write-only target models. During the execution of a transformation source models may be navigated, but changes are not allowed. Target models cannot be navigated. ATL is a hybrid model transformation language containing a mixture of declarative and imperative constructs. The preferred declarative style allows to simply express mappings between the source and target model elements. However, when coping with problems of higher complexity, imperative constructs ease the specification of mappings that can hardly be expressed declaratively. A transformation program is composed of rules that specify how source model elements are matched and navigated to create and initialize the elements of the target models. OCL is used for matching model elements and for specifying properties of target elements. Besides basic model transformations (called modules), ATL defines an additional model querying

[3] Homepage of the ATLAS INRIA & LINA research group: http://www.sciences.univ-nantes.fr/lina/atl/atlProject/atlas/

facility that enables to specify requests onto models. ATL also allows code factorization through the definition of ATL libraries.

ATL provides both implicit and explicit scheduling. The implicit scheduling algorithm starts with calling a rule that is designated as an entry point and may call further rules. After completing this first phase, the transformation engine automatically checks for matches on the source patterns and executes the corresponding rules. Finally, it executes a designated exit point. Explicit scheduling is supported by the ability to call a rule from within the imperative block of another rule. ATL transformation descriptions are transformed to instructions for the ATL Virtual Machine, which executes the transformations. This is analogous to Java and the Java Virtual Machine. The semantics of ATL has been formalized by using abstract state machines [Ruscio06].

The current implementation of ATL still has some limitations. For example, rule organization is flat making it hard to organize large numbers of rules.

2.3.3.1 Modules

An ATL module corresponds to a model to model transformation. The structure of an ATL module comprises a header section, an optional import section, a set of helpers and a set of transformation rules. In the header section source and target models of the transformation are defined. External libraries may be included in the import section. Then a set of helpers may be defined. Helpers may be global variables or functions defined with OCL. Functions can be global or bound to the context of a metaclass. The different types of rules are described in the following subsections.

2.3.3.1.1 Matched Rules

A matched rule is the default construct for the declarative part of transformations in ATL. The definition comprises the source pattern, the target pattern, an optional imperative block and the optional definition of local variables. The source pattern consists of source pattern elements. Each source pattern element defines a local model element variable with a given type (i.e. metaclass) from a given metamodel. The set of potential matches of the rule is the cartesian product of the sets matching each source pattern element. A source pattern element matches all model elements that conform to the type of the model element variable. This set can optionally be constrained by a guard condition expressed in OCL. Each match of the rule generates a target element for each target pattern element. Again each target pattern element defines a local model element variable with a given type (i.e. metaclass) from a given metamodel. In addition, a target pattern element comprises a set of bindings that assign values expressed in OCL to meta properties of the generated target elements. Fi-

nally, imperative code in the optional imperative section is executed. For details about syntax and semantics see [ATL06a]. The syntax of a matched rule definition has the following form:

```
rule Name
{
  using  -- optional
  {
    variable : type = OCL-expression;
    ...
  }
  from s : Source-Metamodel!Source-Metaclass [( OCL-expression )] -- optional guard
  [, ...]
  to t : Target-Metamodel!Target-Metaclass
  (
    target-meta-property <- OCL-expression [, ...]
  )
  [, ...]
  do -- optional
  {
    imperative part of the rule
  }
}
```

It is important to know that (currently) each tuple of source model elements must not be matched by more than one (matched) rule. Thus, source patterns have to be designed carefully and in case of equal types matched by different rules, it must be ensured that the guards of these rules partition the source elements into disjunctive sets. This restriction ensures that the rule matching algorithm terminates, because each source model element can be matched at most once.

The following example demonstrates the use of matched rules to transform a UML model to a Java model, i.e. a model representing a Java program, which can be transformed to Java code. One rule is responsible for mapping UML classes to Java classes, while another rule is responsible for mapping UML packages to Java packages. Bindings for the *name* attributes of Java classes and packages are defined. In addition, these two rules are implicitly related by the binding expression for the *package* attribute of the matched rule *Class2JavaClass,* which references the UML package in the source model to which the UML class belongs. The resolution algorithm of ATL resolves this to the target model element generated by the matched rule *Package2Package*.

```
rule Package2Package
{
    from p : UML!Package
    to jp : JAVA!Package
    (
        name <- p.name
    )
}

rule Class2JavaClass
{
    from c : UML!Class
    to jc : JAVA!JavaClass
    (
        name <- c.name,
        package <- c.package
    )
}
```

2.3.3.1.2 Lazy Rules

A lazy rule is a declarative rule, which is explicitly called. It is used for the application from within a matched rule. Lazy rules are extensions of matched rules but are not triggered automatically on elements of the source model, but called explicitly every time when referenced in a binding. Additionally, unique lazy rules are only executed once, thus when called multiple times they always return the same result.

2.3.3.1.3 Called Rules

A called rule is an imperative rule which is explicitly called. It is similar to a procedure with parameters in traditional programming. Called rules are typically used for dealing with global variables or generating output elements independent from a matching source pattern. Instead of a source pattern a parameter list has to be defined. They can only be called from imperative blocks of other rules.

2.3.3.1.4 Entrypoint and Endpoint Rules

Entrypoint rules and endpoint rules are called rules without parameters. Entrypoint rules are called before application of the declarative (i.e. matched) rules to the input model and endpoint rules are called after the output model has been generated.

2.3.3.1.5 Rule Inheritance

Inheritance is an important concept in object-oriented approaches [Gall95]. A class may inherit fields and methods from its superclass. Inherited methods can be overwritten and new fields and methods can be added to a subclass. One of the proclaimed benefits of using inheritance is that it eases code reuse. Also many design patterns are based on the inheritance concept [Gamma95].

In ATL the general concept for inheritance is transferred to matched transformation rules. When several rules share a common part, this part can be moved upwards in the rule inheritance hierarchy to the parent rule. To specify polymorphic rules the parent rule specifies source and target element names and types which are inherited to its children. Rules may be abstract, i.e. they cannot be applied directly.

A child rule matches a subset of what its parent rule matches. The source pattern must have the same number of elements. Each child source element must correspond to a unique parent source element and each child element type must conform to a type of the corresponding parent element, i.e. be of the same type or a subtype. The guards of the source elements are anded. Further, a child rule specializes target elements of its parent rule. Target elements can be added in child rules. Child target elements with corresponding parent elements, i.e. with the same variable name, can have different types (but must be a subtype of the parent type), and have more bindings or redefine bindings. Only one child rule can match. There may be at most one default subrule including the parent rule if it is not abstract. A default rule is a rule without guard that is matched by default.

The informal semantics for rule inheritance is:

1. root rules (i.e. without parent) are matched,
2. for each potential match, every subrule with a guard is tested,
3. the one that matches, if any, is selected,
4. if none matches, the default rule, if any, is selected,
5. if selected rule is not a leaf (i.e. if it has subrules), then go to 2
6. target elements are created by using the most specific types.

The most specific bindings are used to initialize target elements, i.e. redefined bindings in the parent rule are not executed.

2.3.3.2 Queries

An ATL query corresponds to a model to primitive data type transformation. A query comprises an OCL query expression. In this work queries are used for checking constraints and for the serialization of platform specific models to code. For the latter the query expression iterates over all elements of a specific type in the source model using ATL helpers to recursively construct a string that corresponds to the code. Examples are given in Chapter 5. For checking constraints with queries see 4.1.1.

2.3.3.3 Refining Mode

Two different execution modes are available for ATL modules, the normal execution mode and the refining execution mode. In the normal execution mode an initially empty target model is created from the source model. The refining execution mode is used for inplace transformations, i.e. transformations that modify a given source model resulting in a target model. Technically, in the refining mode of ATL, transformation rules are executed for source elements that should be modified. All other source elements are then implicitly copied to the target model by the transformation engine. Within this work, the normal execution mode is used for the transformation from the platform independent design models to platform specific models as presented in Chapter 5. The refining execution mode is used for the transformation within the platform independent design models as presented in Chapter 4.

2.3.3.4 Tools

The ATL Integrated Development Environment (IDE)[4] is developed as a plug-in for the Eclipse platform[5]. The plug-in comprises an editor with syntax highlighting and code outline, code wizards, the administration of transformation runtime configurations and a debugger. For further details see [ATL05a]. The Kernel MetaMetaModel (KM3) [ATL05d] allows the definition of metamodels in an easy Java-like textual notation, and a number of standard bridges allow the transformation between different textual syntaxes and their corresponding model representations. The definition of a metaclass with KM3 has the following form (cf. 5.2.1):

[4] Homepage for ATL tools: http://www.eclipse.org/m2m/atl/

[5] Homepage for the Eclipse platform: http://www.eclipse.org

```
class Method extends ClassMember
{
    reference parameters[*] ordered container : MethodParameter oppositeOf method;
    attribute body : String;
}
```

In addition to the use of the ATL IDE with Eclipse ATL transformations can also be executed as standalone programs or integrated amongst others into the Java-based build tool Ant[6].

2.3.4 Transformation Modularization

In the same manner as in traditional software engineering software artifacts such as classes and libraries are composed, reused and adapted, it is important that in model driven engineering transformations can also easily be composed, reused and adapted. Transformation modularization into smaller units is an important prerequisite for reuse and helps reducing the complexity of transformations. Based on [Kurtev06a] the basic concepts for transformation modularization are presented. Additionally, a simple transformation metamodel is presented.

Modularization of the metamodel often affects the modularization of the transformation definitions. In general, transformations should be modularized in a way so that tangling and scattering of transformation functionality is minimized. This is not always possible in the case of crosscutting concerns, thus the application of aspect-oriented techniques to the model transformation domain would be interesting. As a general rule transformations should be decomposed along the dimensions of concern by means of the modularization features of a transformation language.

Figure 10 depicts the metamodel for modeling transformation modularization as a profileable extension of the UML metamodel, cf. 2.2.4. Thus, a UML model with a default profile mapping can be used for modeling transformation modularization. All derived properties in the metamodel are derived from their corresponding “intuitive” properties in the UML metamodel. A further formalization is not given here. The upper part comprises the generic concepts that are independent of the specific transformation language. A transformation is a package that contains a set of rules. The basic concepts for transformation modularization are (1) inheritance, i.e. a rule can inherit functionality from its super rule; and rule calls

[6] Homepage for Ant: http://ant.apache.org/

which are specialized to (2) implicit rule calls, i.e. rule calls without explicit references to rule names in declarative approaches; and (3) explicit rule calls in imperative approaches. In hybrid approaches such as QVT and ATL both rule call types are possible. The figure also comprises a specialization for the rule types of ATL as presented in 2.3.3.1. The specialized rules of ATL imply constraints on the possible rule calls: a matched rule is always called implicitly from another matched rule and a called rule is always called explicitly from any rule. Because of these constraints the dependency stereotype may be omitted in a transformation modularization diagram.

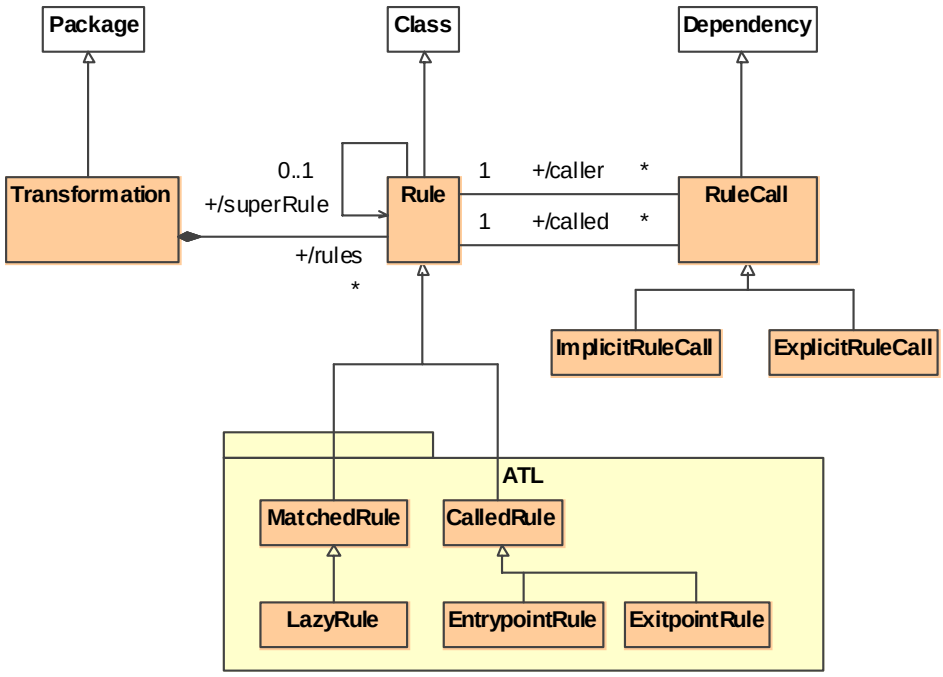


Figure 10. Transformation metamodel

2.3.5 Discussion

From the experiences gained during this work a purely declarative transformation approach such as purely relational or purely graph-based transformations is not practical for real-life transformations. A hybrid approach allows to use imperative constructs for transformation tasks which would be too cumbersome or even impossible to express with declarative constructs. Nevertheless, the declarative constructs of a hybrid approach should be used as far as possible. Thus, the hybrid approaches QVT and ATL and the hybrid graph-transformation based approach VIATRA are best suited for real-life transformation appli-

cations. The former two approaches fit better in the OMG meta architecture as presented in 2.2 with only ATL providing appropriate tool support at the time of writing. VIATRA on the other hand would be best suited for more formal applications due to being based on graph transformations, but unfortunately appropriate tool support was not available at the time of writing. Now, with the latest version from October 2006 increased interoperability between MOF and the internal model representation of VIATRA is available, but still the practical applicability would have to be investigated.

Another important aspect when deciding between transformation languages are the interoperability options between them. Interoperability between ATL and QVT is discussed in [Jouault06a]: QVT transformations can be mapped (by a transformation) to the ATL Virtual Machine, thus QVT transformations can be run in the ATL runtime environment. Inversely, ATL transformations can be mapped (again by a transformation) to QVT operational mappings transformations. On the other hand the research group behind VIATRA proclaims that QVT transformations can be transformed to Abstract State Machine (ASM) and Graph Transformation (GT) rules [VIATRA].

The conclusion is visualized in Figure 11: ATL has the highest interoperability, transformations can be mapped to QVT operational mapping and transitively to VIATRA. QVT transformations can be mapped to VIATRA, but only to the low level ATL Virtual Machine, meaning that QVT transformations can be run on the ATL Virtual Machine, but the transformation specification itself gets lost. VIATRA transformations in general cannot be mapped to QVT or ATL.

Thus, when specific features of QVT are not needed, such as that QVT is a OMG standard or that QVT provides relations together with a visual notation, ATL can as well be used. Additionally, at the time of writing, ATL has the benefit of a stable implementation. Both QVT and ATL can take advantage of the interoperability with VIATRA, which is based on the long established theories on graph transformations, and thus provides a better formal foundation, e.g. for proving the correctness of transformations.

A recent development is the foundation of the Eclipse Model-to-Model Transformations (M2M) project [M2M]. The objective of this project is to provide a framework for model-to-model transformation languages. The core part will provide an infrastructure for plugging in transformation engines. The first transformation engine available under the M2M project is the ATL transformation engine. An implementation of a transformation engine supporting QVT will follow shortly. Another main objective of the M2M project is to provide bridges between transformation languages, i.e. a transformation written in one transformation language could be transformed into a transformation in another language. Thus,

the ATL transformations presented in this work could easily be migrated to QVT transformations, as soon as the corresponding bridging tools are available.

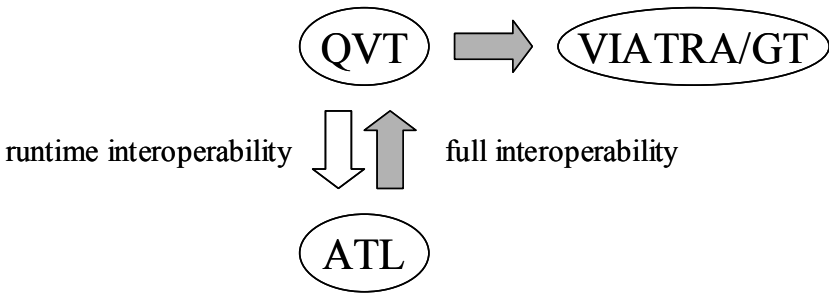


Figure 11. Interoperability between transformation approaches

3 MODEL DRIVEN WEB ENGINEERING

This chapter presents the fundamental ideas of our approach. First, the differences between an elaborationist and a translationist approach and the implications on this work are discussed. Following, the application of the well-known principle of separation of concerns for modeling and for transformation decomposition are presented. After giving an overview over the technical environment that was used to actually run the transformations defined in this work the related work for model driven Web engineering is discussed.

3.1 Elaborationist versus Translationist Approach

There are two interpretations of the MDA vision termed elaborationist and translationist approaches [McNeile03]. Following the elaborationist approach, the specification of the application is built up step by step by alternating automatic generation and manual elaboration steps on the way from PIM to PSM to code [Kleppe03]. For instance, a tool automatically transforms the PIM to a skeleton PSM, which then has to be elaborated by the developer by customizing the generated models and by adding missing details. The apparent problem of this approach is that the lower level models can get out of step with the higher level models by the elaboration activity. Some tools support the regeneration of the higher level models from the lower level models, also called reengineering. The process of full synchronization of the higher level models with the lower level models in both directions is called round-trip engineering. Today, most approaches based on MDA are elaborationist approaches, which have to deal with the problem of model and/or code synchronization. The elaborationist approach could also be seen as a semi-automatization of the familiar object-oriented development approach following analysis, design and implementation steps.

In the translationist approach, the transformations from PIM to PSM and then further to code are fully automatic, i.e. PSMs and code do not have to and must not be elaborated by

the developer. This avoids synchronization problems between higher level models and lower level models. When the PIMs are modified then the PSMs are just regenerated from the PIMs and the code is regenerated from the PSMs. On the other hand, usually the information captured by the PIMs is not sufficient for executing the transformations to the lower level models fully automatically. Thus, additional information, i.e. additional models, is needed as input for the transformations. The translationist approach originates from works on real-time and embedded systems with special emphasis on modeling executable behavior by UML state machines and activities. For more information see [Mellor02].

On the one hand, the approach presented in this work follows the elaborationist approach for the stepwise construction of the platform independent design models of a Web application (see Chapter 4). On the other hand, it follows "primarily" (see next paragraph) the translationist approach because the platform independent design models are automatically transformed to platform specific models, which are then automatically serialized to code (see Chapter 5). These platform specific models and the generated code must not be modified by the developer because roundtrip-engineering is neither necessary nor allowed. Thus, this approach is in line with the objective of the MDA to decouple the technology that an application runs on from the definition of the application.

For traditional non-Web applications a translationist approach would comprise computationally complete models of behavior and the transformation of these models to executable code. Web applications on the other hand are not monolithic applications and they mostly build on software components, for instance a software component providing services for a banking application. Some of these components may already exist and they just have to be integrated in the Web application, thus they do not have to be implemented by the developer. The appropriate term for behavior in this context is service with Web Services being a technology for the implementation of services [W3C02]. This approach focuses on the modeling and transformation of "coarse grained" behavior with so-called process models, see 4.5. Such a process model comprises the composition of "fine grained" behavior by means of UML activities. Fine grained behavior is represented by UML operations which correspond to services. Thus, an operation call corresponds to a service call, for instance the invocation of a Web Service. Process models are transformed to fully executable code, which comprises the invocation and composition of services. Services themselves are not generated by this approach, but implementation skeletons can be generated. Therefore, this approach may be considered as being only "primarily" translationist. This concept for behavior of Web applications fits in the Service Oriented Architecture (SOA) approach [Dostal05] because the basic idea of the SOA approach is to see the realization of a business process as a composition of services. Hence, the application logic of a system is distributed over several independent and loosely coupled services. These services are provided by ser-

vice providers and used by service consumers and to find a service some kind of directory service is necessary. An extension of this approach to additionally include the modeling and transformation of fine grained behavior is an interesting future research topic, see also the conclusions chapter.

On the other hand the automatic transformation to the platform specific implementation models is preceded by the construction of the platform independent design models. The iterative, systematic and stepwise construction of the design models is a key feature of the UWE methodology. The core modeling activities are the requirements analysis, content, navigation, process and presentation design. One contribution of this work to the UWE approach is the automatization of these construction steps with transformations as presented in Chapter 4. These transformations are executed only on the platform independent level and after each transformation follows a manual refinement (i.e. elaboration) step by the developer. As a result of each transformation step a default model is generated, for instance from the content model a default navigation model is generated. These default models are already complete in the sense that the next transformation step could be applied. The only exception is the transition from the analysis model to the design models. All features of classes in the content model and the process model have to be completed manually as the analysis model lacks the necessary details.

Theoretically, it would be possible, following the MDA pattern, to replace a transformation step within the platform independent models followed by a manual refinement step, with a single transformation step that injects the additional information from the manual refinement step from an additional model. Practically, the complexity to define the corresponding metamodels for such additional models that reflect changes to extended UML models and for enabling the developer to maintain these models would be too high. Instead, the approach of this work faces the maintenance and change management challenges imposed by the semi-automatic construction of the design models by two measures. First, incremental updates are taken into consideration in the transformation design allowing the reapplication of transformations without loss of manually added information by the developer, see 4.1.2. Explicit trace models are used to capture the transformation history. Second, OCL constraints for each design model ensure the consistency of modifications carried out manually by the developer.

3.2 Separation of Concerns

Separation of concerns is a well-established general technique in software engineering to reduce the complexity of a system [Dijkstra76]. It is also widely applied for Web application analysis and design. In this work the content, navigation, process and presentation concerns of Web applications are distinguished. At analysis level the content, navigation and process concerns are captured together in the requirements model which is based on using specialized use cases and regular classes. At design level all concerns are addressed separately. Each concern is represented by a corresponding model, i.e. the content model, the navigation model, the process model and the presentation model. More details about the separation of concerns at the analysis and design level are presented in Chapter 4.

On the other hand the vision of the MDA is the automatic transformation of the platform independent design models to models for a specific Web platform, such as for instance the J2EE Web platform or the ASP.NET Web platform as illustrated in Figure 12.

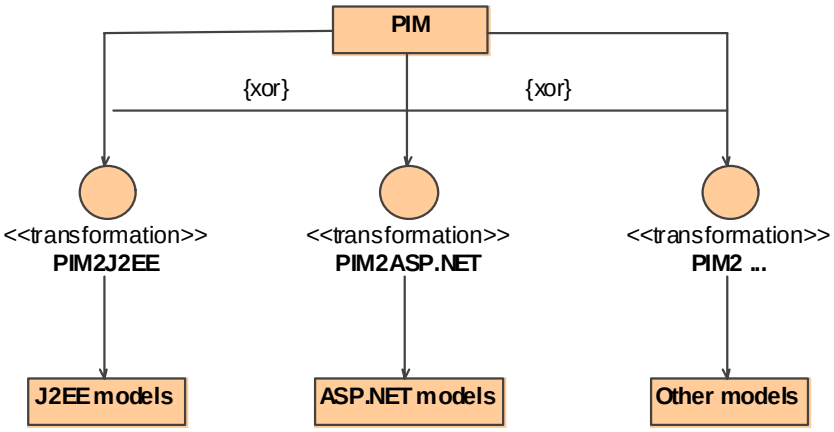


Figure 12. *PIM2PSM* transformations

A strong argument of the MDA is that nowadays technologies are changing rapidly and that for a new platform "just" another transformation has to be constructed together with the corresponding metamodel for the target platform. Unfortunately, this simple idea is in reality too coarse for practical applications. Even in the literature about MDA for Web applications the pattern from Figure 12 is often cited wrongly by mentioning component technologies such as .NET or CORBA as possible target Web platforms (which may only serve for handling the content concern). Also, the proclaimed rapid changes of technology mostly do not happen on the big scale (such as totally new Web platforms every year as for instance J2EE or ASP.NET), but more on the small scale by evolving versions of tech-

nologies or caused by the modular architecture of a Web platform, which allows to plug in different technologies. Thus, it is not sufficient to just consider the target platform as a whole. The focus has to be on the decomposition of the parts of a platform and on the decomposition of the corresponding transformations.

As a result of this discussion, the transformation from the platform independent models to the platform specific models should be decomposed into four different transformations for the content, navigation, process and presentation concerns of a Web application, see Figure 13. Each partial transformation is targeted at a specific part of the Web platform that is responsible for handling the corresponding concern. Of course, one part of the platform could handle several concerns. The Web platform should be designed in a way that one part could be exchanged without influencing the other parts and the corresponding transformations, for example using CORBA instead of RMI as component technology for the content model. Therefore, in this approach the vision of the MDA of platform specific transformations is refined so that parts of a platform can be exchanged separately. Then, only a new transformation and a corresponding metamodel would have to be defined for the exchanged part. More details about the Web platform and the corresponding transformation are presented in Chapter 5.

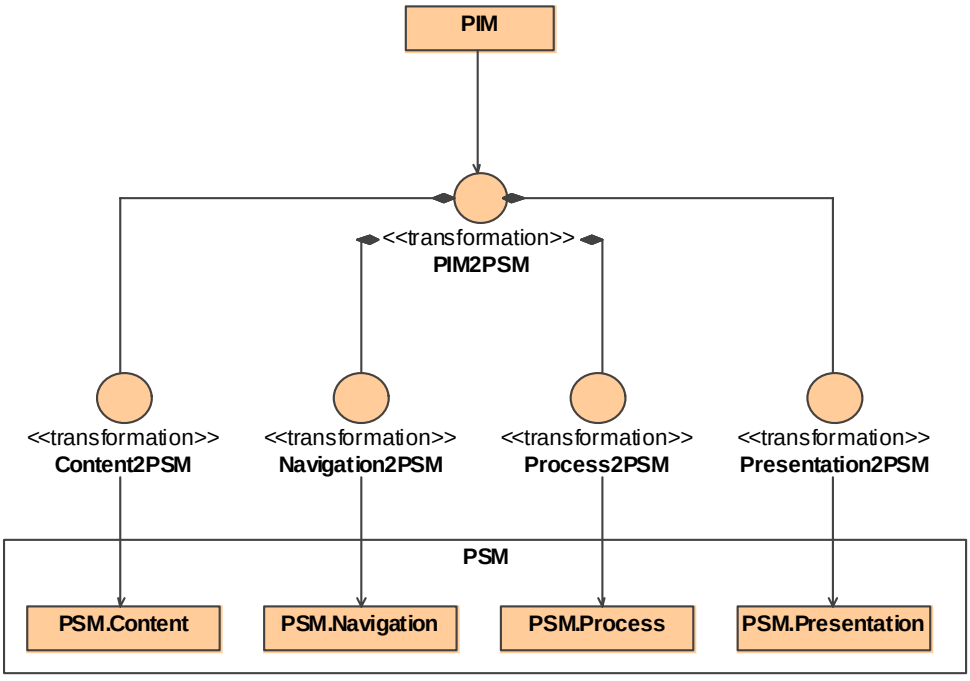


Figure 13. Decomposed *PIM2PSM* transformation

3.3 Transformation Environment

An implementation of the MDA approach is also a technical challenge. In this section technical details of this approach for handling metamodels and transformations are presented.

One of the advantages of MDA as claimed by the OMG is tool interoperability, but often problems arise when dealing with models constructed using a specific tool. For example for UML models not only the version of the UML implemented by a specific tool is relevant, but also the version and type of the meta-metamodel and the version of the model interchange format XMI. Tool interoperability for tools supporting UML 2 and XMI 2, although better than earlier tools supporting the first version of the standards, is still not perfect [Lundell06]. Additionally, not all tools fully comply with the standards or they do not implement all features of a standard such as UML. Further, XMI allows for proprietary extensions, which complicates tool interoperability. A striking example for the misuse of XMI extensions is the version 9.5 of the UML tool MagicDraw, which claims to support UML 2, but saves its models as UML 1.4 models with some added features of UML 2 implemented as proprietary XMI extensions.

This approach does not rely on a specific modeling tool for platform independent analysis and design because it uses the abstraction layer of ATL for handling the external representation of models and metamodels, so called model handlers. When running an ATL transformation, first a model handler is used to read the external model and metamodel representations, then the transformation is run and afterwards the result model is written to an external representation by another (possibly different) model handler. Model handlers can be completely customized and therefore handle any possible data format for models and metamodels. This ensures future tool interoperability. In the worst case a customized model handler would have to be implemented if for a given tool no appropriate model handler is already available.

Currently, two model handlers are available, MDR and EMF, see Figure 14. Both are based on the XMI standard for serializing models. The MDR model handler is based on the NetBeans framework [NetBeans] and allows for the handling of MOF 1.3 and MOF 1.4 metamodel instances, thus it can be used in conjunction with UML tools supporting only the first version of UML such as ArgoUML [ArgoUML], Poseidon [Poseidon] or earlier versions of MagicDraw [MagicDraw]. The EMF model handler is based on the Eclipse Modeling Framework and supports handling of models with an Ecore metamodel [Budinsky03]. Ecore corresponds directly to a subset of MOF called Essential MOF or EMOF [OMG06a], see also 2.2.2.

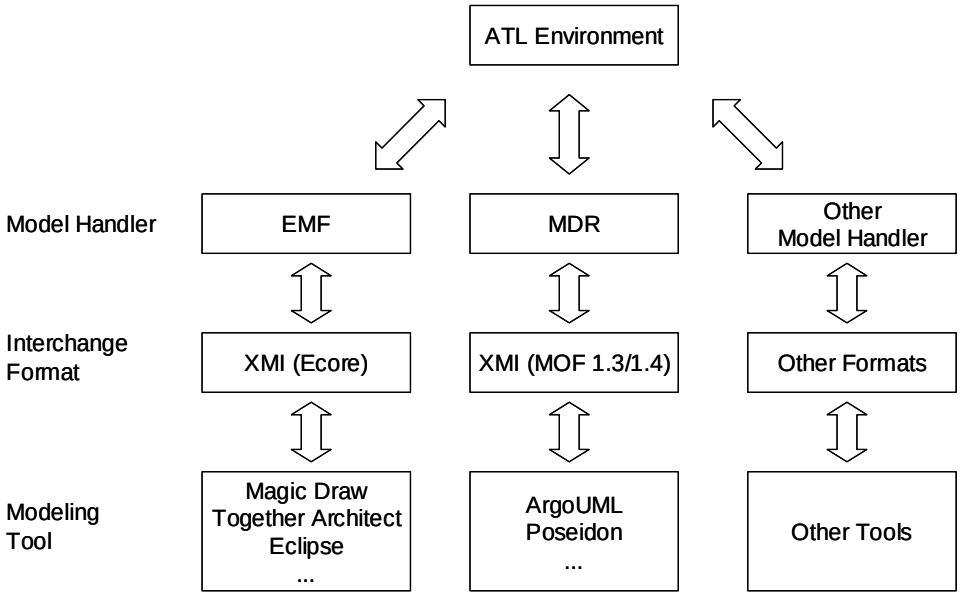


Figure 14. ATL model handlers

For managing the platform independent models and transformations, the transformation environment is based on the Eclipse Ecore implementation of UML 2, which is supported by a broad range of UML 2 tools such as for instance MagicDraw 11.6, Borland Together Architect 2006 for Eclipse or Eclipse itself. For other UML 2 tools it is possible to write bridge transformations or customized model handlers. Unfortunately, with the inclusion of process modeling, which is based on UML 2 activities, UML 1 tools are intrinsically incompatible with this approach due to massive changes in the UML metamodel concerning activities. UML 1 tools could only be used for the modeling of static Web applications, i.e. Web applications without processes.

The platform specific metamodels presented in this work were defined using the Kernel MetaMetaModel (KM3) [ATL05d], which allows the definition of metamodels in an easy Java-like textual notation. These KM3 metamodels were converted to Ecore metamodels by using an ATL built-in bridge for the conversion of different meta-metamodel standards.

For running all transformations the ATL Eclipse plug-in was used, which comprises an editor with syntax highlighting and code outline, code wizards, the administration of transformation runtime configurations and a debugger. A screenshot of the plug-in is shown in Figure 15. For further details see [ATL05a]. It is planned to additionally provide a standalone transformation environment, which does not need the Eclipse environment to be run and is based on the Java deployment tool Ant.

The transformation environment together with the necessary documentation for running the transformations is available on the UWE homepage [UWE].

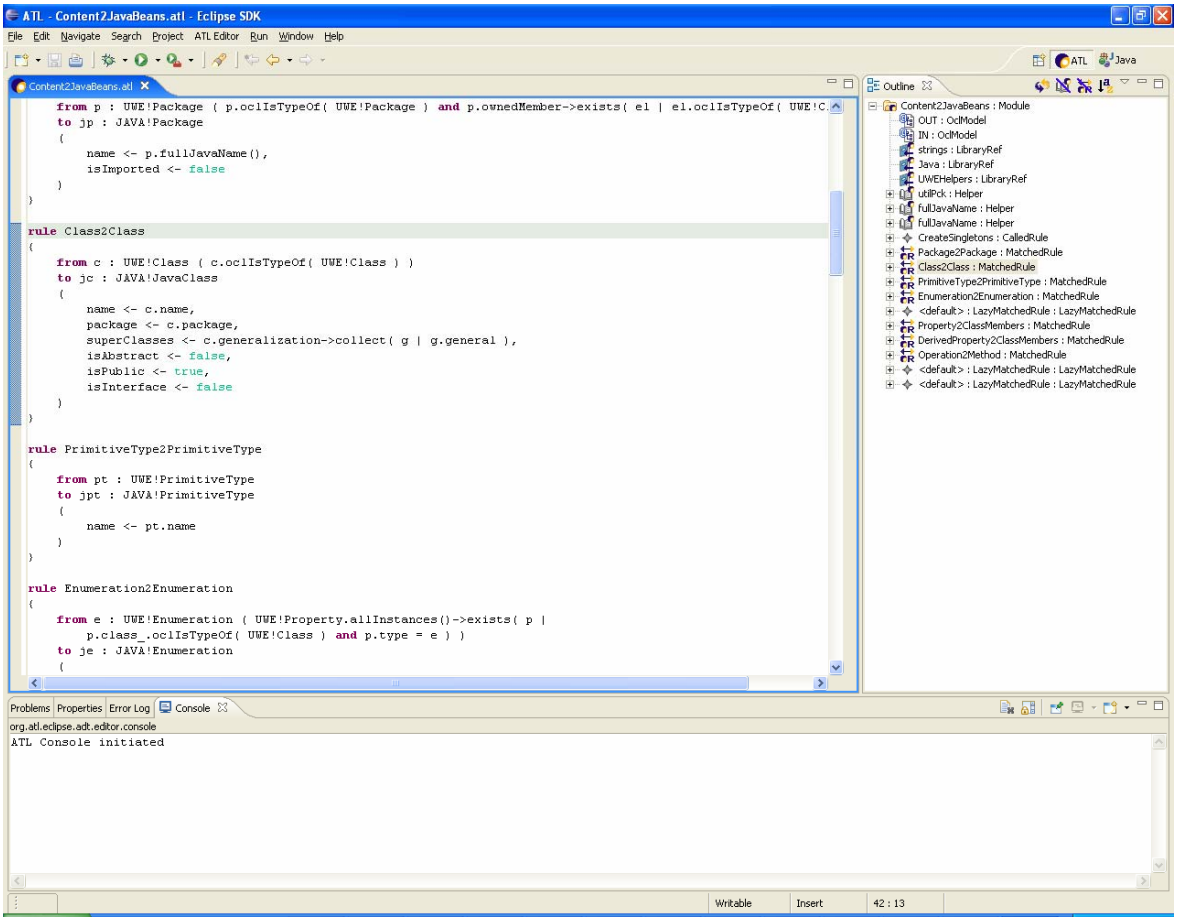


Figure 15. The ATL Eclipse plug-in

3.4 Related Work

In this section an overview of the related work for model driven Web engineering is given.

3.4.1 UML-based Web Engineering (UWE)

UWE is an object-oriented, iterative and incremental approach [UWE] based on the Unified Modeling Language (UML) [OMG05a]. The main focus of UWE is the systematic design followed by a semi-automatic generation of Web applications. A UML profile (cf. 2.2.4.) is used as notation making use of all benefits and tools that support UML. UWE

evolved from an object-oriented methodology for hypermedia design presented in [Hennicker00]. This methodology provides guidelines for the systematic and stepwise construction of models. The core modeling activities are the requirements analysis, content, navigation and presentation design. In [Koch01a] the adaptivity aspect was added to the approach together with a reference model for adaptive Web applications and a complete description of the development process, which is based on the Unified Software Development Process [Jacobson99]. The use of statechart and interaction diagrams for modeling Web scenarios, activity diagrams for modeling tasks (i.e. processes) and deployment diagrams to document the distribution of Web application components was illustrated in [Koch02a].

3.4.1.1 ArgoUWE

The design phase of the UWE development process is supported by the CASE tool ArgoUWE as presented in [Knapp03]. It is implemented as a plug-in module of the open source modeling tool ArgoUML [ArgoUML]. ArgoUWE implements the UWE metamodel[7], and the semi-automatic UWE development steps are realized by directly manipulating the corresponding UWE models, cf. 2.3.1.3. OCL well-formedness rules of the UWE metamodel that allow the designer to check the consistency of the UWE models during editing are also directly implemented with Java code. Although ArgoUWE could be extended to support a major part of the metamodel and transformation rules for analysis and design of Web applications as presented in Chapter 4 of this work, ArgoUWE underlies some severe restrictions that are caused by being based on ArgoUML. A major restriction is that ArgoUML and hence ArgoUWE is still based on UML 1.4, thus only metamodels based on UML 1.4 can be easily integrated, although some features of UML 2 could be simulated with high efforts. The approach presented in this work is based on UML 2, thus the metamodel presented in this work cannot be easily implemented with ArgoUWE. For example the UML metamodel for activities changed drastically from UML 1.4 to UML 2. As the approach for process modeling presented in this work is based on UML 2 activities the adoption of ArgoUWE to support processes as presented here[8] is not possible without rewriting ArgoUML to support UML 2 activities.

[7] In this case an older version of the UWE metamodel with additionally added modeling elements for editing purposes

[8] The current version of ArgoUWE supports an earlier approach to process modeling based on UML 1.4 activity diagrams, see [Knapp05]

One objective of this work is, in contrast to ArgoUWE, to externalize metamodel and transformations for analysis and design of Web applications. Externalize means that the metamodel and the transformations are not hard-coded into the tool. There are no restrictions on the employed modeling tool as long as it supports UML 2 profiles and stores models in the standardized model interchange format and transformations are based on a standardized transformation language in contrast to the direct-manipulation approach of ArgoUWE.

3.4.1.2 UWEXML

UWEXML, an extension of UWE, was the first model driven approach for Web engineering by the author of this work [Kraus02]. The Model Driven Architecture (MDA) was still in its infancy and so was a standardized transformation language for model driven development. Thus UWEXML relied on hard-coded transformations, see 2.3.1.1. UML design models defined with ArgoUWE or any other modeling tool are automatically mapped to XML documents with a structure conforming to their respective XML Schema definitions. Further, XML documents for the content model are automatically mapped to content DOM objects (Document Object Model). DOM objects corresponding to interactional objects are automatically derived from content DOM objects and/or other interactional DOM objects. The XSLT mechanism serves to transform the logical presentation objects representing the user interface to physical presentation objects, e.g. HTML or WAP pages. The transformation is based on a production system architecture for Web applications using the XML publishing framework Cocoon [Cocoon], which provides a very flexible way to generate documents comprising XSLT and XSP (eXtensible server pages) processors.

Figure 16 shows a UML class diagram that represents the UWEXML process overview in a generic way including all models that are built when developing Web applications with UWEXML. The process starts with analysis and design models created by the user in an editor. The design models are transformed by the UWEXML Preprocessor into XML representations which are fed – together with XML documents containing parameters for the generation process – into the UWEXML Generator. The generator generates on the one hand artifacts which can directly be deployed to an application server providing a physical component model and to an XML publishing framework, denoted by the «import» dependency. On the other hand some of the generated artifacts have to be adapted before deployment, denoted by the «refine» dependency. The generator can be customized to a certain degree for different technologies (i.e. the target platform) by exchanging the Java implementation for the web and/or component technology dependent parts. The UWEXML approach was abandoned because it relied on hard-coded transformations and was not flexible enough, in favor of a more generic approach as presented in this work.

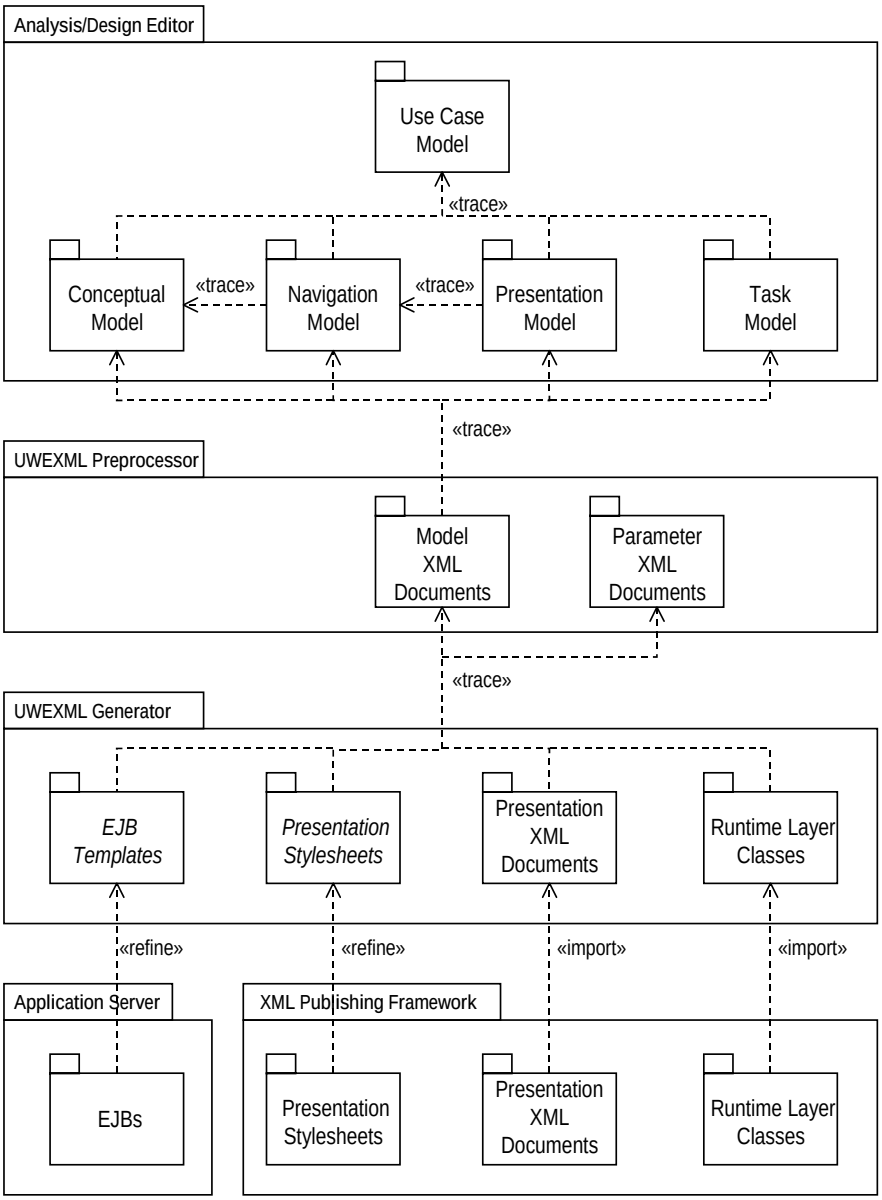


Figure 16. UWEXML process

3.4.1.3 Transformation Techniques and Model Driven Process

An overview of all model driven activities currently investigated by the UWE research team is presented in [Koch06b]. The proposed global UWE process is depicted in Figure 17 as a stereotyped UML activity diagram. This work realizes by transformations a subset of the proposed global UWE process which is essentially the global UWE process minus adaptation modeling, architecture modeling and big picture modeling. The realization of the missing points of the global process is part of the future work of the UWE research group, see also the conclusions chapter.

Models are represented with object nodes and transformations as stereotyped activities (special circular icon). A chain of transformations then defines the control flow. The process starts by defining a requirements model or business model, called computation independent model (CIM) in the terms of MDA. Platform independent design models (PIM) are derived from the requirements model, see [Koch06a]. The set of design models represents the different concerns of Web applications. It comprises the content, the navigation, the process, the presentation and the adaptation concern of Web applications. The next step in the global approach is to integrate the design models mainly for the purpose of verification into a so-called big picture model by graph transformations using the AGG tool (cf. 2.3.1.5). The big picture model is based on UML state machines, which can be checked by the tool Hugo/RT, a UML model translator for model checking and theorem proving [Knapp06]. In a joined work with the author of the WebSA (Web software architecture) approach (cf. 3.4.1) the inclusion of a separate architecture model for capturing the architectural features of Web applications was investigated [Meliá05a]. In the global approach it is proposed to integrate the architecture model with the big picture model to an integrated platform independent model covering functional and architectural aspects. Architecture modeling is further future work of the UWE research group. It is not considered in this work because the author claims that the architecture of a Web application is tightly coupled to the target platform and implicitly encoded in the transformations to the platform specific models, and therefore it is more important to provide a way for transformation and platform modularization to support different architectures (or platforms). The last proposed step in the global process is the transformation of the integration model to platform specific models, just like it is realized in this work.

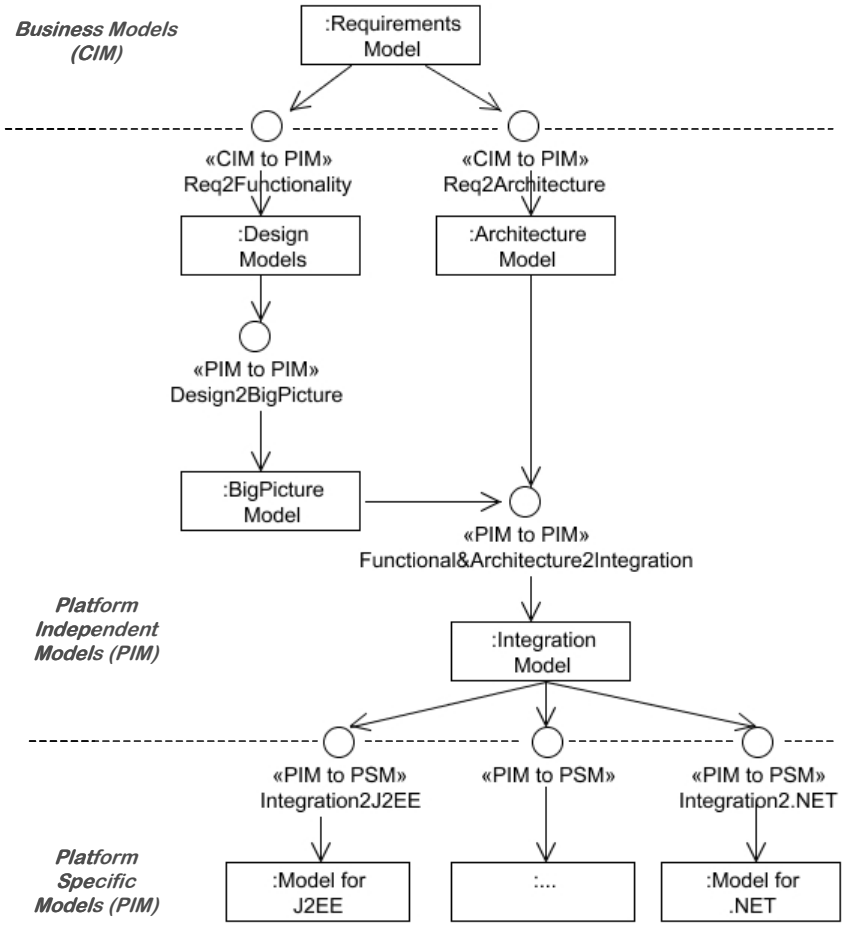


Figure 17. Global UWE process overview (from [Koch06b])

3.4.2 WebSA

The Web Software Architecture (WebSA) approach [Meliá06a] is not a stand-alone approach for model driven Web engineering. It complements other Web design approaches by providing an additional viewpoint for the architecture of a Web application and a model driven development process based on the Unified Process [Jacobson99]. The author claims that the approach can be used in combination with the functional models of every other approach for Web design that is based on a MOF metamodel such as for example UWE, see also 3.4.1.3.

The architecture of a Web application is defined by the means of two architecture models, the subsystem model and the configuration model. The former defines the architectural layers of a Web application and the latter an architecture of implementation components.

The architectural models are integrated with the functional models to a so called integration model by means of a transformation at the platform independent level. The integration model is then transformed to a platform specific model by another transformation.

The metamodels of WebSA are based on MOF and additionally a corresponding UML 2 profile is defined. For the transformation to the integration model a proprietary transformation language called UPT (UML Profile for Transformations) [Meliá06b] is defined which allows the specification of transformations by using a UML profile. For the transformation to the platform specific implementation the transformation language MOFScript is used.

As already stated in 3.4.1.3, the proposal of the integral UWE approach is to integrate the functional models of UWE with the architecture models provided by WebSA. This stands in contrast to this work because the author claims that the architecture of a Web application is tightly coupled to the target platform and implicitly encoded in the transformations to the platform specific models, and therefore it is more important to provide a way for transformation and platform modularization to support different architectures (or platforms). Most important, the resulting integration model in the WebSA approach is enriched with low-level artifacts for the architecture, which still have to be refined by the developer. This hinders a translationist approach by having to complement one more model for the transformation to code.

3.4.3 MIDAS

MIDAS [Cáceres04] is another model driven approach for Web application development based on the MDA approach. For analysis and design it is based on the content, navigation and presentation models provided by UWE and it uses therefore UML with UML profiles as notation. For the platform specific implementation it relies on object-relational techniques for the implementation of the content aspect and on XML techniques for the implementation of the navigation and presentation aspects. A process aspect is not supported. The transformations for mapping the design models to the specific target platform are not defined formally.

3.4.4 WebML

WebML [Ceri02] is a data-intensive approach based on entity relationship modeling. Until now WebML does not use an explicit metamodel. The corresponding tool WebRatio internally uses a Document Type Definition (DTD) for storing content and navigation models, i.e. a grammar-like definition for the structure of XML documents. DTDs do not have the same expressiveness as MOF and lack an easily understandable notation. The XML transformation language XSLT (cf. 2.3.1.8) is used for model-to-code transformations support-

ing presently transformations to Java and JSPs. XSLT is not suitable for more complex transformations and the development of XSLT programs is difficult and error prone.

Currently, efforts are made for enhancing the interoperability of WebML with other model driven Web engineering approaches. In [Schauerhuber06b] an interesting approach is presented to semi-automatically transform a DTD to a MOF compatible metamodel. The transformation uses a set of transformation rules and heuristics, but still requires some user interaction for improving the semantics of the generated metamodel. This approach is applied to WebML, thus enabling the use of standardized MDE technologies for WebML. Additionally, in [Moreno06] a first step towards a UML 2.0 profile for WebML is presented that would also enable standardized MDE technologies.

3.4.5 OOWS

OOWS [Fons03] is an extension of the object-oriented software development method OO-Method [Pastor01] for Web application development. Similar to this work a navigation model represents the navigational aspects of a Web application as views of classes from a class diagram which is similar to the content model. The presentation aspect is integrated with the navigation aspect, a dedicated presentation model for further abstraction of the user interface is not available.

Recently, a model driven extension of OOWS to support business processes has been proposed with emphasis on the integration of external applications, the development of special dedicated user interfaces that guide through processes and the consideration of automatic as well as manual tasks [Torres06]. Therefore, the navigation model of OOWS has been extended by the inclusion of graphical user interface elements to allow for the interaction between users and business processes using a UML-like notation.

Processes are captured in the business process model using an extended version of the Business Process Modeling Notation (BPMN) [OMG06c] and a corresponding extended metamodel for business modeling. BPMN stems from the B2B (business-to-business) field and is similar (but not identical) to UML activities, which stem from the software engineering field. It provides concepts such as flow objects (corresponding to activity nodes or events), connecting objects (corresponding to activity edges), swimlanes and artifacts (corresponding to object nodes). In contrast to this work also manual tasks are considered in the process model, i.e. tasks that are manually carried out by humans and not automatically by the system by invoking operations or Web services.

The business process model is the starting point for model transformations. Operational Mappings, the imperative part of QVT (see 2.3.2.2), is used as transformation language.

The transformations are implemented with Borland Together Architect 2006 for Eclipse (see 2.3.2.3). The process model (in BPMN) is transformed to a (platform independent) default navigation model for the user interaction with the process, which then has to be refined by the designer. The navigation model is then transformed to a concrete Web technology, i.e. a platform specific model. The proposal will be integrated in the ONME tool[9] for automatic code generation.

In contrast to the approach of this work processes are not represented by a dedicated model, but distributed over the BPMN process model and the navigation model. The latter contains a lower level view of the process model, where the constructs of the process model are resolved into navigation constructs. The approach of this work defines a clear separation of the process and the navigation aspect and specifies how processes are integrated in the navigation model, see also 4.5. Further, the strength of the approach of this work is also the use of standards for all aspects of Web application development. Finally, although the use of the imperative part of QVT is comparable to the use of ATL, it is unclear how much of the model driven approach is already realized.

3.4.6 HyperDE

In contrast to the heavyweight approach presented in this work there is an increasing interest in the use of small domain specific languages that can be used for agile development [Cockburn01] and for quick prototyping.

The open source Web framework Ruby on Rails (or short Rails) [Thomas06] is especially suited for the agile development of Web applications. It is based on the reflective and object-oriented programming language Ruby, which provides extensive metaprogramming possibilities and facilitates the use of internal domain specific languages. The Rails framework allows the development of Web applications following the Model/View/Controller (MVC) pattern. Following its two guiding principles called “don’t repeat yourself” and “convention over configuration” much less code and configuration data than with other Web frameworks is necessary. For example the mapping between classes and database tables is derived automatically from class and field names. A technique called scaffolding allows rapid prototyping by quickly providing most of the logics and views for common operations, such as CRUD (create, read, update and delete database operations).

[9] OlivaNova Model Execution System, CARE Technologies, www.care-t.com

A modification of the Ruby on Rails framework called HyperDE is presented in [Nunes06]. HyperDE is based on SHDM, a method for the design and implementation of Web applications for the semantic Web [Lima03]. The MVC implementation of Ruby on Rails is extended by navigation primitives of SHDM and the persistence layer is modified to operate on a RDF database [Lassila99] where the user defined navigation model and application instance data is stored. Additionally, HyperDE provides a domain specific language by which model instances can directly be manipulated. HyperDE itself provides a Web interface, so that Web applications can directly be created or modified.

3.4.7 Moreno et al.

Moreno et al. focus on the integration of Web applications with third party systems, following the MDA approach. In [Moreno05c] a high-level model based integration framework for interoperation with third party systems is presented. They argue that for service-oriented scenarios a central content model is not appropriate and that for interoperation with external systems explicit models for required and provided interfaces, processes etc. are necessary. Although a rich and complex set of modeling constructs is presented, no explicit semantically rich metamodel is defined. Instead, a set of platform independent models divided into the layers user interface, business logic and data is introduced. For each concept of such a model, the mapping to a UML stereotype is described textually. In addition, no details about transformations are given, but the authors state that they will use QVT in their future work. In [Moreno05a] this approach is applied to CORBA, EJBs and RMI and in [Moreno05b] adapted to modeling and integration of cooperative portlets. A portlet is an individual Web-based component that typically handles requests and generates only a fragment of the total markup that a user sees from his or her browser [Díaz04].

3.4.8 Muller et al.

Another interesting model driven approach stems from Muller et al. [Muller05]. Like the approach taken in this work total code generation is an important goal. But, in contrast to this work, a heavyweight non-profilable metamodel is used for the hypertext model and the presentation model because the authors argue that an extension of the UML would not be appropriate for giving a sufficient degree to model designers. Nevertheless UML is used for the business model. Presentation modeling is based on using templates. A language called Xion is used to express constraints and actions. Further, it serves as a query language for abstraction of data access and it is also a platform independent action language based on OCL and transferable to different target platforms. The whole approach is supported by a visual model driven tool called Netsilon.

3.4.9 W2000

W2000 [Baresi06] originates from the HDM methodology (Hypertext Design Model, [Garzotto93]), a hypermedia and data-centric Web design approach, but it also adopts some features from the UML to support the concept of business processes. It distinguishes the information (i.e. content), navigation, service (i.e. process) and presentation concerns. W2000 follows a similar approach for process modeling as presented here although operations are defined separately from the content model. It is suggested that either activity diagrams or collaboration diagrams are used to define the workflow of processes, but it is left unclear how these processes can be executed or translated to code.

The metamodel of W2000 is defined as a MOF metamodel with a UML profile as notation. Like in this work OCL constraints are used to ensure the well-formedness of models. As presented in [Baresi05], the graph-based transformation language AGG (cf. 2.3.1.5) can be used for modifying the (platform independent) design models, which is supported by a corresponding tool implemented as a Eclipse plug-in. W2000 does not give concrete guidelines for the construction of the platform independent models, although they could be implemented by AGG transformations in a similar way as in this work. The mapping of the platform independent models to a platform specific implementation is part of the future work of W2000, with the next step being to automatically derive J2ME (Java 2 Micro Edition) client applications for mobile devices.

4 PLATFORM INDEPENDENT ANALYSIS AND DESIGN

This chapter presents details about the analysis and design phases of the model driven approach. The aim of the analysis phase is to gather a stable set of requirements. The functional requirements are captured by means of the requirements model. The design phase consists of constructing a series of models for the content, navigation, process and presentation aspects at a platform independent level. Transformations implement the systematic construction of dependent models by generating default models which then have to be refined by the designer.

The Web Engineering field is rich in design methods, supporting the complex task of designing Web applications (see also 3.4). These methods propose the construction of different views comprising at least a content model, a navigation model and a presentation model (although naming them differently). Each model is built out of a set of modeling elements, such as nodes and links for the navigation model or image and anchor for the presentation model. In addition, all these methodologies define or choose a concrete notation for the constructs they define.

Although all methodologies for the development of Web applications use different notations and propose slightly different development processes, they could be based on a common metamodel for the Web application domain. A metamodel is a precise definition of the modeling elements, their relationships and the well-formedness rules needed for creating well-defined models. A particular methodology based on this common metamodel may only use a subset of the constructs provided by the metamodel. A common Web application metamodel should therefore be the unification of the modeling constructs of current Web methods allowing for their better comparison and integration. A first proposal for such a common metamodel for Web application development was presented in [Kraus03a] and [Kraus03b]. Since then this metamodel has evolved as presented here. In this work the objective is on the model driven approach and thus the metamodel of this work is restricted to the core aspects and elements of UWE that are currently fully supported in the model driven process. These core aspects of Web applications are requirements, content, naviga-

tion, process and presentation. The integration of the remaining aspects for adaptivity and architecture into a model driven approach is part of future work.

In the following sections each step for the construction of the analysis and design models of a Web application is presented, comprising the corresponding part of the metamodel together with a description of the respective transformations written in ATL that implement the systematic construction of models, and the activities needed for the manual refinement. These transformations are inspired by the informal rules for the systematic development of Web applications described in [Hennicker00]. An excerpt from the case study introduced in 1.3 serves as a running example. The metamodel is defined as a "profileable" extension of the UML2 metamodel [OMG05a], cf. 2.2.3. It provides a precise description of the concepts used to model Web applications and their semantics and it is structured into different packages as depicted in Figure 18. The dependencies in the diagram represent the dependencies between the corresponding models, hence matching the order for the systematic construction of models. The profile definition for the metamodel is given in the appendix A.

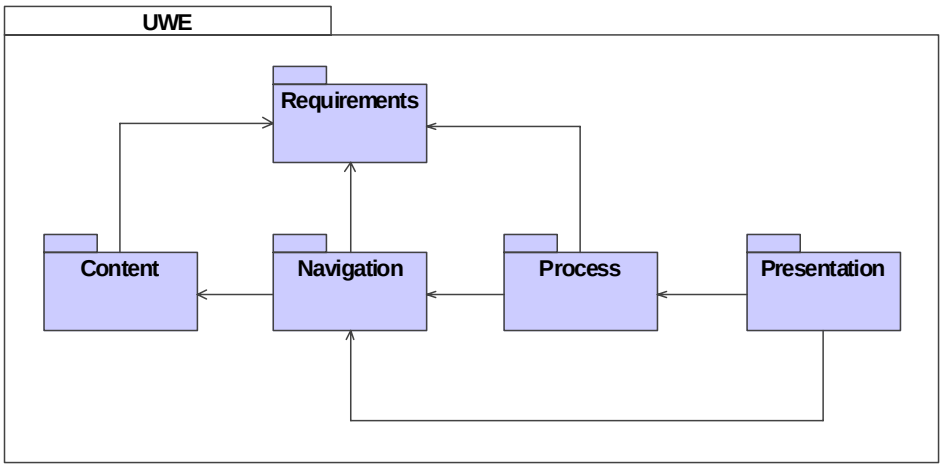


Figure 18. Metamodel Package Structure

Prior to the definition of the metamodel some clarifying comments about the meaning of the terms metamodel and model as used in this work are needed. It is important to distinguish between global metamodels, global models and views. This chapter defines a global metamodel for analysis and design of Web applications. This global metamodel is structured along the concerns of Web applications into several views represented by packages of the global metamodel as depicted in Figure 18. A concrete Web application is represented by exactly one global model that conforms to this global metamodel. For each view of the global metamodel a corresponding view of the global model is defined, comprising

all modeling elements that conform to the corresponding view of the global metamodel. In the following, a specific view of the global model will be called a model and a specific view of the global metamodel will be called a metamodel. For example the navigation model is equal to the navigation view of the global model of a Web application which comprises all modeling elements that conform to the navigation metamodel. The relationships between models and metamodels are illustrated with an example in Figure 19: a global model conforms to the global metamodel; it comprises a content model and a navigation model which conform to the corresponding metamodels, i.e. all modeling elements of a specific model are instances of metaclasses of the corresponding metamodel.

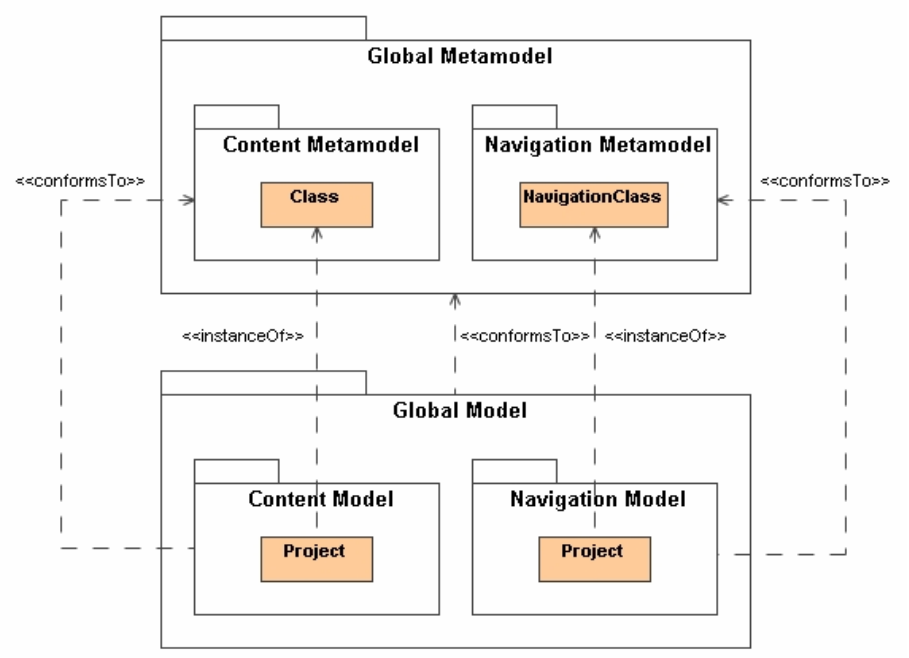


Figure 19. Relationships between models and metamodels

The systematic and stepwise construction of models by means of transformations and manual refinement steps corresponds to the stepwise refinement of the global model. An example for the global model after the derivation of the navigation model from the content model is depicted in Figure 19. Using exactly one global metamodel helps to avoid consistency problems between models because well-formedness rules can then be defined globally, i.e. across model borders. It is further important to note that the transformations presented in this chapter always operate on one global model. Before and after running a transformation all the well-formedness rules for the global metamodel are checked in order to ensure the correctness of the global model before and after running the transformation.

Concerning the correctness of the transformations, the approach benefits from the design of the ATL transformation language. ATL transformation rules are intrinsically confluent due to the fact that source models are read-only and target models are write-only, i.e. it is ensured that the execution order of the rules does not have an impact on the resulting output model.

When comparing the metamodel of this approach with other Web approaches it has to be taken into consideration that the metamodel is designed to be self-contained. This means that every model element is either used for deriving other dependent model elements (as presented in later sections) or/and is needed for the automatic generation of code. Apart from the model driven Web approaches discussed in 3.4 also pure Web design approaches are taken into consideration for a comparison with this approach. Finally, an outline to the extension possibilities for the metamodel is given in each section.

4.1 General Techniques

In the following sections first some general techniques are presented which are required in the rest of the chapter.

4.1.1 Checking Well-Formedness of Models

In this work OCL class invariants attached to metamodels are used to define the well-formedness rules for models. The following example constraint is taken from the metamodel for modeling requirements of Web applications presented in 4.2.1. The constraint that a Web use case must have exactly one subject of type *Class* is written in OCL as:

```
context WebUseCase inv WebUseCaseContentClass :
   self.subject->one( c | c.oclIsTypeOf( Class ) )
```

Note that the property *subject* is defined in the superclass *UseCase* of the UML metamodel, and in the subclass *WebUseCase* of our metamodel this property is constrained.

Before the execution of a transformation, the well-formedness rules for the source model are checked by evaluating an ATL query which is composed of OCL expressions (cf. 2.3.3.2). Therefore all class invariants for the source model are translated manually to an ATL query following the schema presented in the following. A query with the name *CheckConstraints* evaluates a model to a boolean value. Running this query corresponds to checking all constraints for a model, and the value of the query indicates if all constraints

are fulfilled. For each constraint to check, an ATL helper method *check_<constraint name>* with a boolean return value is defined for the same context as in the OCL invariant declaration. All the helper methods defined like this are called from the expression body of the ATL query *CheckConstraints* for all elements of the context type. An additional helper method *assert* is used to log those model elements which do not fulfill a constraint.

```
helper context UWE!NamedElement def : assert( checkResult : Boolean, constraintName : String )
    : Boolean = if checkResult then true else
        false.debug( self.oclType().toString() + ' ' + self.fullName() + ' Constraint ' + constraintName )
    endif;
```

The helper method *assert* is defined for all named elements and takes the result of the constraint checking and the name of the constraint as arguments. All model elements presented in this work are named elements, i.e. they specialize either directly or indirectly the UML metaclass *NamedElement*. If the checking fails for a model element then its type, its name and the name of the constraint is written to the console by using the predefined helper *debug*. Only if the checking for all constraints and for each element of the corresponding context type is successful, then the overall query returns true, indicating that all constraints are fulfilled. The corresponding ATL code for the example above is:

```
query CheckConstraints =

    UWE!WebUseCase.allInstances()->forAll( x |
        x.assert( x.check_ WebUseCaseContentClass(), 'WebUseCaseContentClass' ) ) and

    … check further constraints

helper context UWE!WebUseCase def : check_ WebUseCaseContentClass() : Boolean =
    self.subject->select( c | c.oclIsTypeOf( UWE!Class ) )->size() = 1;
```

As the method body of ATL helpers is defined with OCL expressions, the original expression for the OCL class invariant can be adopted almost unaltered except for the following necessary modifications:

- Prepending of metamodel names before the names of types, e.g. *UWE!Class* instead of *Class*
- Substitution of OCL constructs not yet supported by ATL, e.g. *any()* or *one()*
- Quoting of ATL keywords, e.g. *"context"* instead of *context* (does not occur in the example above)

- Usage of *if-then-else-endif* expressions if parts of an OCL expression are possibly undefined, e.g. *if a.oclIsUndefined() then OclUndefined else a.b endif* instead of *a.b*

Details about the peculiarities concerning the usage of OCL in ATL are presented in [ATL06a]. For the implementation of the constraint checks for the UWE metamodel as presented in this chapter see B.2.1.2.

4.1.2 Transformation Traces

In order to capture the transformation history during the semi-automatic construction of the design models of a Web application a way to track the execution of transformations, i.e. transformation traces, is needed. These traces are used on the one hand to support incremental updates of the models, allowing the reapplication of transformations without loss of manually added information by the developer. On the other hand, the trace information is also needed to resolve relationships between modeling elements transformed by transformation rules of previous transformations runs.

To capture transformation trace information, a new metaclass *TransformationTrace* is introduced, which is a specialization of the UML metaclass *Abstraction,* which in turn is a specialized dependency, see Figure 20. In this chapter metaclasses from the UML metamodel are in general presented with a white background. This is conform with the usual way of defining trace dependencies in UML because the predefined stereotype «trace» is also an extension of the metaclass *Abstraction*. The notation for dependencies with a dashed line and an arrow is inherited from UML dependencies. The name of the transformation trace denotes the name of the transformation rule that created the trace.

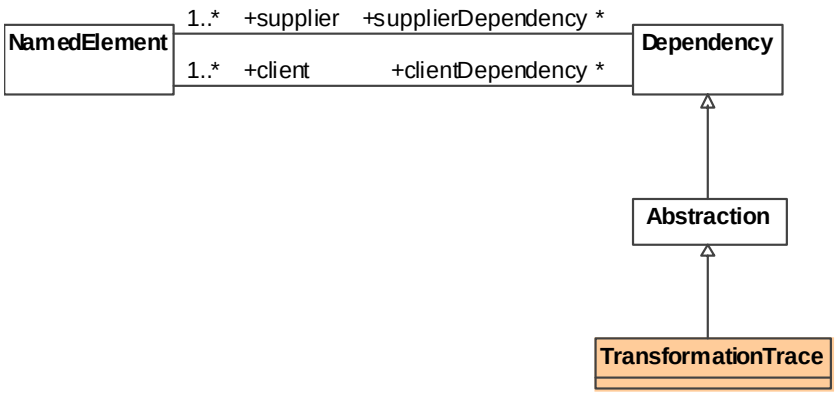


Figure 20. Metamodel for transformation traces

In Figure 21 an example for a transformation trace is given for a Web application model comprising a content class *Project* with one attribute *name*. The (global) model is refined by the transformation *RequirementsAndContent2Navigation* presented in 4.4.2.1. The resulting refined (global) model comprises an additional navigation class which was created by the rule *ContentClass2NavigationClass* from the content class *Project* as denoted by the corresponding transformation trace. Additionally, the attribute *name* of the content class was mapped to a corresponding navigation property of the navigation class by the rule *Property2NavigationProperty* as denoted by another transformation trace.

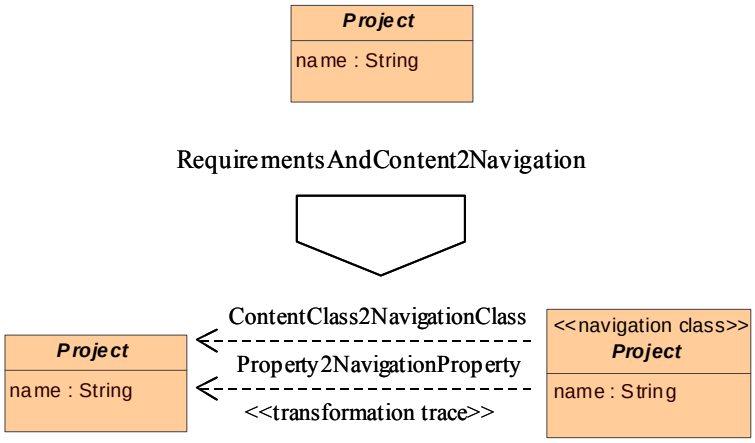


Figure 21. Example for transformation trace

The following ATL rule *CreateTrace* is called from the imperative part of a transformation rule presented in the following sections to create a transformation trace.

```
rule CreateTrace( sourceEl : UWE!NamedElement, targetEl : UWE!NamedElement,
    ruleName : String )
{
    to t : UWE!TransformationTrace
    (
        name <- ruleName,
        supplier <- Set( UWE!NamedElement ) { sourceEl },
        client <- Set( UWE!NamedElement ) { targetEl }
    )
}
```

To get the source or target elements of a transformation trace the ATL helpers *getTraceSource* and *getTraceTarget* are used while *hasTraceSource* and *hasTraceTarget* are used to query if a trace already exists.

```
helper context UWE!NamedElement def : getTraceSource( ruleName : String )
    : UWE!NamedElement =
    let ts : Set( UWE!NamedElement ) = UWE!TransformationTrace.allInstances()->
        select( t | t.name = ruleName and t.client->includes( self ) )->
        collect( t | t.supplier )->flatten() in
            if ts->size() > 0 then ts->asSequence()->first() else OclUndefined endif;

helper context UWE!NamedElement def : hasTraceSource( ruleName : String ) : Boolean =
    not self.getTraceSource( ruleName ).oclIsUndefined();

helper context UWE!NamedElement def : getTraceTarget( ruleName : String ) :
    UWE!NamedElement =
    let ts : Set( UWE!NamedElement ) = UWE!TransformationTrace.allInstances()->
        select( t | t.name = ruleName and t.supplier->includes( self ) )->
        collect( t | t.client )->flatten() in
            if ts->size() > 0 then ts->asSequence()->first() else OclUndefined endif;

helper context UWE!NamedElement def : hasTraceTarget( ruleName : String ) : Boolean =
    not self.getTraceTarget( ruleName ).oclIsUndefined();
```

An example for using transformation traces to support incremental updates is depicted in Figure 22. The Web application model after the first application of the transformation *RequirementsAndContent2Navigation* depicted in Figure 21 was extended by adding a new attribute *description* to the content class *Project*. When the transformation *RequirementsAndContent2Navigation* is run again then the transformation traces created in previous runs are used to determine which new model elements should be created. For the content class *Project* a transformation trace for the rule *ContentClass2NavigationClass* already exists, hence this class is not matched by this rule. The same holds for the attribute *name* and the rule *Property2NavigationProperty*, but it does not hold for the new attribute *description*, hence a corresponding new navigation property and a new transformation trace are created.

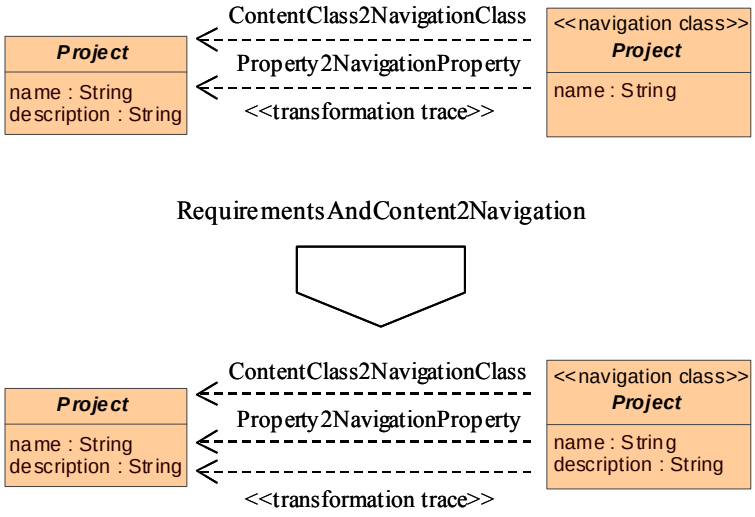


Figure 22. Example for using transformation traces for incremental update

The general pattern for a transformation rule that matches only model elements for which a corresponding transformation trace does not yet exist and that creates a new transformation trace looks as follows:

```
rule TrafoX
{
    from x : MM!X ( not x.hasTraceTarget( 'TrafoX' ) )
    to y : MM!Y ( … )

    -- imperative part of the rule
    do
    {
        CreateTrace( x, y, 'TrafoX' );
    }
}
```

If a transformation rule extends another transformation rule, then there is no need to handle trace information in the subrule, i.e. no additional imperative part is needed for that purpose.

In addition to incremental updates, trace information can also be used to query the results of previous transformation runs, i.e. to query the source and target elements of a specific transformation rule of a previous transformation run. In the following example a Web process element was first mapped to an operation element by the transformation rule *SimpleProcess2Operation* in a previous transformation run. In another transformation the

transformation rule *CreateProcessDataAndFlowForSimpleProcess* presented in 4.5.2.2 initializes a local variable *o* with the target element of the transformation rule *SimpleProcess2Operation* of the previous transformation run.

```
rule CreateProcessDataAndFlowForSimpleProcess
{
    from pc : UWE!ProcessClass ( … )
    using
    {
        o : UWE!Operation = pc.webProcess.getTraceTarget( 'SimpleProcess2Operation' );
        …
    }
    to …
}
```

The inclusion of transformation traces sometimes makes transformations rules cumbersome to read, especially for refinement transformations as those presented in this chapter. Therefore, those parts of the rules that handle transformation traces are not included in this chapter. For their detailed code see B.3.

4.1.3 Expression Language

Some of the model elements presented in the following sections require that the Web application developer can specify expressions for the Web application model. Expressions are used for:

- Definition of derived attributes of navigation classes (simple expression)
- Guard expressions for links (simple expression)
- Formatting expressions for user interface elements (formatting expression)

The most natural choice for UML based metamodels would be the use of the expression part of the Object Constraint Language (OCL). Although a standardized metamodel for OCL exists, this is not supported by most modeling tools for further use in a model driven environment. Instead, OCL expressions are exported in a textual representation. Within a model driven environment these textual expressions would have to be parsed, and the corresponding OCL model would have to be instantiated. Additionally, links to the user model would have to be reestablished.

Any expression language could be used in conjunction with the metamodel presented in this chapter. However, due to the complexity of OCL expressions as a general language for the use in model transformations and since the metamodel for OCL expressions is anyway not supported by tools, the use of a simpler expression language was preferred in this work, facilitating the transformation of these expressions to code.

Expressions are represented as textual values of type string. The chosen expression language used in the following is the unified expression language (unified EL) from the J2EE environment used for Java Server Pages (JSP) and Java Server Faces (JSF), for details about the syntax see [J2EE]. Summarized, this expression language allows:

- Accessing of model instance properties at runtime
- Navigation expressions ("dot" notation as in OCL)
- Accessing collections of elements
- Using arithmetic, logical, relational and conditional operators. In addition a special operator *empty* is provided
- Invocation of operations of model instances

The unified expression language is used in two different ways for simple expressions and formatting expressions. For simple expressions the unified EL is used in its pure syntax while for formatting expressions an additional syntactic construct allows the concatenation of expressions and string literals represented by a formatting string.

Simple expressions use the pure syntax of the unified EL and they must always access the implicit context variable *self* provided by the runtime environment. The simple expression *"not empty self.projects"* depicted in Figure 23 is for example used for the guard expression of a link to a Web process indicating that this link should only be accessible if the value of the property *projects* of the implicit context variable *self* is not empty.

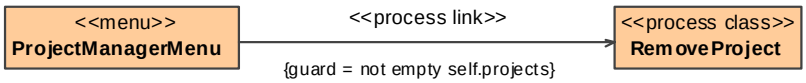


Figure 23. Use of an expression language

Formatting expressions are represented by a formatting string, such as for example the formatting string *"Validation Project ${name}"* which is used for formatting the label of

an anchor to a validation project. Each occurrence within the formatting string using the simple syntax *${expr}* represents an expression *expr*. The formatting string is evaluated from left to right. Each expression is coerced to a string and then concatenated with any intervening text. The resulting string value represents the value of the formatting expression.

4.2 Requirements

Although requirements analysis is a key factor in the development of software systems, only few Web approaches pay special attention to requirements. A detailed requirement analysis can help to reduce costs at later development stages. Different techniques can be used for requirements specification, from informal textual descriptions to formal specifications [Kappel03a].

In this approach UML use cases are used to define the functional requirements of a Web application while UML classes represent the content requirements. Use cases are a well proven technique for specifying the functional requirements of a software system, not least because the UML provides a graphical and intuitive notation for use cases. In contrast to requirements description techniques such as stories or other techniques that rely on the use of natural languages, use cases provide a sufficient degree of precision required for the use in a model driven approach because the syntax of use cases is accurately defined by the UML metamodel.

The requirements specification presented here comprises the construction of an analysis content model for defining the structure and data of a Web application, and the construction of a Web use case model for the definition of the functionality of a Web application. The analysis content model uses UML classes and associations to specify the identified concepts from the problem domain that should provide the functionality represented by the Web use cases. Different types of Web use cases are introduced for treating static and dynamic functionality, which corresponds to the navigation and the process aspect, respectively.

Following the comparative study given in [Escalona04b], (not necessarily model driven) Web methodologies that pay special attention to requirements analysis are mainly NDT [Escalona04a], OOHDM [Schwaabe98] and W2000 [Baresi06]. Other approaches use either classical techniques or ignore this phase of the development process. All of the above mentioned approaches start the modeling process by defining UML use cases to capture the functional requirements. Similar as in this work, W2000 distinguished two types of use

cases representing navigation and process functionality. NDT has its main focus on requirements, thus it is the most comprehensive approach to requirements modeling. However, in addition to general use cases formatted templates are used to further detail the requirements specification. A formatted template is a table with specific fields that have to be completed by the developer. Because the entries of such a table are written in a natural language the application of model driven techniques is difficult. OOHDM uses a special technique called user interactions diagrams (UID) to specify user interactions in the requirements phase. These diagrams correspond to activity diagrams in our approach with the difference that here user interaction is modeled at the design level in a more detailed way represented by process flow models (see 4.5.2). Generally, only functional and content requirements are considered in this approach, while for example NDT also considers non-functional requirements.

While this approach shares the basic ideas for requirements modeling with UWE, some subtle differences exist. As stated in [Koch06b], UWE uses the WebRE approach for modeling requirements presented in [Escalona06] and [Koch06a]. Although WebRE defines different use case types for navigation and process functionality, the metamodel presented in the next section further introduces additional use case types for specialized kinds of processes. The main difference to this work is that WebRE (and thus UWE) proposes modeling elements for are more detailed specification of the navigation, process and presentation concerns at the requirements level. These additional modeling elements are used within activities that describe the behavior of the corresponding Web use cases. The approach of this work abstains from introducing more details for these concerns at requirements level but introduces similar concepts as in WebRE, but with a finer granularity, during the construction of the navigation and the process models.

The metamodel for requirements modeling presented here could easily be extended by introducing new Web use case types or by extending existing use case types to support new modeling aspects of Web applications, such as for example for personalized Web applications. Further, by defining additional Web process use case types, special kinds of Web application functionality could be handled in a particular way. For instance, special Web process use case types could represent database operations of data-intensive Web applications.

4.2.1 Metamodel

The modeling elements of regular UML class diagrams without operations are used for the analysis content model to capture the structure and data of a Web application. On the other hand specializations of UML use cases are used for the Web use case model with the ab-

stract metaclass *WebUseCase* as super type of Web use case types as depicted in Figure 24. The context for the functionality of a Web use case is represented by the derived attribute *contentClass*. The term context as used here refers to the abstract location where the user of the Web application is currently located. The context is represented by the content classes. Within a given context the user can perform certain functionalities represented by the Web use cases. The optional change of the context when executing a web use case is represented by the derived attribute *target*. The target attribute is represented by an association between a Web use case and a content class. Two concrete Web use case types are distinguished: *Navigation* and *WebProcess*.

Navigation use cases represent navigation functionality, i.e. the static functionality of a Web application, in a very abstract way. Static means that the state of the content objects does not change when executing a navigation use case. The only navigational detail that can be expressed relevant at analysis level is the target content class that should be reachable by navigation, represented by the derived attribute *target*. A constraint ensures that there is an association defined in the analysis content model between the associated content class of the use case and the target content class. Additional information such as direction and multiplicities is not relevant for the analysis content model.

Dynamic functionality of Web applications is represented by the *WebProcess* use case type. The execution of a Web process typically changes the state of the system by executing actions on the content model. The *WebProcess* use case type represents the general case of a Web process with an arbitrary workflow. Two specialized Web process types are further defined, edit use cases and simple processes, in order to treat some common functionality of Web applications especially. Edit use cases represent data modification functionality of the associated content class. Simple process use cases represent the atomic invocation of behavior of the associated content class. A Web process, which is neither an edit use case nor a simple process is also called a complex process. A simple process always implicitly defines a trivial workflow containing an action that invokes the corresponding behavior. The special treatment of simple processes allows the automatic generation of a corresponding operation in the content model and the corresponding trivial workflow in the process model in the following steps of the methodology. If the execution of a Web process changes the context, i.e. the context shall change from the content class of the Web process to another content class resulting from the execution of the Web process, then this has to be represented by the attribute *target*.

Note that the create, retrieve, update and delete (CRUD) operations found in data-intensive Web approaches such as for example WebML [Ceri02] are realized in this work by the more general Web processes: create and delete operations are realized by simple processes

and update operations are realized by edit processes. The retrieve operation does not have to be realized explicitly.

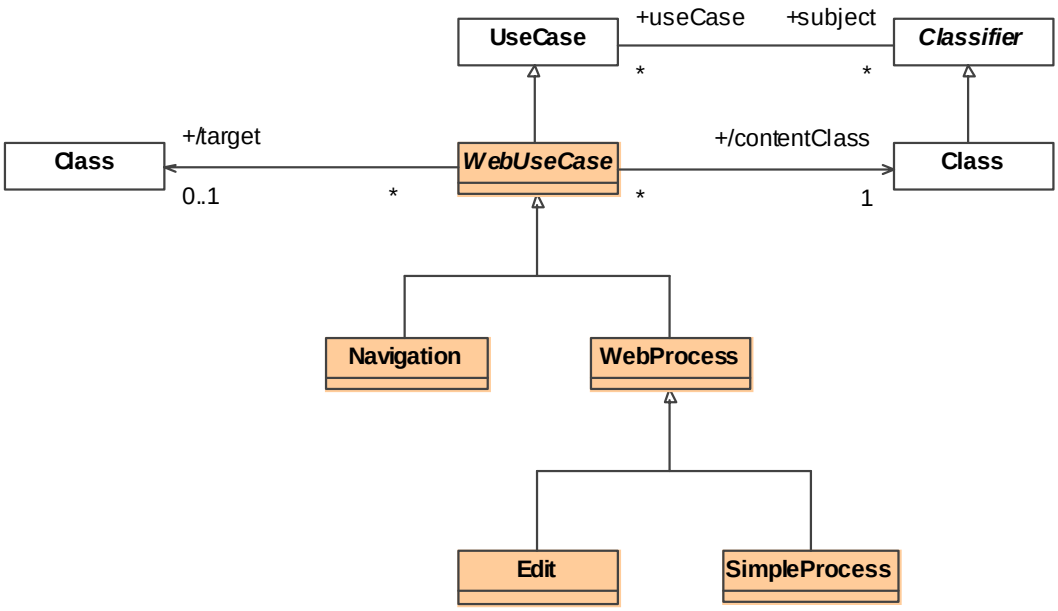


Figure 24. Metamodel for requirements modeling

Derived Attributes

The derived attribute *contentClass* of a Web use case is defined to be exactly the one element from the subject collection of the use case which has the exact type *Class*, see constraint *WebUseCaseContentClass*.

```
context WebUseCase def : contentClass : Class =
    self.subject->any( c | c.oclIsTypeOf( Class ) )
```

If an association between a Web use case and a content class exists, then the derived attribute *target* of this Web use case is defined to be this content class, see also constraint *WebUseCaseTarget*. If no such association exists then the target is undefined.

```
context WebUseCase def : target : Class =
    Association.allInstances()->
        select( a | a.endType->size() = 2 and a.endType->includes( self ) )->
        collect( a | a.endType->excluding( self )->first() )->any( t | t.oclIsTypeOf( Class ) )
```

Constraints

A Web use case must have exactly one subject of type *Class*.

```
context WebUseCase inv WebUseCaseContentClass :
    self.subject->one( c | c.oclIsTypeOf( Class ) )
```

At most one association between a Web use case and a content class may exists.

```
context WebUseCase inv WebUseCaseTarget :
    Association.allInstances()->
        select( a | a.endType->size() = 2 and a.endType->includes( self ) )->
        collect( a | a.endType->excluding( self )->first() )->
        select( t | t.oclIsTypeOf( Class ) )->size() <= 1
```

For a navigation use case a target content class has to be defined and a corresponding owned attribute has to exists for the corresponding content class.

```
context Navigation inv NavigationTarget :
    self.target->notEmpty() and self.contentClass.ownedAttribute->exists( p | p.type = self.target )
```

For an edit use case no target content class must be defined.

```
context Edit inv EditTarget :
    self.target->isEmpty()
```

Notation

The UML notation proposes that a use case should visually be located inside its subjects, i.e. its content class, although not all UML modeling tools support the visual nesting of use cases inside classes. In order to avoid name collisions, it is suggested that a content class owns its Web use cases. The target of a Web use case is notated as an association between the use case and the corresponding content class. For the definition of the corresponding UML profile see A.1.

4.2.2 Analysis Content: Example

The analysis content model is a class model that captures structure and data of a Web application. A part of the DANUBIA case study introduced in 1.3 serves as a running example for this chapter. For a more detailed presentation of the case study see chapter 6. Its main objective is the *management* of environmental *projects*, in short named projects in the following. Two different kinds of projects are distinguished, *user projects* and *validation projects*. A *user project* serves to examine certain questions, e.g. “how will the expected frequency of the occurrence of extreme discharge at a gage P change within the next 100 years?”. A user project can be associated to a *validation project* which is used to validate simulation configurations. For the two different kinds of projects an inheritance relationship has been introduced in the analysis content model. Further, attributes (without type

information) have been added, such as *name* and *description* of a project. The analysis content model resulting from this description is depicted in Figure 25.

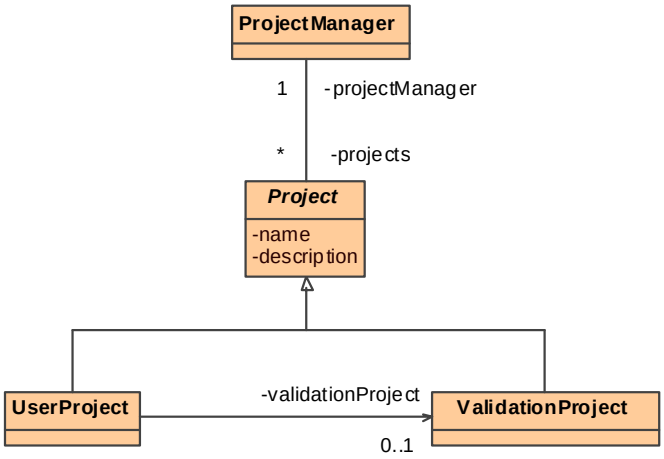


Figure 25. Analysis content model

4.2.3 Web Use Cases: Example

Web use cases, i.e. specialized UML use cases, are used for modeling the required functionality of a Web application. For this purpose, for each class from the analysis content model a use case diagram is constructed that comprises the analysis content class and all of the corresponding Web use cases which are placed inside the box representing the analysis content class.

The Web use cases for the project manager are depicted in Figure 26. The Web process *Add Project* expresses that the user can add new projects. The Web process *Add Project* is not a simple process because it requires a dedicated workflow in which we would like the user first to decide which kind of project he wants to add. Then he should enter exactly the information necessary for the selected kind of project. The target content class *Project*, which is represented by an association between the Web process use case and the content class, specifies that after the completion of the process the resulting project is shown to the user. The simple process *Remove Project* expresses that the user can remove a project. The navigation use case *View Projects* expresses that the user can navigate to the list of projects, represented by the association to the corresponding content class, i.e. the target of the navigation use case. For user projects, the user can navigate to the corresponding validation project target and the user can edit the user project as depicted in Figure 27. All constraints for the requirements model are fulfilled because each Web use case has exactly one analysis content class as subject. Further, no more than one association between a specific

Web use case and a content class exists. For each navigation use case a target is defined and a corresponding association exists in the analysis content model. Finally, for the edit use case a target is not defined.

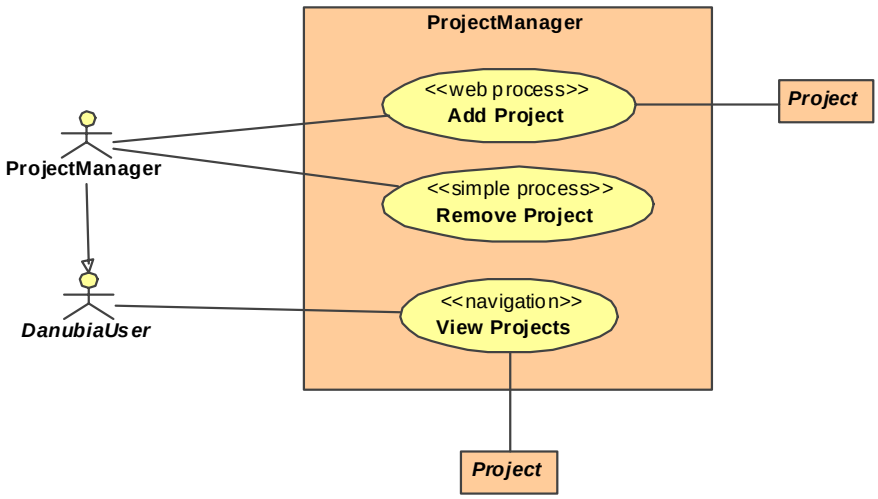


Figure 26. Use cases for content class *ProjectManager*

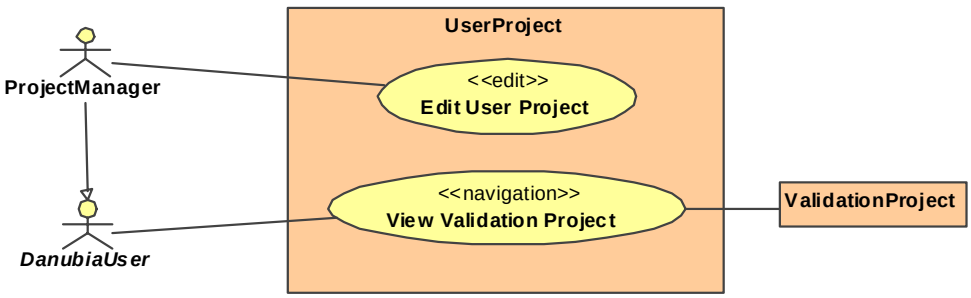


Figure 27. Use cases for content class *UserProject*

4.3 Content

The objective of content modeling is to define the structural and behavioral aspects of the problem domain of a Web application. The structural aspects correspond to the information space of a Web application while the behavioral aspects correspond to the atomic units of

behavior, see below. Navigation and presentation aspects are not taken into account when constructing the content model.

Well-known object-oriented modeling activities, which are as well applied to traditional non-Web application development, are the foundation for content modeling, thus regular UML classes represent the problem domain of a Web application. In addition to static structural features (attributes and associations) which are referenced in the navigation model, the content model also comprises dynamic behavioral features (operations) which are referenced in the process model.

Content modeling is normally based on either entity relationship (ER) diagrams [Chen76] or object-oriented techniques using UML class diagrams. Data-intensive approaches such as WebML [Ceri02] or W2000 [Baresi06] originate from the field of database systems and are hence based on entity relationship diagrams for modeling the information space. However, ER diagrams cannot be used to represent the behavioral properties of an application. Therefore, approaches based on ER diagrams have to define additional modeling elements for expressing the dynamic aspects of the problem domain. WebML for example introduces the concept of operations at the hypertext, i.e. navigation, level, thereby breaking up the separation of the content and the navigation concerns. Other approaches, including UWE, are based on object-oriented methods, and thus most of them use UML classes for content modeling in a very similar way as the approach presented in this work.

One limitation of this work is that behavior is modeled at the granularity of operation signatures. Thus, the implementation of operations themselves cannot be generated automatically but has to be either predefined by e.g. a Web service, or only implementation skeletons can be generated, which then have to be completed by the developer. Possible ways of extending this approach by modeling executable behavior of operations are discussed in the conclusions chapter.

The content model is automatically derived from the requirements model by applying the transformation *Requirements2Content* presented in 4.3.2. The resulting default content model has to be refined by the developer by adding additional classes, attributes, operations, associations etc.

4.3.1 Metamodel

Content modeling does not require any additional constructs. Regular UML classes are used for content modeling in order to make content modeling as similar as possible to the modeling of traditional non-Web applications.

4.3.2 Transformation Requirements2Content

The transformation *Requirements2Content* depicted in Figure 28 automatically derives a content model from the requirements model, i.e. the analysis content model and the Web use case model. It comprises two transformation rules which are outlined below and detailed in B.3.1. The well-formedness of the input model is checked before the transformation is executed, see 4.1.1. Further, for each rule transformation traces are generated as described in 4.1.2.

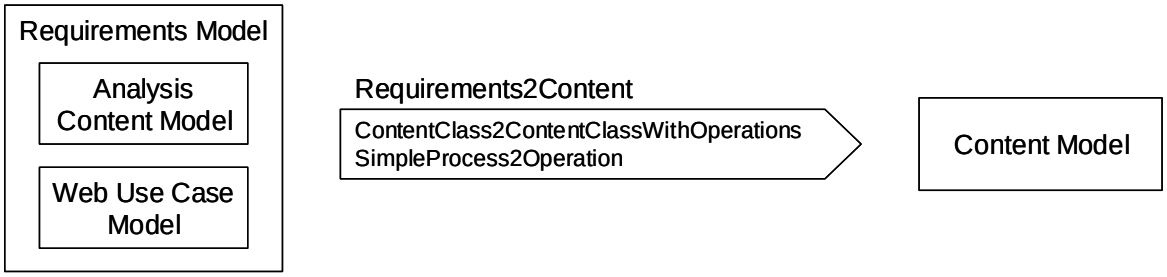


Figure 28. Transformation *Requirements2Content*

The resulting content model for the analysis content model depicted in Figure 25 and the Web use case model for the project manager depicted in Figure 26 is shown in Figure 29. In comparison to the analysis content model the operation *removeProject* for the simple process use case *Remove Project* has been added. The other Web use case types affect only the automatically derived models at later steps during the semi-automatic construction of the design models: navigation use cases determine the navigation model, while all Web process use case types (simple processes, complex processes and edit processes) are required for the construction of the process model.

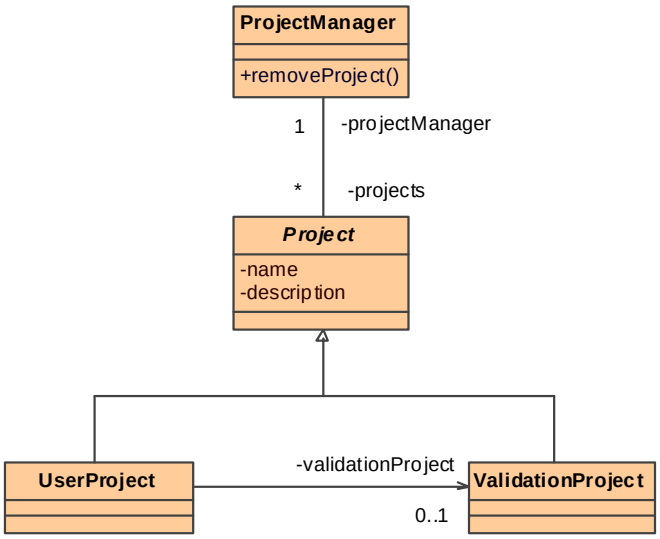


Figure 29. Content model derived by transformation *Requirements2Content*

Rule ContentClass2ContentClassWithOperations

The rule *ContentClass2ContentClassWithOperations* maps each content class to a content class with added operations created by the second rule *SimpleProcess2Operation*. Note that the ATL expression

```
thisModule.resolveTemp( sp, 'op' )
```

is necessary to reference a specific target element *op* of the rule *SimpleProcess2Operation* that matches the source element *sp*, see also [ATL06a]. The effect of these rules for a part of the running example is depicted in Figure 30. In comparison to the rule implementation in B.3.1 target bindings that are not relevant for understanding the rule have been omitted.

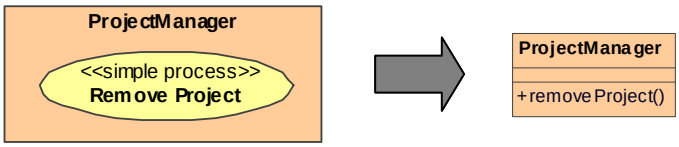


Figure 30. Illustration of the rules for adding operations

```
rule ContentClass2ContentClassWithOperations
{
    from c : UWE!Class ( c.oclIsTypeOf( UWE!Class ) )
```

```
    to tc : UWE!Class
    (
        ownedOperation <- c.ownedOperation->union(
            c.useCase->select( uc | uc.oclIsKindOf( UWE!SimpleProcess ) )->
            collect( sp | thisModule.resolveTemp( sp, 'op' ) ) ),
        …
    )
}
```

Rule SimpleProcess2Operation

For each simple process in the requirements model an operation is generated. The name of the operation is calculated by discarding all spaces from the name of the simple process and converting the first character to a lower case representation. The functions *regexReplaceAll* (for replacing substrings by using regular expressions) and *firstToLower* are provided by ATL [ATL06a]. An auxiliary rule *Type2ReturnParameter* (detailed in B.3.1) is used to generate the return parameter of the operation.

```
rule SimpleProcess2Operation
{
    from sp : UWE!SimpleProcess
    to tsp : UWE!SimpleProcess ( … ), -- target for copying source element
    op : UWE!Operation
    (
        name <- sp.name.regexReplaceAll( ' ', '' ).firstToLower(),
        type <- sp.target,
        ownedParameter <- if sp.target.oclIsUndefined() then Sequence {} else
            Sequence { thisModule.Type2ReturnParameter( sp.target ) } endif
    )
}
```

4.3.3 Manual Refinement

The automatically derived content model has to be manually refined by the developer to add on the one hand model properties that were not present in the analysis content model. On the other hand the developer can enrich the content model by model elements which were not relevant at the requirements level.

The following actions are mandatory for a valid content model for the further steps in the model driven process if the corresponding information is not yet available in the analysis content model:

- Specify the types for all attributes, this may require the definition of new types, such as for example enumeration types
- Specify the multiplicities for all properties
- Specify which ends of an association are navigable
- Specify names for all navigable (in terms of UML properties) association ends

Other actions, which have an influence on the models in the following steps, are optional. Optional means that the model is valid without any of these actions:

- Add additional classes, attributes, operations and associations
- Specify if a property is ordered for all multi-valued ends of an association
- Add inheritance and abstract classes
- Add parameters and return types to operations. Note that for operations which are used in simple processes, parameters correspond to the input of an operation call and the return type to the output of an operation call in the workflow of the process.

The automatically derived design content model for the running example was manually refined by specifying attribute types and adding additional attributes such as the *id* attribute. The association end *projects* has been declared as ordered. A parameter *project* has been added to the operation *removeProject* to specify that the user first has to enter the project he wants to remove. The resulting content model is depicted in Figure 31. The detailed content model for the case study is presented in 6.1.2.

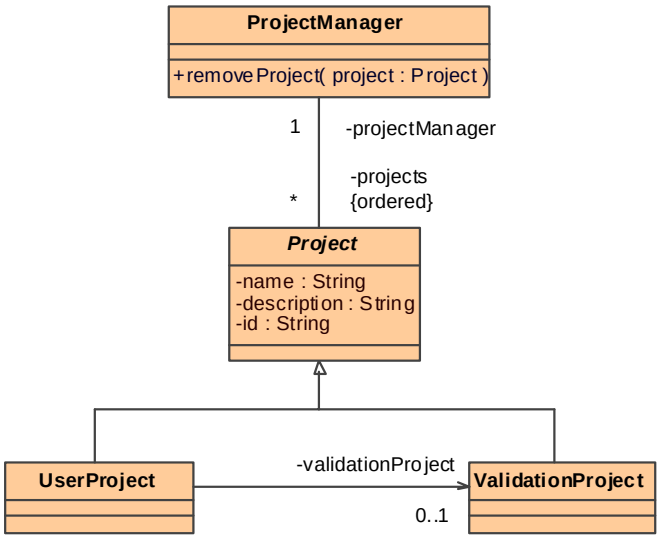


Figure 31. Refined design content model

4.4 Navigation

The objective of navigation modeling is to specify the navigability through the content of a Web application, i.e. to define a static navigation view of the content. Nodes represent information from the content model and links express the navigation paths between nodes.

In this approach nodes are specialized UML classes and links are specialized UML associations. In contrast to other approaches the pure UML notation is used for navigation modeling in order to provide a uniform notation for the metamodel. The navigation model is constructed in several steps as presented in the following sections. The first step is called the navigation space model. It specifies which nodes can be visited by direct navigation from other nodes. After the construction of the navigation space model access structures are added to the navigation model. Finally, menus organize the outgoing links of navigation classes.

This approach shares the central concepts of nodes and links with other Web approaches. Some approaches merge additional concerns, which are represented in this approach by separate models, with the navigation concern. OO-H [Cachero02], WebML [Ceri02] and OOWS [Fons03] use the navigation model for representing the process concern. The latter additionally merges the presentation concern with the navigation concern. Apart from the

distinct separation of concerns, the main differences between this approach to navigation modeling and other approaches are the different kind of nodes and links defined.

The special node types for navigation modeling in WebML are pages, content units and operation units. Pages are containers of content units that are presented together to the user of the Web application. Content units are views on the entities from the content model. Operation units model arbitrary actions that can be triggered during navigation. WebML provides a set of predefined operations for typical database actions. Although in this approach there is no direct correspondence to the concept of a page, nodes that belong together can be expressed by composition relationships. Content units correspond to navigation classes and access primitives from the navigation model and process classes from the process model. Operation units correspond to call operation actions in the process flow model. In contrast to WebML data manipulation operations are not predefined (but might be in a future evolution), thus all kind of operations are treated in the same way and this approach does not presume the existence of a database. WebML further distinguished contextual and non-contextual links. Contextual links carry context information. In WebML contextual links are required for technical reasons to transport database identifiers. In the approach of this work the context of a link is always implicitly given by the participating navigation properties of a link (see next section).

OO-H distinguished the following node types: navigation targets, navigation classes, service nodes and collections. Navigation targets serve as containers for other nodes and are used for structuring the navigation space. Service nodes represent the invocation of an operation from the content model and collections represent the choice of an outgoing link, i.e. a menu. Navigation targets correspond to nested navigation classes. Navigation classes correspond to navigation classes and collections to menus. As already mentioned above, OO-H merges the modeling of processes with navigation modeling. Each service node corresponds to a call operation action in the process flow model. Additionally, OO-H defines different link types which can all be mapped to elements of the navigation metamodel of this approach.

In addition to the concept of a node as a view of a content class, W2000 [Baresi06] introduces the concept of a navigation cluster that has no direct correspondence in other (including this) approaches. Such a navigation cluster represents an interaction context. It is a essentially a container that groups a set of closely related nodes. The different cluster types such as structural clusters, association clusters or collection clusters correspond to different viewpoints of the system. The overall navigation across the application is determined by shared nodes that belong to different clusters.

OOHDM [Schwaabe98] defines navigation classes which corresponds to navigation classes of this approach. A specialized form of database query language is used to define attributes of navigation classes and links. Additionally, the concept of navigation contexts is introduced which has no direct correspondence in other (including this) approaches, but is to some degree similar to navigation clusters in W2000. Navigation contexts allow the definition of an internal navigation structure for a set of instances of related navigation classes that fulfill a certain condition. Navigation contexts are also used for realizing access structures.

This approach is limited by the used expression language for the definition of navigation properties and guards of links, cf. 4.1.3 and the next section. The metamodel can easily be extended by new node types and new link types. For a further evolution it would be desirable to incorporate some of the complex navigation constructs from W2000 (navigation clusters) and OOHDM (navigation contexts), although these concepts take advantage of the corresponding proprietary notation and it is unclear if a satisfactory corresponding UML notation can be defined.

The navigation metamodel and the stepwise construction of the navigation model is presented in the following sections.

4.4.1 Metamodel

The basic elements in navigation models are nodes and links. The corresponding modeling elements in the metamodel are *Node* and *Link,* which are derived from the UML elements *Class* and *Association*, respectively. The backbone of the metamodel for navigation modeling is shown in Figure 32. The node metaclass is abstract, which means that only further specialized classes may be instantiated. Furthermore a node can be designated to be an entry point of the application with the *isHome* attribute. If a node should be reachable from everywhere within the navigation model without explicit links, then the *isLandmark* attribute has to be set. The *Link* class is also an abstract class. Links connect a source node with a target node as expressed by the two associations between link and node. With the *isAutomatic* attribute a link is automatically followed. Additionally, a guard expression can be defined, to specify when a link can be followed, see below. The element *Node* is further specialized to the concrete node type *NavigationClass*. Further specialized classes are used for modeling access structures (see below) and for the integration of processes (see 4.5.1). A navigation class is a navigational view of a content class, represented by the association to a content class. Navigation classes comprise a list of navigation properties which represent properties from the content model, expressed by the attribute *contentProperties*. An optional *derivationExpression* can be used to specify how the navigation property is de-

rived from the content properties. A navigation link is used for modeling the navigation between nodes with the usual semantics for hypermedia applications. Navigation links are always uni-directional. In case of bi-directional navigation links two uni-directional navigation links have to be used. Inheritance between navigation classes has the usual object-oriented semantics. In addition, navigation links to a super navigation class represent dynamic navigation, i.e. depending on the actual type of the target navigation class at runtime, the corresponding navigation class is presented to the user.

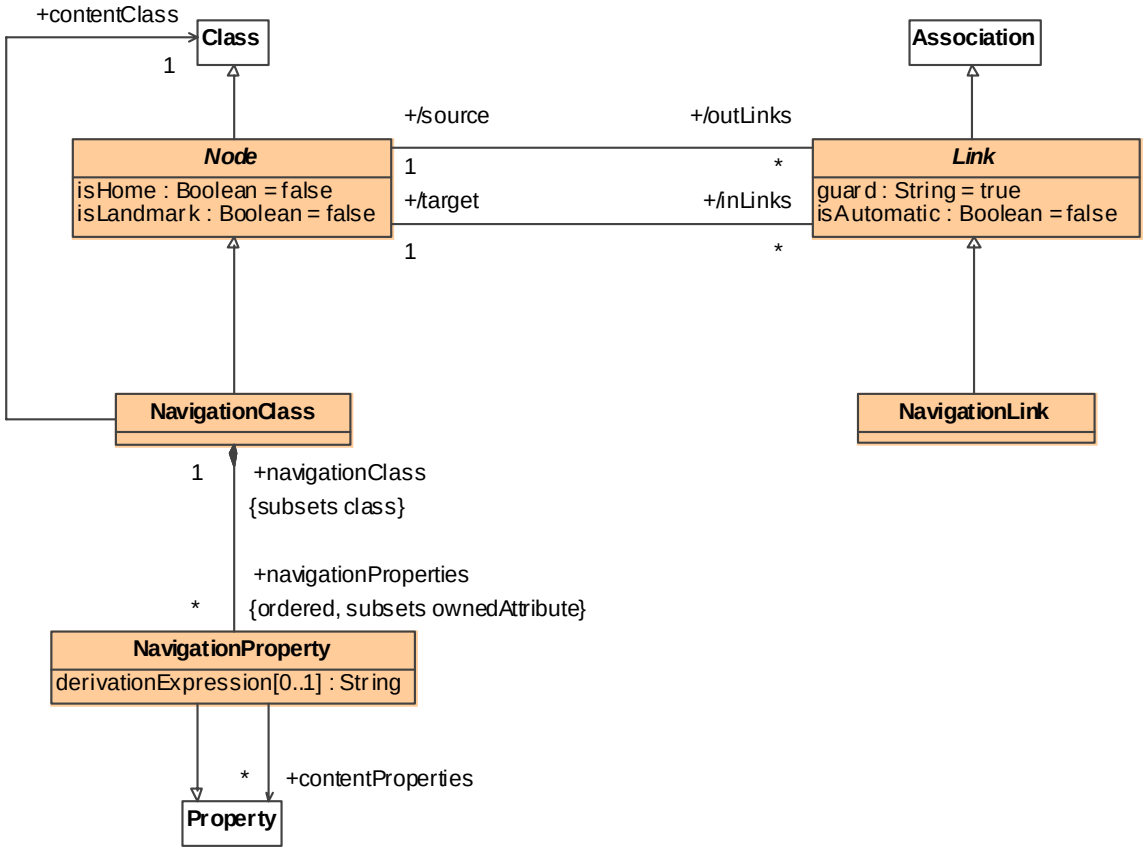


Figure 32. Metamodel for navigation modeling (backbone)

For further structuring the navigation model, two additional specialized node types are introduced: access primitives and menus, see Figure 33. Access primitives are used to define how collections of nodes should be accessed. Access primitives are further specialized to indices, guided tours and queries. An index represents the direct access to all instances of the target node type by providing the user with a list of all elements to choose from for continuing the navigation. A guided tour represents the sequential access to all elements. A sort expression can be defined with the attribute *sortExpression*. The query element represents the possibility to search for instances of the target node type where a filtering expres-

sion can be defined by the attribute *filterExpression*. Menus are specialized navigation classes that are used to structure the outgoing links from a navigation class. They have to be associated to a navigation class by a composition.

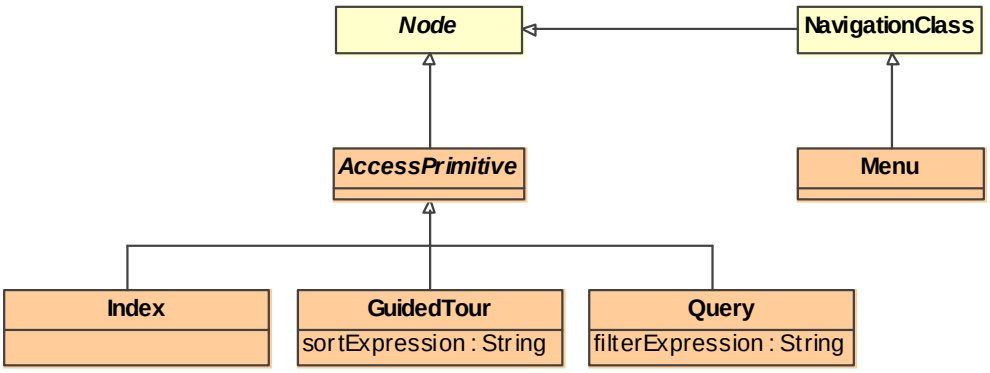


Figure 33. Metamodel for navigation modeling (access structures)

Derived Attributes

The attributes *source* and *target* of a link are derived from the members of the association super class.

```
context Link def : source : Node = self.ownedEnd->first().type
context Link def : target : Node = self.ownedEnd->first().opposite.type
```

The derived attributes *outLinks* and *inLinks* of a node are derived from the derived attributes *source* and *target* of the corresponding links.

```
context Node def : outLinks : Set( Link ) = Link.allInstances()->select( l | l.source = self )
context Node def : inLinks : Set( Link ) = Link.allInstances()->select( l | l.target = self )
```

Constraints

In a specific namespace at most one home node may be defined.

```
context Namespace inv NamespaceUniqueHomeNode :
    self.member->select( e | e.oclIsKindOf( Node ) )->
    select( n | n.isHome )->size() <= 1
```

Only navigation classes (and subclasses) can be home or landmark nodes.

```
context Node inv NodeHomeOrLandmark :
    self.isHome or self.isLandmark implies self.oclIsKindOf( NavigationClass )
```

Each node must be reachable. Therefore it has to be either a home or landmark node or the node (or a super node) must be navigable from some other node. Here, navigable means that some other node owns a corresponding attribute.

```
context Node inv NodeReachability :
    not ( self.isHome or self.isLandmark )  implies
    let allNodes : Set( Node ) = self.allParents()->including( self ) in
        Node.allInstances()->exists( n | n.ownedAttribute->exists( p | allNodes->includes( p.type ) ) )
```

Node inheritance is restricted to navigation classes. The content class associated to a navigation class has to conform to the corresponding content class of a super navigation class. Inheritance among different types of navigation classes is not permitted.

```
context Node inv NodeInheritance :
    if self.oclIsKindOf( NavigationClass ) then
        self.parents()->forAll( sn | sn.oclType = self.oclType and
            self.contentClass.conformsTo( sn.contentClass ) )
    else self.parents()->isEmpty() endif
```

A navigation class contains only navigation properties.

```
context NavigationClass inv NavigationClassOwnedAttributeType :
    self.ownedAttribute->forAll( p | p.oclIsKindOf( NavigationProperty ) )
```

The type of a navigation property has to be either a data type or a (navigation) node type.

```
context NavigationProperty inv NavigationPropertyType :
    self.type.oclIsKindOf( DataType ) or self.type.oclIsKindOf( Node )
```

Links only connect (navigation) nodes, must be binary and unidirectional.

```
context Link inv LinkMembers :
    self.memberEnd->size() = 2 and self.ownedEnd->size() = 1 and
    self.memberEnd->forAll( p | p.type.oclIsKindOf( Node ) )
```

All navigable properties of a node corresponding to the incoming links of an access primitive must have multiplicity one.

```
context AccessPrimitive inv AccessPrimitiveIncoming :
    let ps : Set( Property ) = Node.allInstances()->collect( n | n.ownedAttribute )->flatten()->
        select( p | p.association.oclIsKindOf( Link ) and p.type = self ) in
            ps->forAll( p | p.lower = 1 and p.upper = 1 )
```

An access primitive has exactly one outgoing link.

```
context AccessPrimitive inv AccessPrimitiveOutgoing :
    self.outLinks->size() = 1
```

The one and only outgoing link of an index leads to a navigation class and the corresponding navigable property has multiplicity many.

```
context Index inv IndexOutgoing :
```

```
self.ownedAttribute->forAll( p | p.association.oclIsKindOf( Link ) implies
    p.isMultivalued() and p.type.oclIsKindOf( NavigationClass ) )
```

The one and only outgoing link of a guided tour leads to a navigation class and the corresponding navigable property has multiplicity many.

```
context GuidedTour inv GuidedTourOutgoing :
  self.ownedAttribute->forAll( p | p.association.oclIsKindOf( Link ) implies
    p.isMultivalued() and p.type.oclIsKindOf( NavigationClass ) )
```

The one and only outgoing link of a query leads to an index and the corresponding navigable property has multiplicity one.

```
context Query inv QueryOutgoing :
  self.ownedAttribute->forAll( p | p.association.oclIsKindOf( Link ) implies
    p.lower = 1 and p.upper = 1 and p.type.oclIsKindOf( Index ) )
```

Derivation Expressions

If the derivation of a navigation property from content properties is non trivial, a derivation expression has to be defined as a simple expression of the expression language, see 4.1.3. The derivation of a navigation property is trivial, if it is derived directly from exactly one content property, for example the derivation expression

```
self.name
```

for the derivation of the navigation property *name* from the content property *name* of the content class *Project*. The derivation expression for a trivial derivation may be omitted. A non-trivial derivation expression for a navigation property representing the name of the corresponding validation project would be for example

```
empty self.validationProject ? “<none>” : self.validationProject.name
```

Guard Expressions

If the availability of a link should depend on some condition, a guard expression has to be defined as a simple expression of the expression language, see 4.1.3. The context of this expression is the source content class and the expression has to evaluate to a boolean value. The following guard expression for the link from a user project to the corresponding validation project

```
not empty self.validationProject
```

ensures that the link is only available if the validation project exists. Note that this guard condition could be omitted because a link is only shown when the target object exists. The

following guard expression shows the link only if the validation project exists and if it has a name:

```
empty self.validationProject ? false : not empty self.validationProject.name
```

Notation

Stereotyped UML class diagrams are used for navigation modeling. For a definition of the corresponding UML profile see A.4.

4.4.2 Navigation Space

The first step in the stepwise construction of the navigation model is called the navigation space model. The navigation space model specifies which nodes can be visited by direct navigation from other nodes. It is not yet specified how these nodes are accessed, which is done in the following steps by adding access structure elements. The navigation space model comprises navigation views of those content classes which can be visited by navigation through the Web application and navigation links (special kind of associations) that specify which navigation views can be reached through navigation. A navigation view of a content class may contain only a subset of the attributes of a content class or define additional attributes which are derived from the content class. In the following two sub sections, first the automatic derivation of the navigation space model from the requirements model and the content model is presented, followed by a description of the manual refinement activities.

4.4.2.1 Transformation RequirementsAndContent2Navigation

The transformation *RequirementsAndContent2Navigation* automatically generates an initial navigation model from the requirements and the content model. It comprises three transformation rules which are outlined below and detailed in B.3.4.

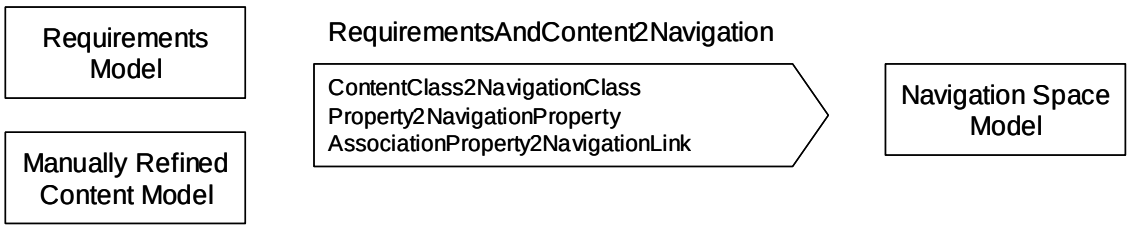


Figure 34. Transformation *RequirementsAndContent2Navigation*

The resulting navigation space model for the running example is depicted in Figure 35. For each class in the content model a corresponding navigation class was generated by the rule *ContentClass2NavigationClass*. The abstract content class *Project* was mapped to a corresponding abstract navigation class. Therefore, at runtime only instances of the sub navigation classes *UserProject* or *ValidationProject* may exist. The attributes of a project have been mapped to corresponding navigation properties by the rule *Property2NavigationProperty*. Finally, for each navigation use case in the requirements model and a corresponding association end in the content model a navigation link has been generated by the rule *AssociationProperty2NavigationLink*, such as for example the navigation link from the project manager to a project. This navigation link represents dynamic navigation, i.e. depending on the actual type of the project at runtime, either a user project or a validation project is presented to the user.

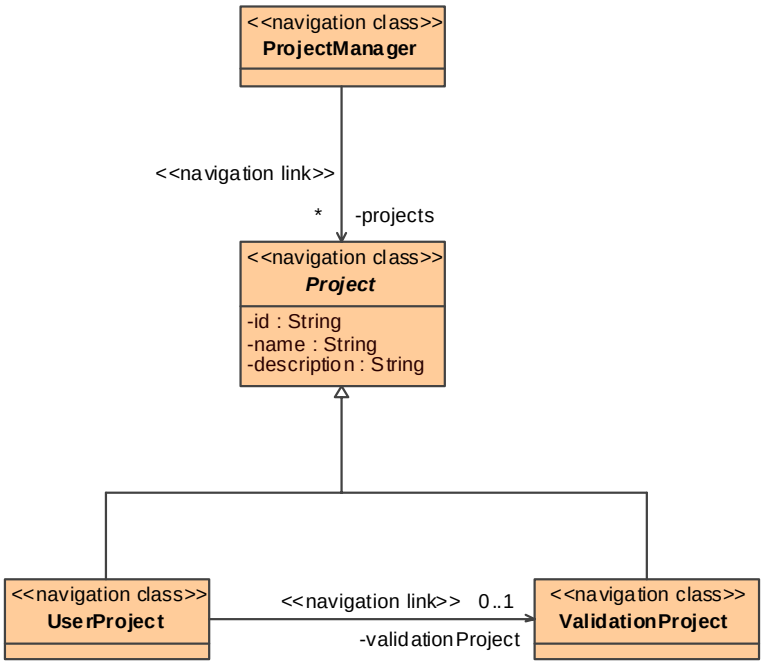


Figure 35. Navigation space model derived by transformation *RequirementsAndContent2Navigation*

Rule ContentClass2NavigationClass

Each content class that is either subject of a Web use case or target of a Web use case, as returned by the helper *isRelevantForNavigation*, is mapped to a navigation class with the same name. Additionally, a reference to the corresponding content class is assigned. In comparison to the rule implementation in B.3.4 the following details have been omitted below: target bindings that are not relevant for understanding the rule, and targets for mapping inheritance between content classes to inheritance between navigation classes.

```
rule ContentClass2NavigationClass
{
    from c : UWE!Class ( c.isRelevantForNavigation() )
    to tc : UWE!Class ( … ), -- target for copying source element
    nc : UWE!NavigationClass
    (
        name <- c.name,
        contentClass <- tc,
        ownedAttribute <- c.ownedAttribute->collect( p | thisModule.resolveTemp( p, 'np' ) ),
        …
    ),
    …
```

```
}
```

Rule Property2NavigationProperty

Each content property that is owned by a content class and not part of an association is mapped to a navigation property. In comparison to the rule implementation in B.3.4 target bindings that are not relevant for understanding the rule have been omitted.

```
rule Property2NavigationProperty
{
    from p : UWE!Property ( p.oclIsTypeOf( UWE!Property ) and
            p.class_.isRelevantForNavigation() and p.association.oclIsUndefined() )
    to tp : UWE!Property ( … ), -- target for copying source element
    np : UWE!NavigationProperty
    (
        name <- p.name,
        class_ <- p.class_,
        type <- p.type,
        contentProperties <- Sequence { p },
        …
    )
}
```

Rule AssociationProperty2NavigationLink

Each property of an association that is owned by a content class (i.e. each navigable association end) is mapped to a navigation link and two corresponding navigation properties. An additional condition is that a corresponding navigation use case exists in the requirements model. Unidirectional associations are mapped to one navigation link and bi-directional associations to two navigation links. In comparison to the rule implementation in B.3.4 target bindings that are not relevant for understanding the rule have been omitted.

```
rule AssociationProperty2NavigationLink
{
    from p : UWE!Property (
            if p.class_.oclIsTypeOf( UWE!Class ) then
                p.oclIsTypeOf( UWE!Property ) and
                not p.association.oclIsUndefined() and
                p.class_.useCase->exists( uc |
                    uc.oclIsKindOf( UWE!Navigation ) and uc.target() = p.type )
            else false endif )
    to tp : UWE!Property ( … ), -- target for copying source element
```

```
    nl : UWE!NavigationLink
    (
        …
    ),
    nps : UWE!Property
    (
        association <- nl,
        owningAssociation <- nl,
        type <- p.class_,
        …
    ),
    np : UWE!NavigationProperty
    (
        name <- p.name,
        class_ <- p.class_,
        type <- p.type,
        association <- nl,
        contentProperties <- Set { p },
        …
    )
}
```

4.4.2.2 Manual Refinement

The automatically derived navigation space model has then to be refined manually. The only required activity is the designation of a home node for the application by setting the *isHome* attribute. Other optional activities are:

- Definition of additional navigation classes
- Definition of additional navigation properties
- Definition of additional navigation links
- Renaming of automatically derived model elements
- Deletion of automatically derived model elements
- Designation of landmark nodes
- Designation of automatic links
- Definition of guard expressions for links

The automatically derived navigation space model for the running example was manually refined, resulting in the navigation model depicted in Figure 36. First, the navigation class *ProjectManager* was designated as entry point of the Web application by setting the *isHome* attribute. Second, a back navigation link was added to allow the user to navigate from a project back to the project manager.

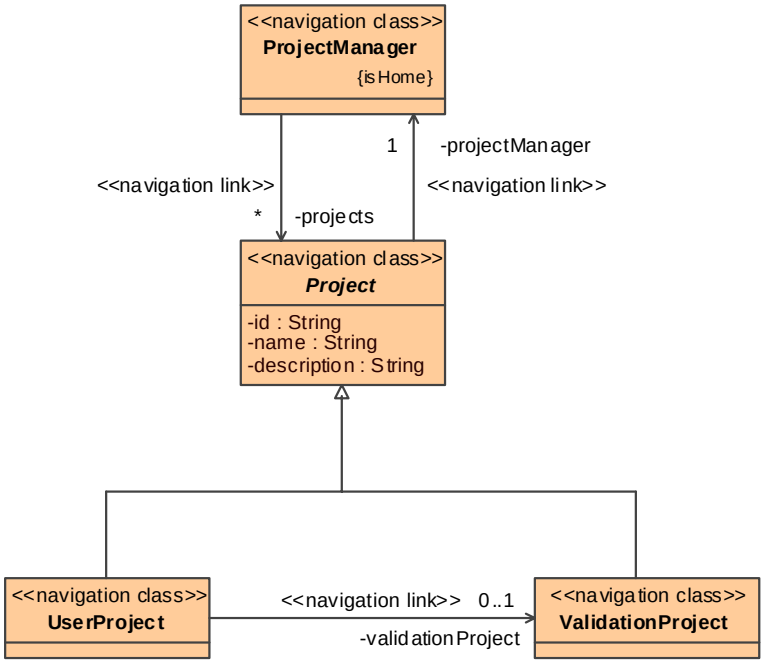


Figure 36. Manually refined navigation space model

4.4.3 Addition of Indices

After the construction of the navigation space model in which navigation classes and navigation links were defined that span the navigation space, access structures have to be added to the navigation model, in order to define how the access to the targets of navigation links with multi-valued end should be realized. Therefore, indices are automatically added to the navigation model. The resulting navigation model can then optionally be refined.

4.4.3.1 Transformation AddIndices

The transformation *AddIndices* depicted in Figure 101 adds indices to the navigation model, which is outlined in this section. It comprises one transformation rule which is outlined below and detailed in B.3.5.

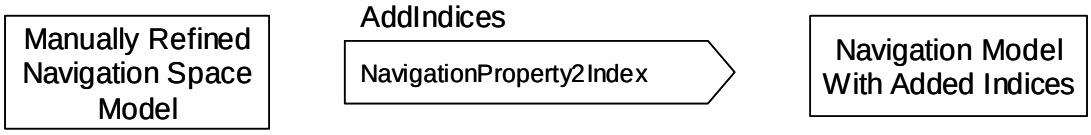


Figure 37. Transformation *AddIndices*

In Figure 38 the application of this transformation to the running example is depicted. The multi-valued navigation property belonging to the link from the project manager to projects was transformed to a link to an index and a corresponding link to the original target of the link.

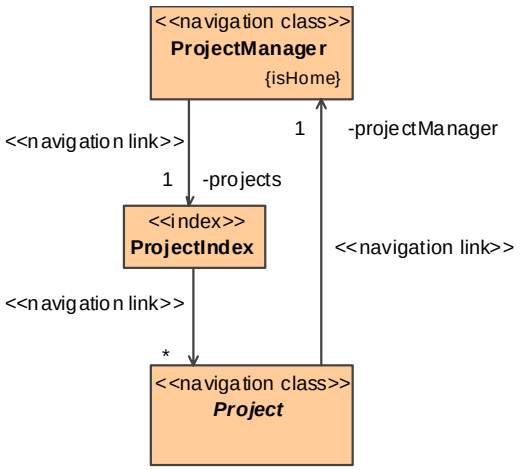


Figure 38. Navigation model with added indices derived by transformation *AddIndices*

Rule NavigationProperty2Index

The rule *NavigationProperty2Index* adds indices to the navigation model. This is done by matching all multi-valued navigation properties that are ends of a link where the source is a navigation class and the target is a navigation class (exact type). Such a navigation property is changed to point to a generated index element. Additionally, an outgoing link from this index to the original target navigation class is generated. Note that apart from the nodes and links always the corresponding properties have to be created, too. In comparison to the rule implementation in B.3.5 target bindings that are not relevant for understanding the rule have been omitted.

```
rule NavigationProperty2Index
{
```

```
    from np : UWE!NavigationProperty ( np.isMultivalued() and
        np.association.oclIsKindOf( UWE!Link ) and
        np.class_.oclIsKindOf( UWE!NavigationClass ) and
        np.type.oclIsKindOf( UWE!NavigationClass ) )
    to tnp : UWE!NavigationProperty
    (
        name <- np.name,
        class_ <- np.class_,
        type <- index,
        association <- np.association,
        derivationExpression <- np.derivationExpression,
        contentProperties <- np.contentProperties
    ),
    index : UWE!Index
    (
        name <- if UWE!Property.allInstances()->select( p |
                p.isMultivalued() and p.association.oclIsKindOf( UWE!Link ) and p.type = np.type )
            ->size() > 1 then np.class_.name else '' endif + np.type.name + 'Index',
        ownedAttribute <- Sequence { npt }
    ),
    nl : UWE!NavigationLink
    (
        owner <- np.class_.owner
    ),
    nps : UWE!Property
    (
        association <- nl,
        owningAssociation <- nl,
        type <- index
    ),
    npt : UWE!Property
    (
        association <- nl,
        class_ <- index,
        type <- np.type
    )
}
```

4.4.3.2 Manual Refinement

The refinement of the automatically derived navigation model with indices is optional. The following activities are possible:

- Definition of additional access primitives

- Deletion of automatically derived indices
- Renaming of automatically derived indices
- Replacement of navigation links with composite associations

For the running example it was chosen to replace the navigation link from the project manager to the project index with a composite association, in order to reduce the number of navigation links.

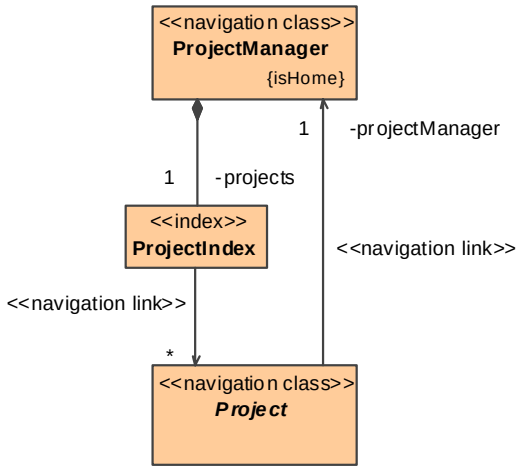


Figure 39. Navigation model with refined indices

4.4.4 Addition of Menus

After the addition of indices as described in the last section, menus are added to the navigation model to organize the outgoing links of navigation classes. A transformation automatically adds a menu to each navigation class with outgoing links. The resulting navigation model can then be manually refined optionally.

4.4.4.1 Transformation AddMenus

The transformation *AddMenus* depicted in Figure 40 automatically adds menus to the navigation model. It comprises two transformation rules which are outlined below and detailed in B.3.6.

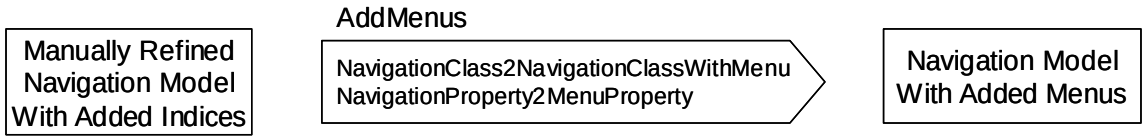


Figure 40. Transformation *AddMenus*

The automatically generated menus for the project and the user project navigation classes are depicted in Figure 41. The inheritance relationship between a user project and a project was mapped to an inheritance relationship between the corresponding menus. Thus, the user project menu inherits the navigation link to the project manager from the project menu.

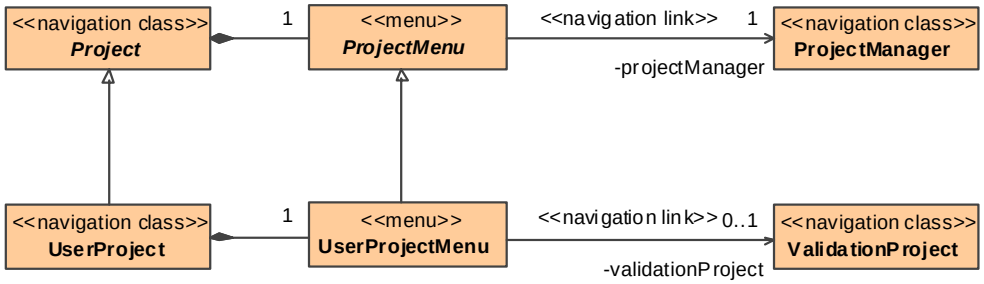


Figure 41. Navigation model with added menus derived by transformation *AddMenus*

Rule NavigationClass2NavigationClassWithMenu

The rule *NavigationClass2NavigationClassWithMenu* creates a menu for each navigation class with at least one outgoing link, or if for the corresponding content class at least one Web process use case is defined, because this menu is then required for the integration of processes as presented in 4.5.1. All outgoing links are moved to the menu node and the menu node is composed with the original navigation class. In comparison to the rule implementation in B.3.6 the following details have been omitted below: target bindings that are not relevant for understanding the rule, and targets for mapping inheritance between navigation classes to inheritance between menus.

```
rule NavigationClass2NavigationClassWithMenu
{
    from nc : UWE!NavigationClass ( nc.oclIsTypeOf( UWE!NavigationClass ) and
        ( nc.ownedAttribute->select( p |
            not p.isComposite and p.association.oclIsKindOf( UWE!Link )
            and not p.type.oclIsTypeOf( UWE!Menu ) )->size() > 0
```

```
        or nc.contentClass.useCase->exists( uc | uc.oclIsKindOf( UWE!WebProcess ) ) )
    using
    {
        menuNps : Sequence( UWE!Property ) = nc.ownedAttribute->select( p | not p.isComposite
            and p.association.oclIsKindOf( UWE!Link ) and not p.type.oclIsTypeOf( UWE!Menu ) );
        otherNps : Sequence( UWE!Property ) = nc.ownedAttribute - menuNps;
    }
    to tnc : UWE!NavigationClass
    (
        ownedAttribute <- otherNps->including( apt ),
        …
    ),
    menu : UWE!Menu
    (
        name <- nc.name + 'Menu',
        ownedAttribute <- menuNps,
        contentClass <- nc.contentClass,
        …
    ),
    a : UWE!Association
    (
        …
    ),
    aps : UWE!Property
    (
        association <- a,
        owningAssociation <- a,
        type <- nc,
        …
    ),
    apt : UWE!NavigationProperty
    (
        association <- a,
        class_ <- nc,
        type <- menu,
        aggregation <- #composite,
        isComposite <- true,
        …
    ),
    …
}
```

Rule NavigationProperty2MenuProperty

This rule converts all properties of a navigation class (exact type) which are part of a link to the corresponding properties of the links from the menu generated by the rule *NavigationClass2NavigationClassWithMenu*. In comparison to the rule implementation in B.3.6 target bindings that are not relevant for understanding the rule have been omitted.

```
rule NavigationProperty2MenuProperty
{
    from np : UWE!NavigationProperty ( np.class_.oclIsTypeOf( UWE!NavigationClass ) and
        np.type.oclIsKindOf( UWE!Node ) and not np.type.oclIsTypeOf( UWE!Menu ) and
        not np.isComposite and np.association.oclIsKindOf( UWE!Link ) )
    to tnp : UWE!NavigationProperty
    (
        class_ <- thisModule.resolveTemp( np.class_, 'menu' ),
        …
    )
}
```

4.4.4.2 Manual Refinement

The automatically derived navigation model with added menus can optionally be refined by the following activities:

- Definition of new menus in order to further structure outgoing links
- Renaming of menus

The automatically derived navigation model for the running example has not been manually refined.

4.5 Process

The navigation model of a Web application represents the static information structure accessible to a user of the system. Processes on the other hand represent the dynamic aspects of a Web application.

Process modeling (also called task modeling) stems from the Human Computer Interaction (HCI) field [Harmelen01]. A process is composed of one or more sub processes and/or ac-

tions that a user may perform to achieve a goal. A goal represents a desired change in the state of the system and may be realized by formulating a plan composed of processes and then performing those processes. Here the concept process is considered in a broader sense by taking into account actions performed by the system and actions performed by the user, see 4.5.2.

Different UML notations have been proposed for process modeling. Wisdom is a UML extension that proposes the use of a set of stereotyped classes that make the notation not very intuitive [Nunes00]. Markopoulos et al. make two different proposals: a UML extension of use cases [Markopoulos00] and another one based on statecharts and activity diagrams [Markopoulos02]. As already sketched in previous works of the author process modeling as proposed here is based on UML activities [Koch03a], [Koch04a]. Activities in general can be considered as "roadmaps" of system functional behavior [Lieberman01], or, especially for Web applications we may speak of "roadmaps" of user interaction with the system. For the case that the content model is implemented by Web services the process model represents a choreography of Web services to achieve a desired behavior.

In contrast to other Web methodologies which realize processes with nodes and links as part of the navigation model, such as for example OO-H [Cachero02] or WebML [Ceri02], in this work processes are treated as an additional concern and are represented by a full-fledged model. Processes in OOWS [Fons03] are captured in the business process model using an extended version of the Business Process Modeling Notation (BPMN) [OMG06c] and a corresponding extended metamodel for business modeling. BPMN stems from the B2B (business-to-business) field and is similar (but not identical) to UML activities. In contrast to this work also manual tasks are considered in the process model, i.e. tasks that are manually carried out by humans and not automatically by the system by invoking operations or Web services. However, processes in OOWS are not represented by a dedicated model, but distributed over the BPMN process model and the navigation model. The latter contains a lower level view of the process model, where the constructs of the process model are resolved into navigation constructs. W2000 [Baresi06] follows a similar approach for process modeling as presented here although operations are defined separately from the content model. It is suggested that either activity diagrams or collaboration diagrams are used to define the workflow of processes, but it is left unclear how these processes can be executed or translated to code. In contrast, the approach of this work allows a detailed specification of workflows by using UML 2 activities and it is clearly defined how processes are integrated in the navigation model.

The expressiveness of process modeling presented here is limited by the subset of the UML modeling elements applicable for activities which are currently supported (see

4.5.2.1). Therefore, this approach can be extended to support all the remaining UML modeling elements such as for example exceptions, events or structured activity nodes. Additionally, specialized action types could be supported, for example for the direct manipulation of objects, such as reading or writing attributes or associations. Although this type of actions is considered in the UML metamodel, no notation is given and tool support does not exist, and therefore these types of actions are not considered in this work.

Process modeling comprises three parts which are presented in the following sections: process integration for integrating the invocation of processes in the navigation model, process data representing data accessed by processes and process flow representing the dynamic process flow itself which comprises the invocation of operations from the content model. Process data and process flow are developed concurrently, hence they are presented together. Model transformations for successively deriving the process model from the underlying models are given in each section.

4.5.1 Process Integration

In order to invoke dynamic behavior an interface between processes and navigation is needed. This is achieved by integrating the invocation of processes in the navigation model by means of process classes and process links which are derived from the corresponding Web process use cases in the requirements model.

4.5.1.1 Metamodel

The modeling elements relevant for process integration are depicted in Figure 42. The two basic constructs from the navigation metamodel node and link are specialized by introducing the modeling elements process class and process link, respectively.

Each process is represented by a process activity (see 4.5.2) and a process class that is associated to the corresponding Web process use case. Only the latter is relevant for process integration. In general process classes represent data that is used during execution of a process. For each process one process class is designated for integration in the navigation model. Process links are special links used for the invocation of a process. Either the source or the target (but not both) of a process link must be a process class. Following a process link to a process class starts the execution of the corresponding process activity. The input parameter of the activity must be compatible with the source content class. A process link from a process class is automatically followed upon completion of the corresponding process activity. The output parameter of the process activity must be compatible with the target content class. If a process class has no outgoing process links then the navigation context is not changed on invocation of the process.

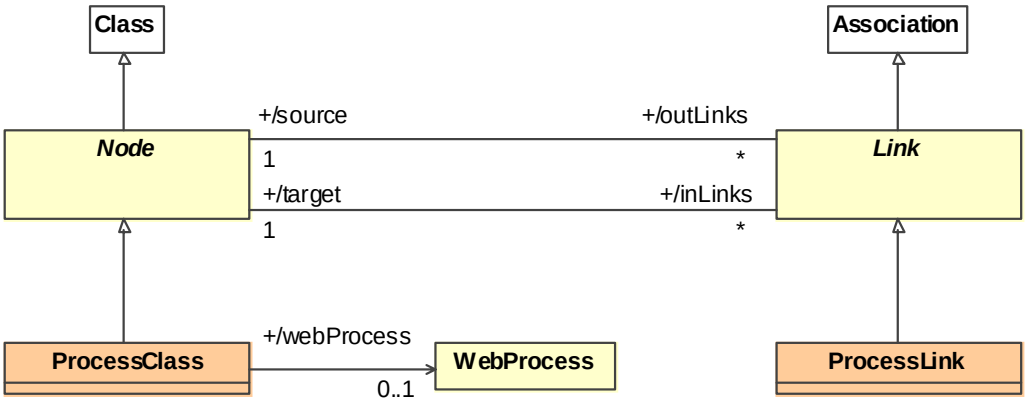


Figure 42. Metamodel for integration of processes in the navigation model

Derived Attributes

The derived attribute *webProcess* of a process class is defined as being the associated Web process use case. This attribute is only defined for the process class representing the process.

```
context ProcessClass def : webProcess : WebProcess =
    if self.inLinks->isEmpty() then OclUndefined else
    self.useCase->any( uc | uc.oclIsKindOf( WebProcess ) ) endif
```

Constraints

All ingoing and outgoing links of a process class must be process links.

```
context ProcessClass inv ProcessClassLinkTypes :
    self.inLinks()->forAll( pl | pl.oclIsTypeOf( ProcessLink ) ) and
    self.outLinks()->forAll( pl | pl.oclIsTypeOf( ProcessLink ) )
```

A process class can have at most one incoming process link and at most one outgoing process link.

```
context ProcessClass inv ProcessClassLinkCount :
    self.inLinks->size() <= 1 and self.outLinks->size() <= 1
```

Every process class that is reachable by following a process link must be associated to exactly one Web process use case.

```
context ProcessClass inv ProcessClassWebProcess :
    self.inLinks->notEmpty() implies
        self.useCase->one( uc | uc.oclIsKindOf( WebProcess ) )
```

One end of process link must be a process class and the other end must be a navigation class.

```
context ProcessLink inv ProcessLinkEnds :
    self.source.oclIsKindOf( NavigationClass ) and self.target.oclIsTypeOf( ProcessClass ) or
    self.source.oclIsTypeOf( ProcessClass ) and self.target.oclIsKindOf( NavigationClass )
```

Notation

The same notation as for navigation models, i.e. stereotyped UML class diagrams, is used for modeling the process integration. For a definition of the corresponding UML profile see A.5.

4.5.1.2 Tranformation ProcessIntegration

The transformation *ProcessIntegration* depicted in Figure 43 enhances the navigation model by adding process classes and process links for the integration of processes. The transformation comprises one transformation rule which is outlined below and detailed in B.3.7.

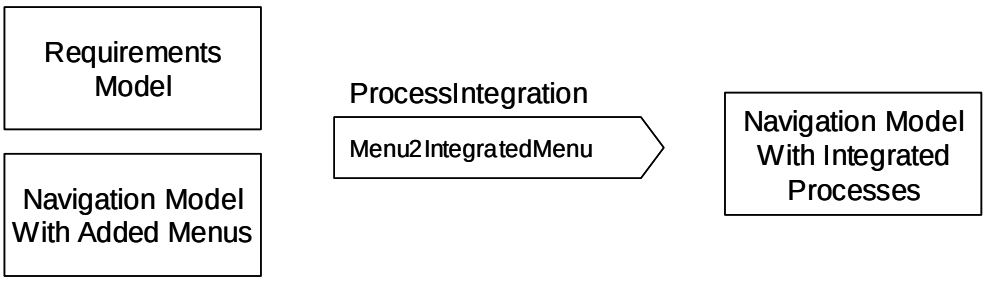


Figure 43. Transformation *ProcessIntegration*

The automatic integration of the two processes *AddProject* and *RemoveProject* of the project manager content class is depicted in Figure 44. For the former an additional exit link was generated, because a target was defined for the corresponding Web process use case depicted in Figure 26.

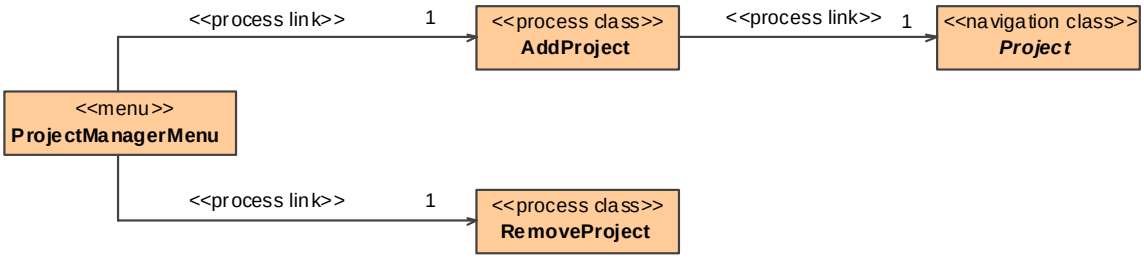


Figure 44. Integrated navigation model for content class *ProjectManager* derived by transformation *ProcessIntegration*

Rule Menu2IntegratedMenu

Each Web process use case from the requirements model is mapped to a process class in the integrated navigation model. A process link connects the menu corresponding to the content class of the Web process use case to the generated process class. An outgoing process link is generated for the optional target of the Web process use case. In comparison to the rule implementation in B.3.7 the following details have been omitted below: target bindings that are not relevant for understanding the rule, and targets for the ends of the process links.

```
rule Menu2IntegratedMenu
{
    from nc : UWE!Menu (
        nc.contentClass.useCase->exists( uc | uc.oclIsKindOf( UWE!WebProcess ) ) )
    using
    {
        wps : Sequence( UWE!WebProcess ) = nc.contentClass.useCase->
            select( uc | uc.oclIsKindOf( UWE!WebProcess ) )->asSequence();
        wpsWithTarget : Sequence( UWE!WebProcess ) = wps->select( wp |
            not wp.target.oclIsUndefined() );
        wpsWithoutTarget : Sequence( UWE!WebProcess ) = wps->select( wp |
            wp.target.oclIsUndefined() );
        wpsOrdered : Sequence( UWE!WebProcess ) = wpsWithTarget->union( wpsWithoutTarget );
    }
    to tnc : UWE!Menu ( … ), -- target for copying source element
    pc : distinct UWE!ProcessClass foreach ( wp in wpsOrdered )
    (
        name <- let n : String = wp.name.regexReplaceAll( ' ', '' ).firstToUpper() in
            if UWE!WebProcess.allInstances()->select( uc |
                uc.name.regexReplaceAll( ' ', '' ).firstToUpper() = n )->size() > 2 then
                    nc.getTraceSource( 'NavigationClass2Menu' ).name else '' endif + n,
        …
```

```
    ),
    pl : distinct UWE!ProcessLink foreach ( wp in wpsOrdered )
    (
        …
    ),
    epl : distinct UWE!ProcessLink foreach ( wp in wpsWithTarget )
    (
        …
    ),
    …
}
```

4.5.1.3 Manual Refinement

The manual refinement of the automatically derived process classes and links comprises the definition of guards for the entry process link, in order to specify under which conditions a process can be executed. For details about the definition of guards see 4.4.1. As depicted in Figure 45 a guard for the process entry link to the process *RemoveProject* has been defined to ensure that the collection of projects is not empty.

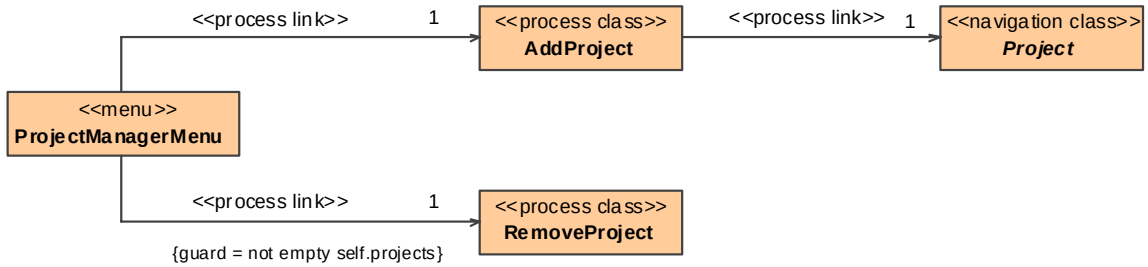


Figure 45. Manually refined process classes and links for content class *ProjectManager*

4.5.2 Process Data and Flow

The behavior of a Web process is defined by the process flow model. The process data model defines the data required for the execution of the process flow model. The process data and the process flow model are usually developed concurrently and are hence addressed together in this section.

UML activities are used for process flow modeling, see [OMG05a] for a description of syntax and semantics of activities. An activity is the specification of parameterized behavior as the coordinated sequencing of subordinate units. The flow of execution is represented by activity nodes connected by activity edges. Control nodes provide flow-of-

control constructs, such as decisions and synchronization. Object nodes represent data flowing along object flow edges. An action node represents executable behavior. Special actions can be used to invoke other activities, thus activities can be composed from reusable units. Call operation actions represent the invocation of operations. The semantic of activities is based on control and data token flows, similar to Petri nets [Priese03].

At runtime an activity has access to the features of its context object and any objects linked to the context object, transitively. The context object of a Web process activity is the corresponding process class. The parameters of the activity must correspond to the types of the corresponding content classes of the navigation classes in the navigation model connected to the Web activity by process links. Only special nodes are allowed here for the process flow model, as the modeling constructs for activities provided by the UML are too complex to be transformed in a generic way to platform specific constructs.

4.5.2.1 Metamodel

Process classes are used for process data modeling as depicted in Figure 46. Process properties, i.e. attributes of a process data class, capture the user input. A content class may be defined for a process class for the definition of a context for the input data represented by the process properties. If a content class is defined then an edit property may be defined for a process property with the impact that on the one hand the initial value shown to the user is determined from the edit property. On the other hand, changes to the process property are forwarded to the edit property. The attribute *rangeExpression* can be used to define a simple expression for the range of values a process property can receive. If a content class is defined for the corresponding process class, then this expression may reference the specified content class.

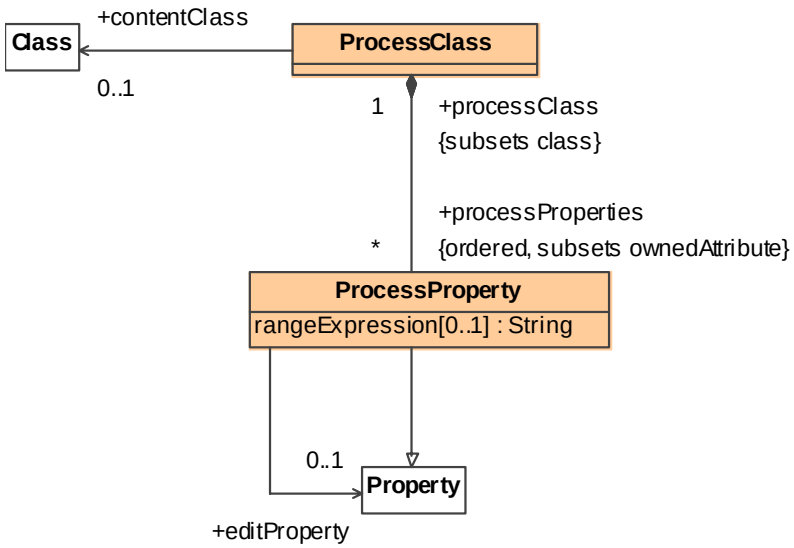


Figure 46. Metamodel for the process data modeling

For process flow modeling only a small extension to UML activity elements has been made as depicted in Figure 47. A process flow is modeled with a special process activity which has an association to the Web process use case from the requirements model on the one hand and to the process class representing the process on the other hand. Further, a special user action is used for modeling interactions with the user of the Web application. The effect of this action is to present the corresponding process data class to the user (cf. 4.6). He or she can enter data corresponding to the process properties of the process class, and when the user has finished entering data, this data is available at the output pins of the action. Each output pin corresponds to one process property of the process class. If a content class is specified for a process class corresponding to a user action then the user action must have an input pin with a type conforming to the type of the content class. Figure 47 also shows the supported modeling elements for activities from the UML metamodel (model elements with white background). Figure 48 and Figure 49 show the supported control node and object node types, respectively.

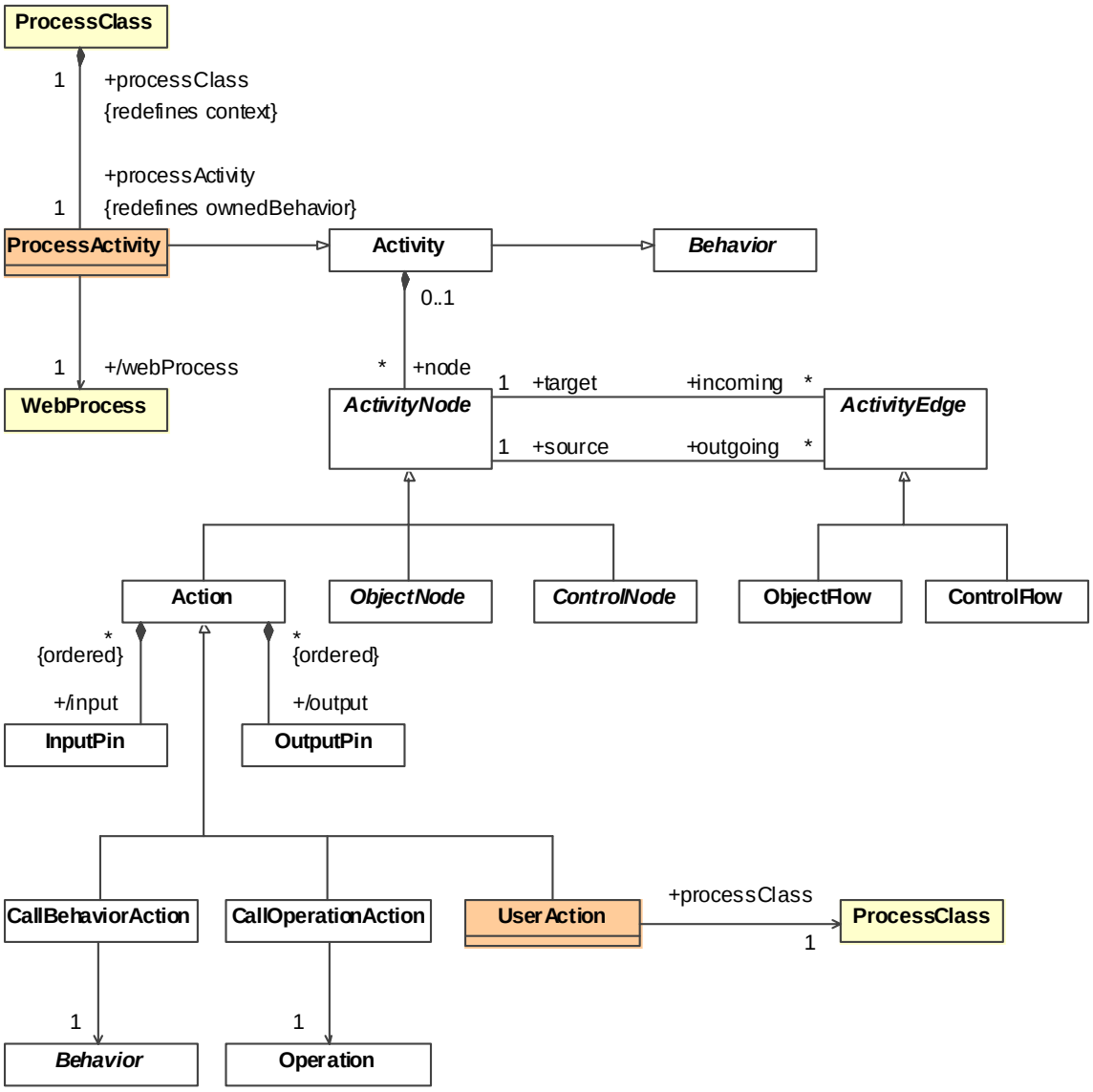


Figure 47. Metamodel for the process flow modeling

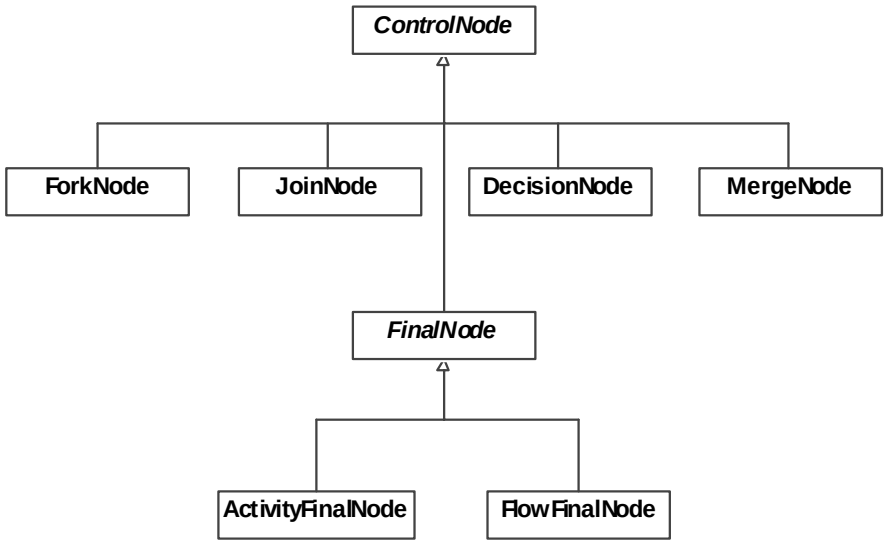


Figure 48. UML control nodes

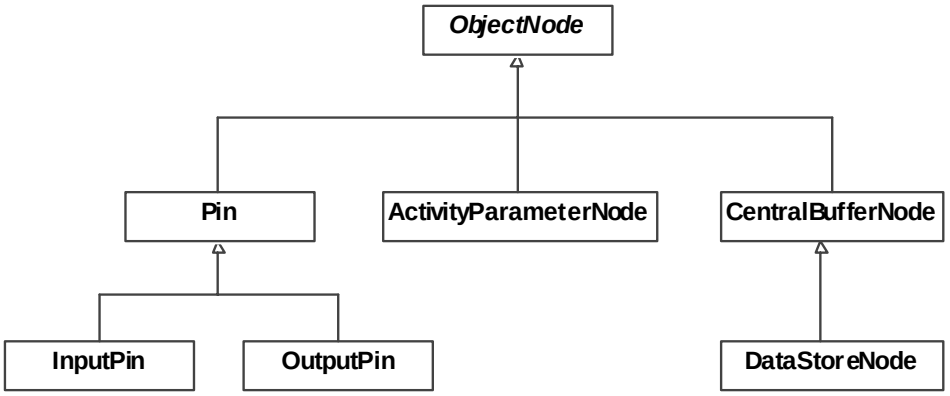


Figure 49. UML object nodes

Derived Attributes

The derived attribute *webProcess* of a process activity refers to the *webProcess* attribute of the corresponding process class representing the process.

```
context ProcessActivity def : webProcess : WebProcess = self.processClass.webProcess
```

Constraints

The type of a process property must be either a data type, i.e. a primitive type or an enumeration type, or a content class.

```
context ProcessProperty inv ProcessPropertyType :
    self.type.oclIsKindOf( DataType ) or self.type.oclIsTypeOf( Class )
```

If an edit property is defined for a process property then a content class has to be assigned to the corresponding process class and the edit property has to be one of the owned properties of this content class or one of its super classes.

```
context ProcessProperty inv ProcessPropertyEditProperty :
    self.editProperty->notEmpty() implies
        self.processClass.contentClass.allParents()->including( self.processClass.contentClass )->
            collect( c | c.ownedAttribute )->flatten()->includes( self.editProperty )
```

The designated process class for a process activity must have an incoming link.

```
context ProcessActivity inv ProcessActivityProcessClass :
    self.processClass.inLinks->notEmpty()
```

A process activity must have exactly one input parameter and at most one output parameter. This implies that it must have exactly one input activity parameter node and at most one output activity parameter node.

```
context ProcessActivity inv ProcessActivityParameter :
    self.parameter->select( p | p.direction = #in )->size() = 1 and
    self.parameter->select( p | p.direction = #out )->size() <= 1 and
    self.parameter->select( p | p.direction = #inout )->size() = 0 and
    self.parameter->select( p | p.direction = #return )->size() = 0
```

The content class of the source navigation class of the process link connecting to the process class of the process activity has to conform to the type of the input parameter of a process activity.

```
context ProcessActivity inv ProcessActivityInputParameter :
    let source : NavigationClass = self.processClass.inLinks->any().source in
    let inputParameter : Parameter = self.parameter->select( p | p.direction = #in )->first() in
    source.contentClass.conformsTo( inputParameter.type )
```

If the process class of a process activity has an outgoing link, then the output parameter has to conform to the content class of the target of this outgoing link. Additionally, the process activity must not have an activity final node.

```
context ProcessActivity inv ProcessActivityOutputParameter :
    self.processClass.outLinks->size() = 1 implies
        self.parameter->select( p | p.direction = #out )->size() = 1 and
        not self.node->exists( n | n.oclIsKindOf( ActivityFinalNode ) ) and
        let target : NavigationClass = self.processClass.outLinks->any().target in
        let outputParameter : Parameter = self.parameter->select( p | p.direction = #out )->first() in
            outputParameter.type.conformsTo( target.contentClass )
```

If the process class of a process activity does not have an outgoing link, then the process activity must have an activity final node.

```
context ProcessActivity inv ProcessActivityFinalNode :
    self.processClass.outLinks->isEmpty() implies
        self.node->exists( n | n.oclIsKindOf( ActivityFinalNode ) )
```

A process activity must not have an initial node.

```
context ProcessActivity inv ProcessActivityInitialNode :
    not self.node->exists( n | n.oclIsKindOf( InitialNode ) )
```

If the process class of a user action is associated to a content class then exactly one input pin has to be defined for the user action and its type has to conform to the content class. Otherwise, no input pin must be defined and at least one incoming control flow has to enter the user action.

```
context UserAction inv UserActionInput :
    if self.processClass.contentClass->isEmpty() then
        self.input->isEmpty() and self.incoming->notEmpty()
    else
        self.input->size() = 1 and self.input->forAll( pin |
            pin.type.conformsTo( self.processClass.contentClass ) )
    endif
```

For each output pin of a user action a process property of the associated process class has to exist with its name matching the name of the output pin and its type conforming to the type of the output pin.

```
context UserAction inv UserActionOutput :
    self.output->forAll( pin | self.processClass.processProperties->exists( p |
        p.name = pin.name and p.type.conformsTo( pin.type ) ) )
```

Notation

Stereotyped UML class diagrams are used for modeling the process data and stereotyped UML activity diagrams are used for modeling the process flow. For a definition of the corresponding UML profile see A.5.

4.5.2.2 Transformation CreateProcessDataAndFlow

The transformation *CreateProcessDataAndFlow* depicted in Figure 51 automatically generates the process data and the process flow for all Web process use cases from the requirements model. The data of a process is captured by process data classes which are a composite part of the designated process class in the navigation model with integrated processes presented in the last section. The flow of a process is represented by a process activity which is owned by the designated process class in the navigation model with integrated processes. For simple processes the corresponding operations in the content model

are mapped to call operation actions in the process flow model. The transformation comprises three transformation rules which are outlined below and detailed in B.3.8.

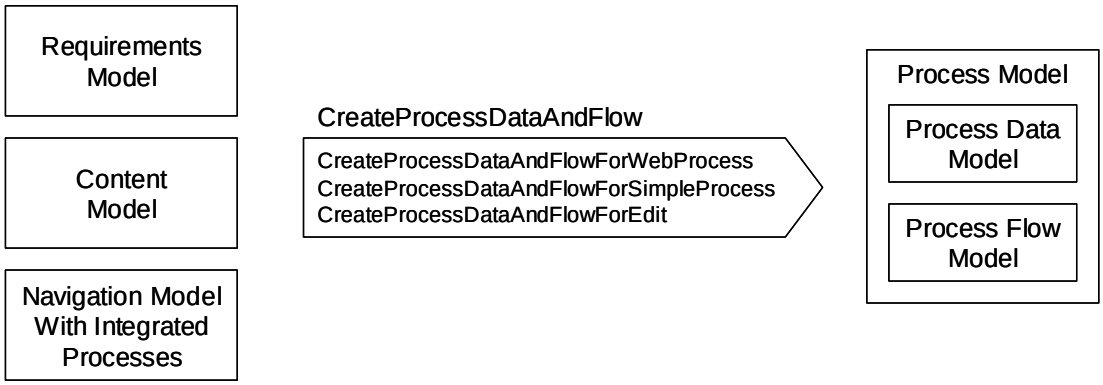


Figure 50. Transformation *CreateProcessDataAndFlow*

Rule CreateProcessDataAndFlowForWebProcess

For general Web processes, i.e. neither edit processes nor simple processes, only the parameters and the activity parameter nodes of the corresponding process activity are generated by the rule *CreateProcessDataAndFlowForWebProcess*, as in the case of the Web process *AddProject* of the running example, see Figure 51. The resulting process flow is thus incomplete and has to be refined by the developer as presented in the next section. If the Web process does not have an exit link then an activity final node is created instead of the output activity parameter node. The local variables *targetSeq* and *nTargetSeq* are defined to simulate the conditional creation of target elements using iterative target pattern elements, see [ATL06a]. In comparison to the rule implementation in B.3.8 the following details have been omitted below: target bindings that are not relevant for understanding the rule, and targets for the parameters of the process activity.

Figure 51. Incomplete process flow for web process *AddProject* derived by rule *CreateProcessDataAndFlowForWebProcess*

```
rule CreateProcessDataAndFlowForWebProcess
{
    from pc : UWE!ProcessClass ( pc.ownedBehavior->isEmpty() and
        pc.webProcess.oclIsTypeOf( UWE!WebProcess ) )
    using
```

```
    {
        source : UWE!NavigationClass = let ls : Set( UWE!Link ) = pc.inLinks in
            if ls->isEmpty() then OclUndefined else ls->any().source endif;
        target : UWE!NavigationClass = let ls : Set( UWE!Link ) = pc.outLinks in
            if ls->isEmpty() then OclUndefined else ls->any().target endif;
        targetSeq : Sequence( Boolean ) = if target.oclIsUndefined() then Sequence {}
            else Sequence { true } endif;
        nTargetSeq : Sequence( Boolean ) = if target.oclIsUndefined() then Sequence { true }
            else Sequence {} endif;
    }
    to tpc : UWE!ProcessClass
    (
        ownedBehavior <- Sequence { pa },
        …
    ),
    pa : UWE!ProcessActivity
    (
        name <- pc.name
        …
    ),

    -- create input activity parameter node
    entryAPN : UWE!ActivityParameterNode
    (
        name <- source.contentClass.name,
        type <- source.contentClass,
        …
    ),

    -- conditionally create output activity parameter node
    exitAPN : distinct UWE!ActivityParameterNode foreach( b in targetSeq )
    (
        name <- target.contentClass.name,
        type <- target.contentClass,
        …
    ),

    -- conditionally create activity final node
    finalNode : distinct UWE!ActivityFinalNode foreach( b in nTargetSeq )
    (
        …
    ),
    …
}
```

Rule CreateProcessDataAndFlowForSimpleProcess

For simple processes the complete process flow and data is generated by the rule *CreateProcessDataAndFlowForSimpleProcess*. The foundation for the generation is the associated operation in the content model. For this operation a call operation action is created.

If the operation has a return type then a corresponding output pin for the call operation action is generated and connected by an outgoing object flow to the output activity parameter node. In the other case, if it has no return type then an activity final node is generated and a control flow from the call operation action to the activity final node, such as for example for the process *RemoveProject* depicted in Figure 52.

If the operation has no parameters (with direction *in*) then an object flow from the input activity parameter node to the target input pin of the call operation action is generated. The target input pin corresponds to the object on which the operation should be invoked. In the other case, if it has parameters, a process data class and a corresponding user action for capturing the input are generated. Further, for each parameter (1) an attribute of the process class, (2) an output pin of the user action, (3) an input pin of the call operation action and (4) an object flow connecting the output pin of the user action with the input pin of the call operation action are generated. Additionally, a fork node is generated with an incoming object flow from the input activity parameter node, an outgoing object flow to the target input pin of the call operation action and an outgoing control flow to the user action. An example for the latter case is again the process *RemoveProject* depicted in Figure 52. For this process a process data class *RemoveProjectInput*, a corresponding user action *RemoveProjectInput* and a call operation action *removeProject* is generated. Further, the parameter *project* of the operation *removeProject* is mapped to a corresponding process property of the process data class, an output pin of the user action and an input pin of the call operation action. Additionally, the required activity edges, the target input pin of the call operation action (for determining on which object the operation should be invoked) and the activity final node are generated.

In comparison to the rule implementation in B.3.8 the following details have been omitted below: target bindings that are not relevant for understanding the rule, targets for the activity parameters, the activity parameters nodes and the optional activity final node, the target for the composite relationship for the generated process class, and targets for the activity edges.

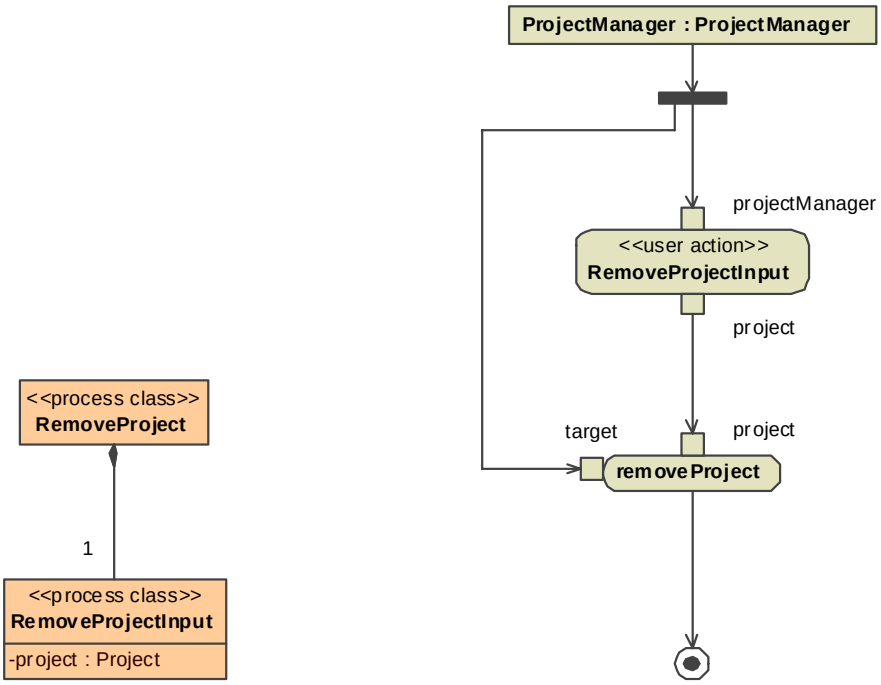


Figure 52. Automatically derived process data and flow for simple process *RemoveProject* derived by rule *CreateProcessDataAndFlowForSimpleProcess*

```
rule CreateProcessDataAndFlowForSimpleProcess
{
    from pc : UWE!ProcessClass ( pc.ownedBehavior->isEmpty() and
        pc.webProcess.oclIsTypeOf( UWE!SimpleProcess ) )
    using
    {
        o : UWE!Operation = pc.webProcess().getTraceTarget( 'SimpleProcess2Operation' );
        inputPar : Sequence( UWE!Parameter ) = o.ownedParameter->select( p |
            p.direction <> #return );
        parSeq : Sequence( Boolean ) = if inputPar->isEmpty() then Sequence {}
            else Sequence { true } endif;
        typeSeq : Sequence( Boolean ) = if o.type.oclIsUndefined() then Sequence {}
            else Sequence { true } endif;
    }
    to tpc : UWE!ProcessClass
    (
        ownedBehavior <- Sequence { pa },
        …
    ),
    pa : UWE!ProcessActivity
    (
        name <- pc.name,
```

```
    …
),

-- create call operation action with target and input pins
coa : UWE!CallOperationAction
(
    name <- o.name,
    operation <- o,
    input <- inputPin->including( targetPin ),
    output <- resultPin,
    target <- targetPin
),
targetPin : UWE!InputPin
(
    name <- 'target',
    type <- o.class_
    …
),
inputPin : distinct UWE!InputPin foreach ( p in inputPar )
(
    name <- p.name,
    type <- p.type,
    …
),

-- create user action and process data class if operation has parameters
userAction : distinct UWE!UserAction foreach ( b in parSeq )
(
    name <- pc.name + 'Input',
    processClass <- inputPC,
    …
),
inputPC : distinct UWE!ProcessClass foreach ( b in parSeq )
(
    name <- pc.name + 'Input',
    …
),

-- create process properties and output pins
pp : distinct UWE!ProcessProperty foreach( p in inputPar )
(
    name <- p.name,
    type <- p.type,
    …
),
outputPin : distinct UWE!OutputPin foreach ( p in inputPar )
```

```
    (
        name <- p.name,
        type <- p.type
    ),

    -- conditionally create output pin
    resultPin : distinct UWE!OutputPin foreach ( b in typeSeq )
    (
        name <- 'result',
        type <- o.type,
        …
    ),

    -- conditionally create fork node if operation has parameters
    forkNode : distinct UWE!ForkNode foreach ( b in parSeq )
    (
        …
    ),
    …
}
```

Rule CreateProcessDataAndFlowForEdit

The complete process flow and data is generated for edit processes by the rule *CreateProcessDataAndFlowForEdit*. For each edit process a process data class with a copy of the attributes (per default only those with a primitive type or an enumeration type) of the corresponding content class is generated. Further, a user action that uses this process data class for receiving input from the user is constructed. An input pin of this user action receives an object flow from the input activity parameter node to specify which object should be edited. Finally, an outgoing control flow from the user action is connected to an activity final node. An example for the edit process *EditUserProject* is depicted in Figure 53. In comparison to the rule implementation in B.3.8 the following details have been omitted below: target bindings that are not relevant for understanding the rule, targets for the input parameter, the input activity parameters node and the activity final node, the target for the composite relationship for the generated process class, and targets for the activity edges.

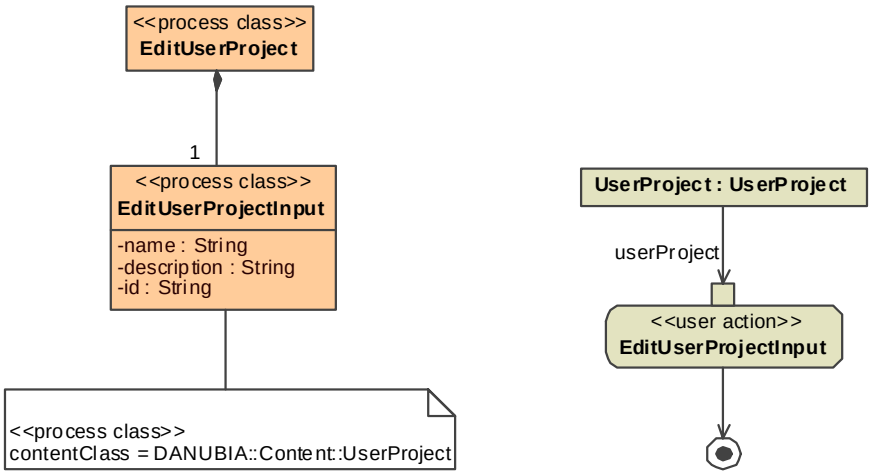


Figure 53. Automatically derived process data and flow for edit process *EditUserProject* derived by rule *CreateProcessDataAndFlowForEdit*

```
rule CreateProcessDataAndFlowForEdit
{
    from pc : UWE!ProcessClass ( pc.ownedBehavior->isEmpty() and
        pc.webProcess.oclIsTypeOf( UWE!Edit ) )
    using
    {
        source : UWE!NavigationClass = let ls : Set( UWE!Link ) = pc.inLinks in
            if ls->isEmpty() then OclUndefined else ls->any().source endif;
    }
    to tpc : UWE!ProcessClass
    (
        ownedBehavior <- Sequence { pa },
        …
    ),
    pa : UWE!ProcessActivity
    (
        name <- pc.name,
        …
    ),

    -- create user action
    userAction : UWE!UserAction
    (
        name <- pc.name + 'Input',
        input <- Sequence { inputPin },
        …
    ),
    inputPin : UWE!InputPin
```

```
    (
        name <- source.contentClass.name.firstToLower(),
        type <- source.contentClass,
        …
    ),

    -- create process data class
    inputPC : UWE!ProcessClass
    (
        name <- pc.name + 'Input',
        contentClass <- source.contentClass,
        ownedAttribute <- pp
    ),
    pp : distinct UWE!ProcessProperty foreach( cp in source.contentClass.allOwnedAttribute()->
        select( p | p.type.oclIsKindOf( UWE!DataType ) and not p.isMultivalued() ) )
    (
        name <- cp.name,
        type <- cp.type,
        lower <- cp.lower,
        upper <- cp.upper,
        editProperty <- cp
    ),
    …
}
```

4.5.2.3 Manual Refinement

For general Web processes, i.e. neither simple processes nor edit processes, the process data and flow has to be completely defined by the developer, with exception of the automatically generated parameters and activity parameter nodes. In Figure 54 the manually defined process flow for the process *AddProject* of the running example is depicted. It comprises three user actions and two call operation actions. The first user action *ProjectKindInput* is used to query the kind of project the user wants to add to the project list. Depending on the output of the user action, which is represented by an enumeration type (see below), either the user action *AddValidationProjectInput* or *AddUserProjectInput* is executed to query the parameters for the subsequent call operation action *addValidationProject* or *addUserProject*, respectively. Note that these two call operation actions require different parameters, which have to be provided by the corresponding user actions. Further, the user action *AddUserProjectInput* requires an input pin for the selection of a validation project from a collection of validation projects (see below). After the termination of either call operation action the corresponding project object is passed through a merge node to the output activity parameter node. Taking advantage of the dynamic navigation feature of

this approach, either the page for a validation project or for a user project is then shown to the user.

The process data required for the process flow of the process *AddProject* is depicted in Figure 55. For each user action a process class was defined. The process class *ProjectKindInput* captures the selection of a project kind. Therefore a special enumeration type *ProjectKind* was defined. The process class *AddValidationProjectInput* corresponds to the parameters of the operation *addValidationProject* and therefore two attributes of type *String* are required. For the operation *addUserProject* an additional attribute *validationProject* is required for the process class *AddUserProjectInput*. The selection of a validation project is optional, hence the multiplicity of the attribute is 0..1. Additionally, a *rangeExpression* has to be defined for attributes which are neither of primitive type nor enumerations, to express in terms of an expression in the expression language the collection from which the value of the attribute should be chosen. In this case the collection is given be the property *validationProjects* of the content class given be the specified *contentClass* for the process class (see below).

Additionally, the definition of the general Web process *AddProject* requires an extension of the content model as depicted in Figure 56. On the one hand the operations *addValidationProject* and *addUserProject,* that are invoked by the introduced call operation actions, have to be added to the content class *ProjectManager*. On the other hand a derived attribute *validationProjects* has to be introduced which is used for the selection of a validation project. Note that the expression language is not expressive enough to express the value of this attribute directly. When generating code for the content model a getter operation *getValidationProjects* is generated that has to be completed by the developer to return the set of available validation projects.

For simple processes which require the input of a value other than a primitive type or an enumeration, the automatically derived process data has to be refined by the developer in order to define the corresponding *rangeExpression* properties as already explained above. In the running example this is the case for the process *RemoveProject* as depicted in Figure 58. Additionally, it was chosen to add a further user action to confirm the remove action, see Figure 57. The input is represented by a particular process class which uses the special enumeration type *YesNoEnum*. Depending on the output of this confirm user action either the corresponding call operation action is triggered or the process terminates because a token reaches the activity final node directly.

Edit processes do not require manual refinement, but for the running example it was chosen not to let the user edit all attributes of the corresponding content class. Therefore, the

automatically generated attribute *id* was removed from the automatically generated process class *EditUserProject* as depicted in Figure 59.

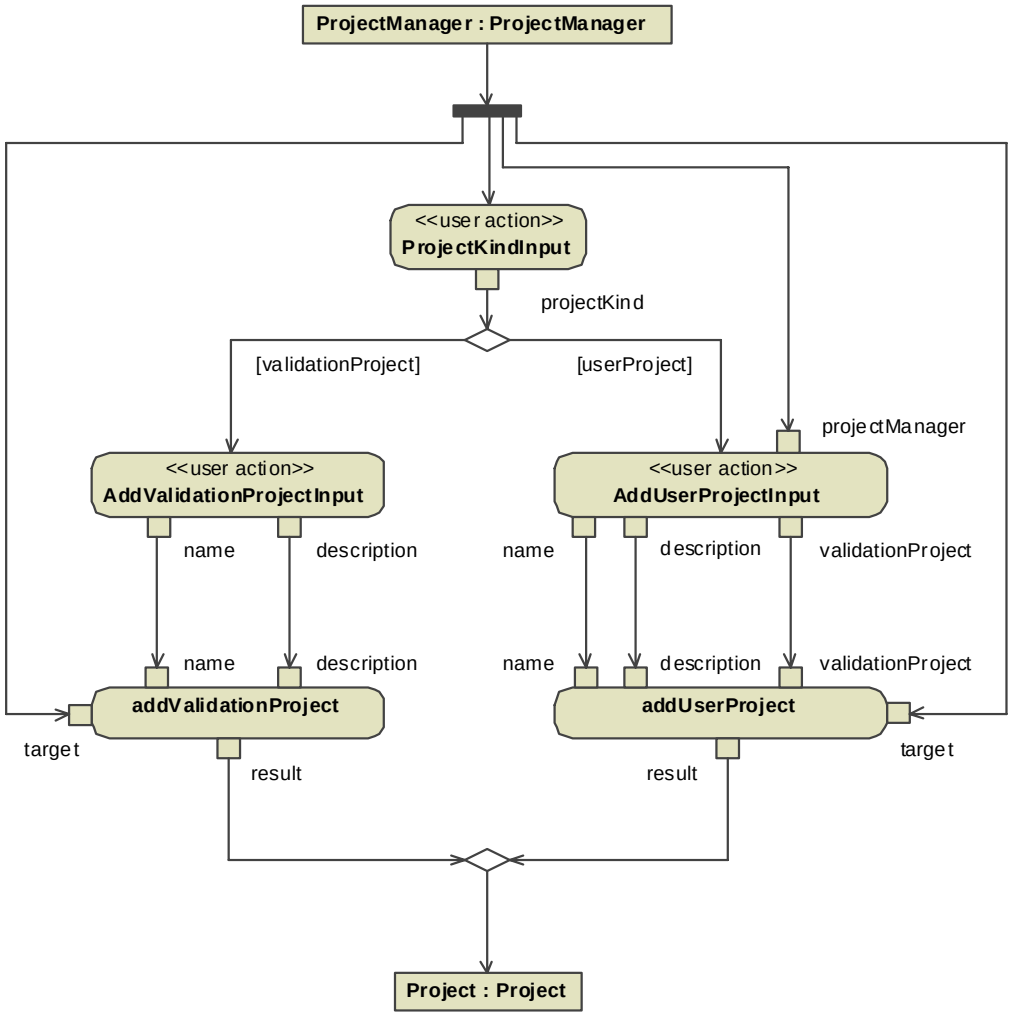


Figure 54. Manually refined process flow for process *AddProject*

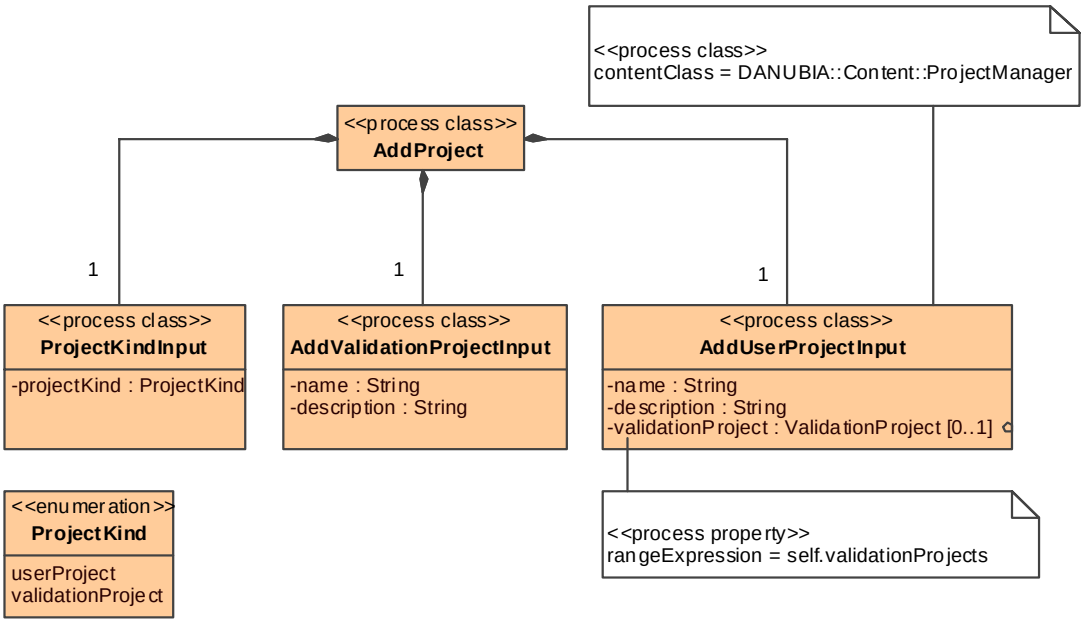


Figure 55. Manually specified process data for process *AddProject*

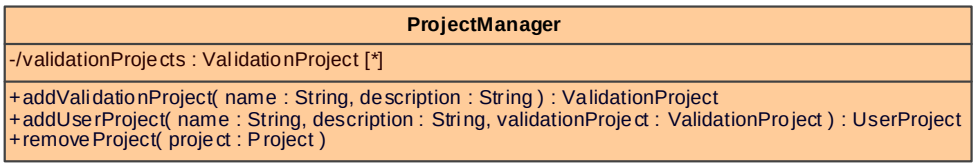


Figure 56. Refined content model for process *AddProject*

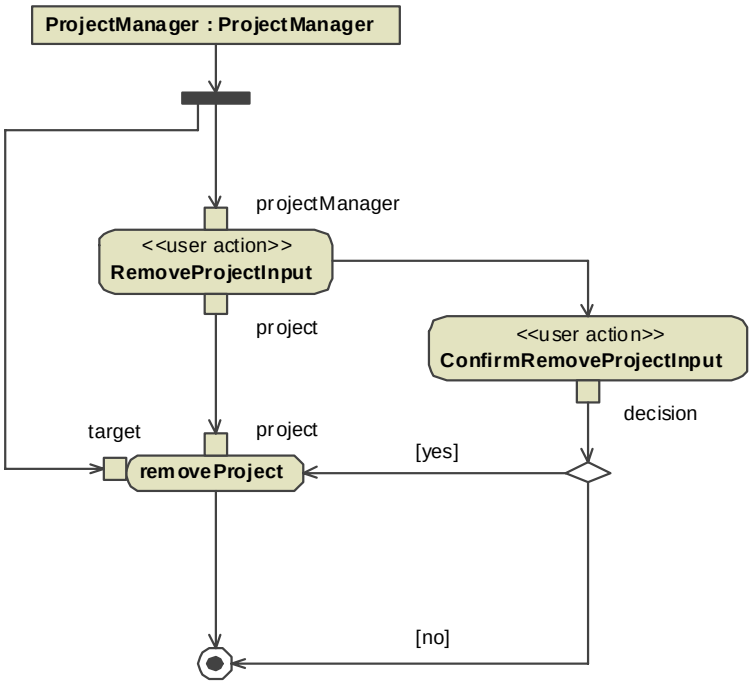


Figure 57. Manually refined process flow for process *RemoveProject*

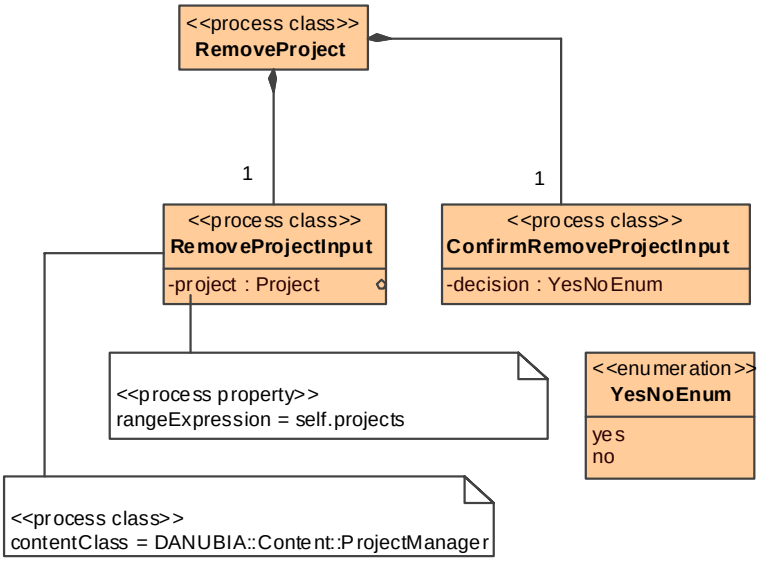


Figure 58. Manually refined process data for process *RemoveProject*

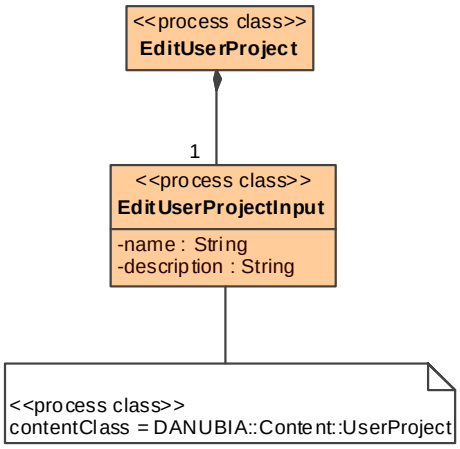


Figure 59. Manually refined process data for process *EditUserProject*

4.6 Presentation

The presentation model defines the layout for the underlying navigation and process models. In the same way as classes describe the structure of objects, specialized classes are used to define the structure of Web pages. Presentation classes represent Web pages and are composed of user interface elements and other presentation classes. In addition to a pure logical layout physical properties of the resulting Web pages can also be defined at the level of the presentation model. This includes the ordering of model elements and the definition of CSS properties (Cascading Style Sheets) for the final presentation model.

Only few Web approaches support presentation modeling at a platform independent level. Presentation modeling in W2000 [Baresi06] and OOHDM [Schwaabe98] is similar to this approach (except for the proprietary notation), while in OOWS [Fons03] the presentation aspect is integrated with the navigation aspect, thus a dedicated presentation model for further abstraction of the user interface is not available. Approaches such as for example OO-H [Cachero03] allow the user to graphically design the layout of a Web application by using a proprietary layout editor. The layout information is then normally saved to XML files. Other approaches, such as for example WebML [Ceri02] do not provide any kind of presentation model and directly translate the navigation model to code.

Presentation modeling based on UML modeling elements as presented here is limited to be an abstraction of the final physical layout due to the inherent limitations of the UML notation itself. For example, the dimensions of user interface elements in a UML diagram are

not part of a UML model and hence cannot be used to be translated to corresponding dimensions in a Web page. This limitations have been overcome to some degree by the introduction of physical layout properties by means of CSS, which can be used (amongst others) to assign physical dimensions to user interface elements. Also, this approach to presentation modeling is targeted at modeling the user interface of traditional Web applications, i.e. Web applications that follow strictly the request-response pattern imposed by the underlying HTTP communication protocol, in contrast to more responsive Web applications, so-called Rich Internet Applications (RIA).

The presentation metamodel could easily be extended by the introduction of additional user interface element types if required. Further, additional behavioral models for handling user interface events would allow modeling more responsive user interfaces of Rich Internet Applications.

A default presentation model is derived from the navigation model by the transformation *NavigationAndProcess2Presentation* presented in the next section. The default presentation model then has to be refined by the developer resulting in the final presentation model.

4.6.1 Metamodel

The backbone of the presentation metamodel is depicted in Figure 60. A presentation class is a specialized class which represents a Web page or a part of it, when presentation classes are composed. Each presentation class is associated to exactly one node from the navigation model. For each presentation class the physical layout may be defined by providing either one or both of the attributes *cssClass* and *cssStyle* (see below). A presentation property is a specialized property that can be associated to a property of a node. Only composite presentation properties are allowed and the type of a presentation property is constrained to either presentation classes or user interface elements.

User interface elements are specialized classes that represent the user interface elements in a Web page. Different types of user interface elements are distinguished, see Figure 61. Anchors represent links in a Web page, and optionally a format expression may be defined for specification of the label that the anchor should have (see below). Other (abstract) super types of user interface elements are output elements, input elements and static elements. For each user interface element the physical layout may be defined by providing either one or both of the attributes *cssClass* and *cssStyle* (see below).

Output elements allow the presentation of dynamic data. Two types of output elements are defined, see Figure 62. Text elements allow the presentation of arbitrary data that can be

represented as text. Image elements allow the output of images which are accessible by a URL. If the data corresponds to relative URLs then a base URL must be defined.

Static elements present static information on a Web page, see Figure 62. Static information is not calculated from the content model, but must be defined at design time. Static texts present text that is defined by the attribute *text* at design time, and static images present images, whose URL is defined at design time by the attribute *url*.

Input elements are user interface elements that are used for capturing input data from the user, see Figure 63. Textual input is represented by the user interface element text input. This includes all kind of input that can be parsed from a string, as for example numbers. Enumeration input is especially used for capturing the choice out of the enumeration literals of an enumeration. Finally, a selection user interface element is used for a selection of objects out of a collection of objects. An optional *format* property can be defined to specify how the objects of the collection should be presented as a string.

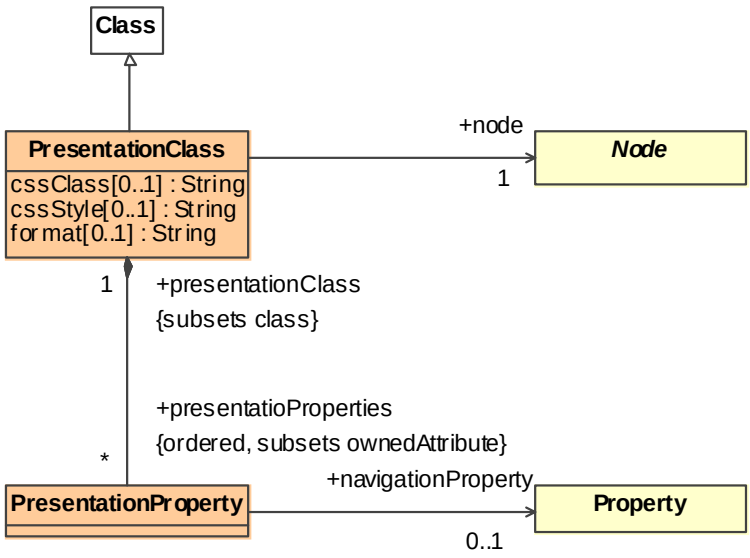


Figure 60. Metamodel for presentation modeling (backbone)

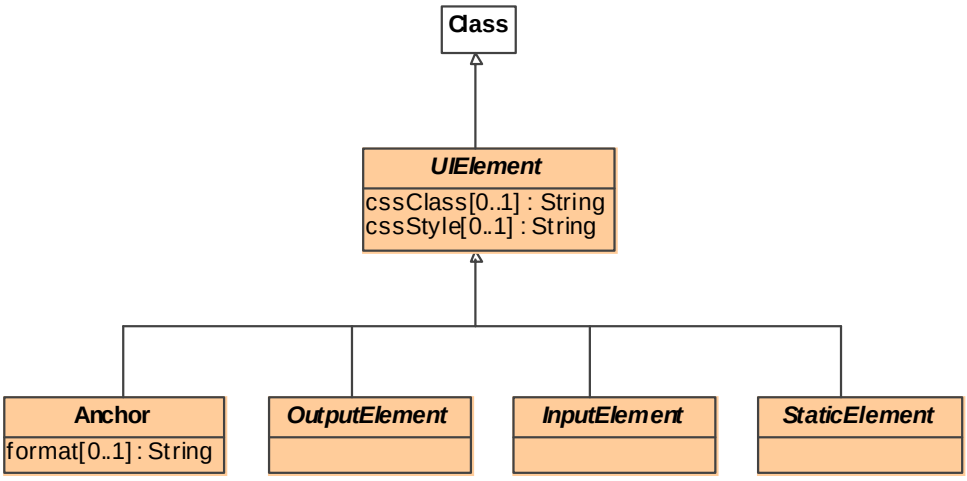


Figure 61. Metamodel for presentation modeling (user interface elements)

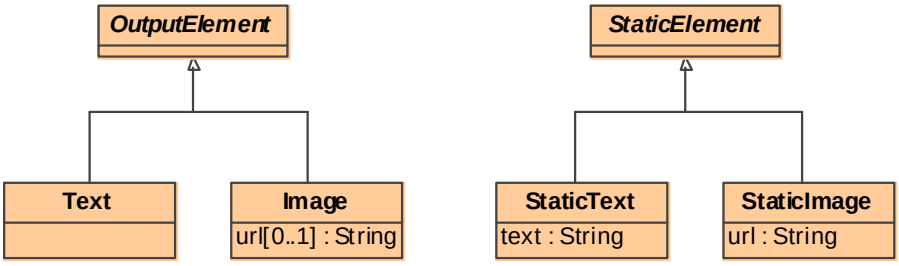


Figure 62. Metamodel for presentation modeling (output and static elements)

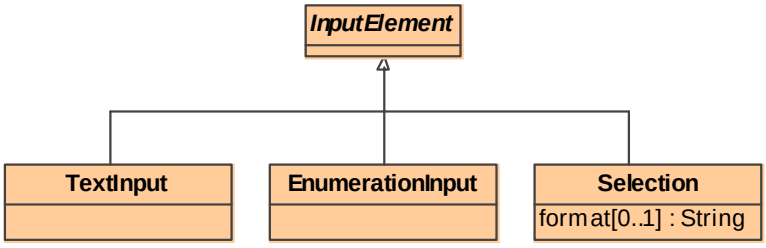


Figure 63. Metamodel for presentation modeling (input elements)

Constraints

For each navigation class, access primitive or process data class exactly one presentation class must be defined.

```
context Node inv NodePresentationClassDefined :
    not self.isAbstract and ( self.oclIsKindOf( ProcessClass ) implies self.inLinks->isEmpty() )
        implies PresentationClass.allInstances()->one( pc | pc.node = self )
```

Inheritance is not allowed for presentation classes.

```
context PresentationClass inv PresentationClassInheritance :
    self.parents()->isEmpty()
```

The type of a presentation property must be either a user interface element or a presentation class.

```
context PresentationProperty inv PresentationPropertyType :
    self.type.oclIsKindOf( UIElement ) or self.type.oclIsKindOf( PresentationClass )
```

If type of a presentation property is a static element then no navigation property must be defined. The presentation properties of the presentation class for a process class have to be associated to a process property. On the other hand, the presentation properties of the presentation class for an access primitive must not define a navigation property. For all other cases the presentation property must be associated to a navigation property.

```
context PresentationProperty inv PresentationPropertyNavigationProperty :
    if self.type.oclIsKindOf( StaticElement ) then
        self.navigationProperty->isEmpty()
    else if self.class.node.oclIsKindOf( ProcessClass ) then
        self.navigationProperty.oclIsKindOf( ProcessProperty )
    else if self.class.node.oclIsKindOf( AccessPrimitive ) then
        self.navigationProperty->isEmpty()
    else
        self.navigationProperty.oclIsKindOf( NavigationProperty )
    endif endif endif
```

Inheritance is not allowed for user interface elements.

```
context UIElement inv UIElementInheritance :
    self.parents()->isEmpty()
```

A user interface element must be the type of exactly one presentation property of a presentation class with composite aggregation kind.

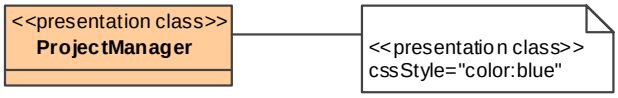
```
context UIElement inv UIElementContainment :
    let ps : Set( PresentationProperty ) = PresentationProperty.allInstances()->select( p |
        p.type = self ) in
            ps->size() = 1 and
            ps->forAll( p | p.isComposite and p.class.oclIsKindOf( PresentationClass ) )
```

Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) is a standard by the World Wide Web Consortium (W3C) for adding style, or physical layout, to Web documents [CSS]. Style sheets describe how documents should be presented on screens or on other media. Documents can be arbitrary XML documents and especially (X)HTML documents. CSS is the default style sheet language for the Web.

CSS defines styles in a declarative way. The language elements are selectors and property definitions. Selectors express when a style definition should be applied. A style definition consists of a list of property definitions of the format "*property-name:property-value"*. Property classes are for example fonts, colors, margins or borders. For detailed information about CSS see [CSS].

Within this work the use of CSS is supported in two variants which may even be combined. The first variant is the direct assignment of a CSS style definition to a presentation class or a user interface element by defining the attribute *cssStyle*. The following style definition renders all the text within a presentation class with the text color blue:



The second variant is the use of style classes by assigning the name of a style class to the attribute *cssClass*. The styles for all elements of a specific class can then be defined globally by using a class selector. For detailed information where this style definition has to be made see 6.2.4.2 .

Formatting Expressions

For the anchor and the selection user interface element a formatting expression can be defined in order to provide the labels required by these user interface elements, see 4.1.3. The context for references to the properties of an object is in the case of an anchor the actual target object. In the case of a selection it is the actual object of the collection that should be rendered on the user interface. For example, a possible format expression for the anchor of the project index could be *"#${id} - ${name}"*. The text outside the *"${}"* expressions represents the static part of the resulting text and the text inside is evaluated at runtime by querying the *id* and *name* properties of the actual item that should be displayed in the index.

Notation

The appropriate notation for the presentation model is a stereotyped composite structure diagram for each presentation class with the user interface elements as parts with the corresponding multiplicities. Equally, regular class diagrams can be used. For a definition of the corresponding UML profile see A.5.

4.6.2 Transformation NavigationAndProcess2Presentation

The transformation *NavigationAndProcess2Presentation* depicted in Figure 64 automatically derives a presentation model from the navigation model and the process model. The transformation comprises four transformation rules which are outlined below and detailed in B.3.9.

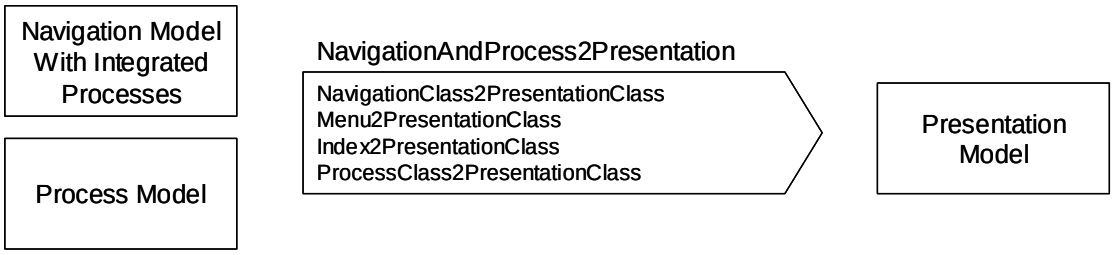


Figure 64. Transformation *NavigationAndProcess2Presentation*

For each node in the navigation model and each process data class a presentation class is constructed and for each attribute a corresponding presentation property with the type of a user interface element is created.

The automatically derived presentation class for the navigation class *ProjectManager* of the running example is depicted in Figure 65. The composite parts of the navigation class, the *ProjectManagerMenu* and the *ProjectIndex*, have been transformed to the corresponding composite parts of the presentation class. The project manager menu contains two anchors for the processes add and remove project.

The presentation class for user projects is depicted in Figure 66. Each attribute of the user project is represented by a text user interface element. The menu comprises an anchor to the edit user project process and two anchors to the navigation classes *ProjectManager* and *ValidationProject*. The former provides a back link to the project manager while the latter leads to the associated validation project.

The presentation classes corresponding to the process data classes of the add project process are depicted in Figure 67. Text input elements are used for capturing textual data that is needed by the process for the creation of a new project. An enumeration input element is required for the user selection of the desired project type. Finally, a selection element is used for the selection of a validation project out of a collection of validation projects.

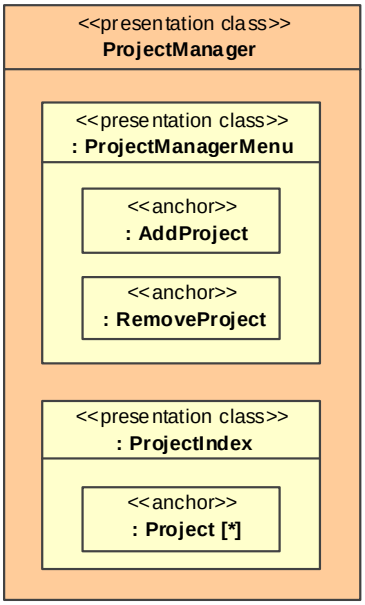


Figure 65. Automatically derived presentation classes for the navigation class *ProjectManager*, menu *ProjectManagerMenu* and index *ProjectIndex*

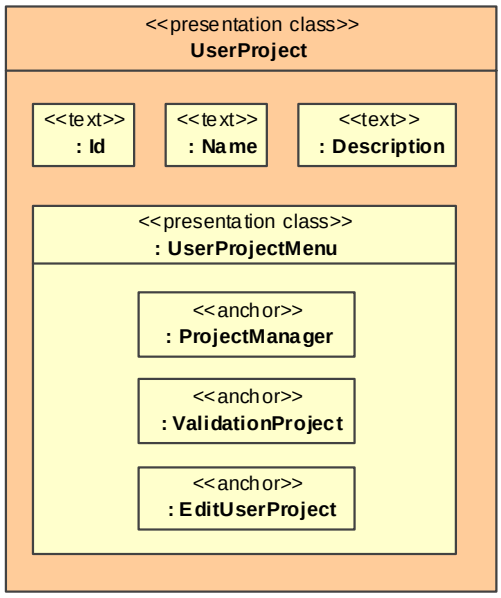


Figure 66. Automatically derived presentation classes for the navigation class *UserProject* and for the menu *UserProjectMenu*

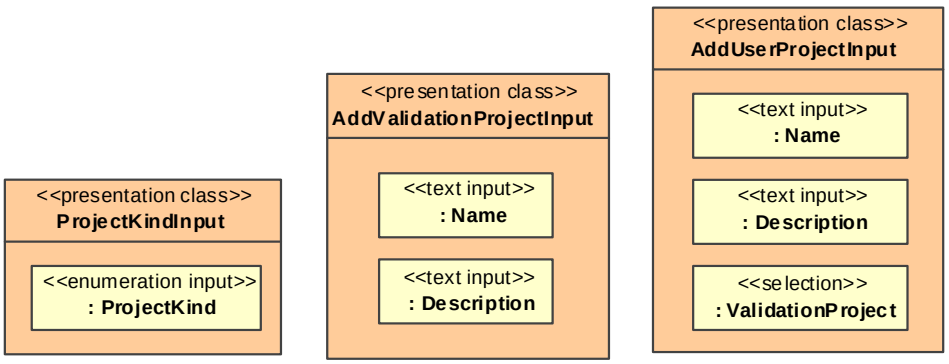


Figure 67. Automatically derived presentation classes for the process classes *ProjectKindInput, AddValidationProjectInput* and *AddUserProjectInput* of process *AddProject*

Rule NavigationClass2PresentationClass

For each navigation class (exact type) a presentation class is generated by this rule. Further, for each attribute of the navigation class (including inherited attributes) with a datatype type, i.e. either a primitive type or an enumeration type, a text user interface element is generated. For each outgoing link an anchor user interface element is generated. In comparison to the rule implementation in B.3.9 the following details have been omitted below: target bindings that are not relevant for understanding the rule, and targets for the

presentation properties that correspond to the composition relationship between the generated presentation class and the generated user interface elements.

```
rule NavigationClass2PresentationClass
{
    from nn : UWE!NavigationClass (
        not nn.isAbstract and nn.oclIsTypeOf( UWE!NavigationClass ) )
    to tnn : UWE!NavigationClass ( … ), -- target for copying source element
    pc : UWE!PresentationClass
    (
        …
    ),
    textUis : distinct UWE!Text foreach ( p in nn.allOwnedAttribute()->select( p |
            p.type.oclIsKindOf( UWE!DataType ) ) )
    (
        name <- p.name.firstToUpper(),
        …
    ),
    anchorUis : distinct UWE!Anchor foreach ( p in nn.allOwnedAttribute()->select( p |
            p.association.oclIsKindOf( UWE!Link ) ) )
    (
        name <- p.type.name,
        …
    ),
    …
}
```

Rule Menu2PresentationClass

For each menu a presentation class is generated by this rule. For each outgoing link an anchor user interface element is generated. In comparison to the rule implementation in B.3.9 the following details have been omitted below: target bindings that are not relevant for understanding the rule, and targets for the presentation properties that correspond to the composition relationship between the generated presentation class and the generated user interface elements.

```
rule Menu2PresentationClass
{
    from nn : UWE!Menu ( not nn.isAbstract )
    to tnn : UWE!Menu ( … ), -- target for copying source element
    pc : UWE!PresentationClass
    (
        …
```

```
    ),
    anchorUis : distinct UWE!Anchor foreach ( p in nn.allOwnedAttribute()->select( p |
            p.association.oclIsKindOf( UWE!Link ) ) )
    (
        name <- p.type.name,
        …
    ),
    …
}
```

Rule Index2PresentationClass

For each index a presentation class is generated by this rule. Additionally, an anchor user interface element is generated for the outgoing link. In comparison to the rule implementation in B.3.9 the following details have been omitted below: target bindings that are not relevant for understanding the rule, and the target for the presentation property that correspond to the composition relationship between the generated presentation class and the anchor.

```
rule Index2PresentationClass
{
    from nn : UWE!Index
    to tnn : UWE!Index ( … ), -- target for copying source element
    pc : UWE!PresentationClass
    (
        …
    ),
    anchorUi : UWE!Anchor
    (
        name <- nn.outLinks->first().target.name,
        …
    ),
    …
}
```

Rule ProcessClass2PresentationClass

For each process class that represents process data a presentation class is generated by this rule. Further, for all attributes of the process class with a primitive type a text input element is generated. Attributes with an enumeration type are mapped to an enumeration input element and all other attributes are mapped to a selection input element. In comparison to the rule implementation in B.3.9 the following details have been omitted below: target

bindings that are not relevant for understanding the rule, and targets for the presentation properties that correspond to the composition relationship between the generated presentation class and the generated user interface elements.

```
rule ProcessClass2PresentationClass
{
    from nn : UWE!ProcessClass ( nn.inLinks->isEmpty() )
    to tnn : UWE!ProcessClass ( … ), -- target for copying source element
    pc : UWE!PresentationClass
    (
        …
    ),
    textInputUis : distinct UWE!TextInput foreach ( p in nn.allOwnedAttribute()->select( p |
        p.type.oclIsKindOf( UWE!PrimitiveType ) ) )
    (
        name <- p.name.firstToUpper(),
        …
    ),
    enumerationInputUis : distinct UWE!EnumerationInput foreach (
        p in nn.allOwnedAttribute()->select( p | p.type.oclIsKindOf( UWE!Enumeration ) ) )
    (
        name <- p.name.firstToUpper(),
        …
    ),
    selectionUis : distinct UWE!Selection foreach ( p in nn.allOwnedAttribute()->select( p |
        p.type.oclIsTypeOf( UWE!Class ) ) )
    (
        name <- p.name.firstToUpper(),
        format <- p.type.name,
        …
    ),
    …
}
```

4.6.3 Manual Refinement

The automatically derived presentation model can optionally be refined by the developer. Possible optional activities are:

- Reordering of presentation properties
- Addition of static elements
- Definition of CSS (Cascading Style Sheets) styles

- Definition of format expressions for anchors and selection elements

For the running example, a static text element has been added manually to the presentation class for the project manager, to provide the project manager page with a caption as depicted in Figure 68. Additionally, the anchor of the project index has been provided with a format expression, in order to render the index elements with a meaningful label. Further, for the anchor of the user project menu, a format expression has been defined to include the name of the validation project in the label of the anchor, see Figure 69.

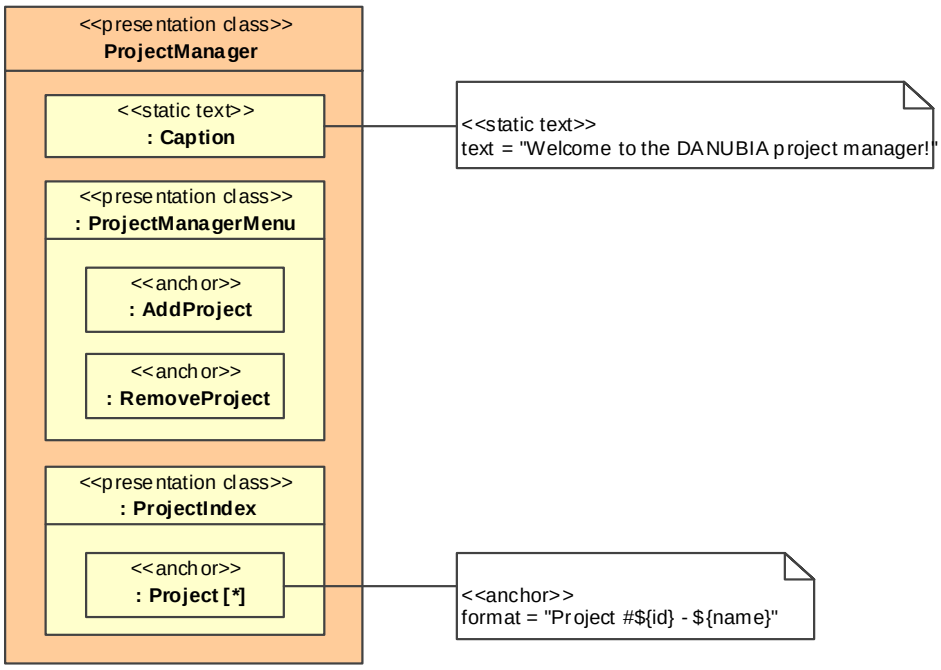


Figure 68. Manually refined presentation classes for the navigation class *ProjectManager*, menu *ProjectManagerMenu* and index *ProjectIndex*

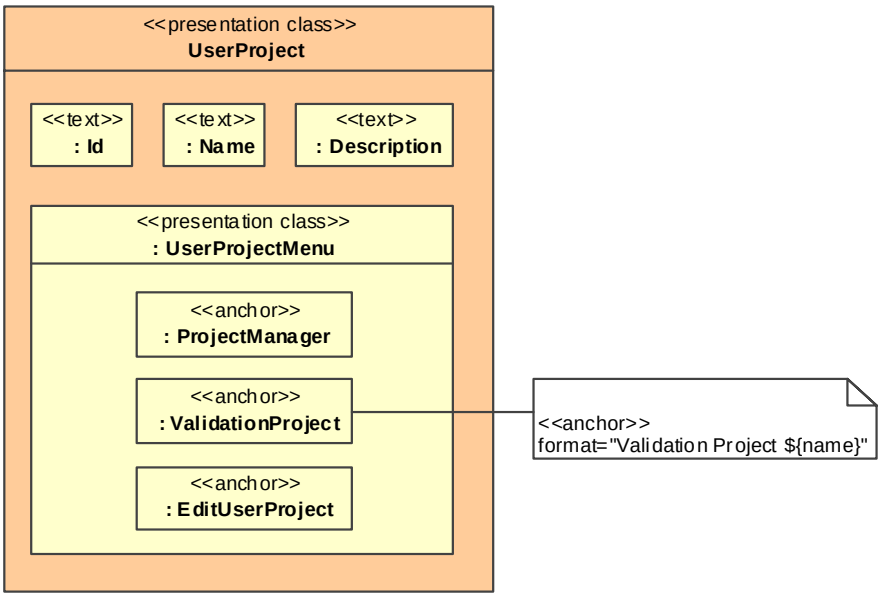


Figure 69. Manually refined presentation classes for the navigation class *UserProject* and for the menu *UserProjectMenu*

4.7 Transition to the Platform Specific Implementation

For the transition to the platform specific implementation all platform independent design models must be complete, i.e. the content, navigation, process and presentation models must have been constructed as presented in the previous sections and all of the well-formedness rules must be fulfilled. The requirements model is not required for this transition, it only serves as starting point for the construction of the design models.

5 PLATFORM SPECIFIC IMPLEMENTATION

In this chapter a model driven implementation approach for Web applications is presented. Following the vision of MDA, the implementation platform is represented by a corresponding metamodel, and a transformation *PIM2PSM* transforms the platform independent design models presented in the last chapter to the platform specific implementation models. In a final step, the platform specific implementation models are serialized to code.

As discussed in 3.2, the transformation from the platform independent models to the platform specific models should be decomposed into four different transformations for the content, navigation, process and presentation concerns of a Web application, see Figure 70. Each partial transformation is targeted at a specific part of the Web platform (or technology) that is responsible for handling the corresponding concern. Depending on the concrete Web platform, one part could be exchanged without influencing the other parts and the corresponding transformations. When a part of the Web platform (or technology) is exchanged, only a new transformation and a corresponding metamodel would have to be defined for the exchanged part. In practice, independence of the parts and the corresponding partial transformations among each other is only achieved if the platform provides some kind of abstraction technique for the communication between the parts.

In the following sections, first a generic platform for Web applications that allows such a decomposition of the transformation to the platform specific models is presented. It is built on an open-source Web framework and a generic runtime environment, representing a family of platforms for supporting the combination of a broad range of technologies. The parts of the platform are designed to be independent from each other by the introduction of corresponding abstraction techniques for the communication among each other. Then the transformations for the content, navigation, process and presentation concerns are presented. The use of two different technologies for the content concern, JavaBeans and RMI, demonstrates the flexibility of the approach..

In comparison to other model driven Web engineering approaches presented in 3.4, technologies are represented by metamodels and code generation is achieved exclusively by using transformations: ATL transformations map the platform independent design models to the platform specific implementation models, and ATL queries serve to serialize these models to code; further, a decomposition of the transformation to the platform specific models is proposed together with a generic platform that can be used for supporting a broad range of technologies.

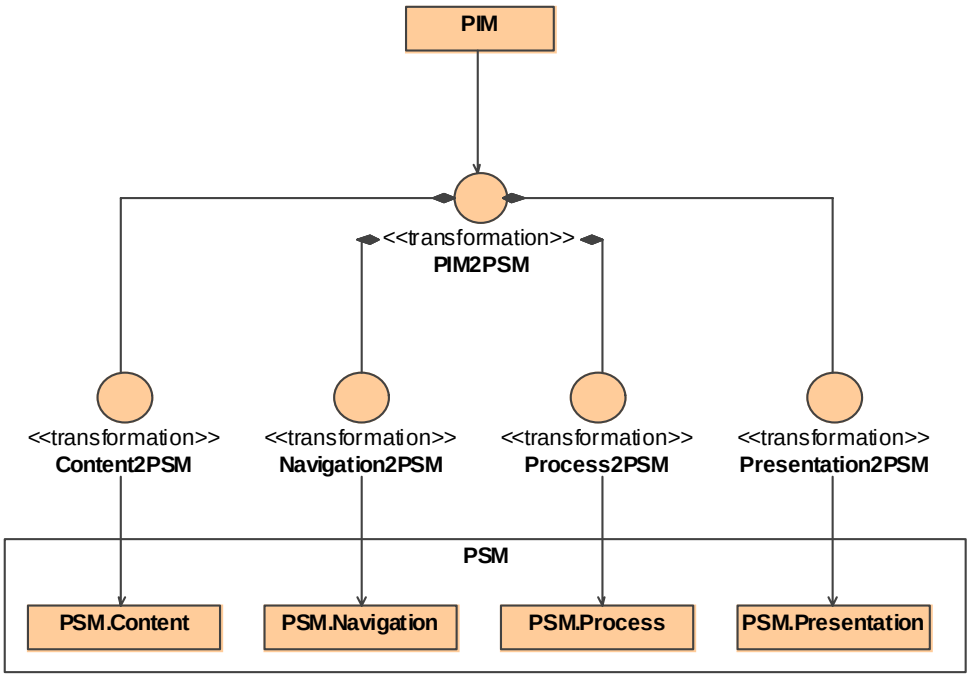


Figure 70. Decomposed *PIM2PSM* transformation

5.1 Generic Platform

A platform is an environment that allows software targeted for this platform to be run. Examples for platforms are hardware platforms, operating systems or virtual machines. A software system itself is a platform if it provides an environment for other software to be run. Other terms for a platform are framework or architecture, depending on the context. Usually a platform is not monolithic, but consists of a kernel and pluggable platform components, which form part of the configuration of a platform. Further, a platform often builds on top of other platforms or it depends on other platforms. Most platforms provide

lightweight extension mechanisms to be extended by the developer for a specific application.

Web platforms which are also called Web containers provide an environment for running Web applications. In the beginning of the Web, Web applications were often developed "from scratch" by directly implementing the HTTP protocol. Nowadays, Web application development is always targeted at a particular kind of Web platform.

Today, a zoo of Web platforms is available for the developer to choose from. In Figure 71 some of the most common current Web platforms are depicted. Most of them are built on the platform for a specific programming language or virtual machine, for instance J2EE builds on the Java platform, ASP.NET builds on the .NET platform and Ruby on Rails builds on the language Ruby. Some of these platforms require a specific Web server, for instance the Internet Information Server (IIS) from Microsoft is needed for ASP.NET, while others are more flexible and can be configured to be plugged into a variety of Web servers. Some platforms even depend on a specific operating system such as ASP.NET, which needs the Microsoft Windows platform[10].

The currently available Web platforms can be partitioned into three categories. The first category is the heavyweight ASP.NET platform, which is strongly dependent on the Microsoft .NET technology and the Windows operating system. The second category comprises the lightweight open-source Tomcat Java Servlet/JSP Container and platforms that build on it such as Struts, Cocoon or the Spring framework. The Java 2 Enterprise Edition (J2EE) is a heavyweight extension of the Java Servlet/JSP Container and corresponds to ASP.NET from Microsoft. Agile and/or lightweight Web platforms such as Ruby on Rails (cf. 3.4.6) that allow for fast development of Web applications fit in the last category.

In addition to being a Web platform the heavyweight platforms ASP.NET and J2EE also provide a complex component model, i.e. .NET components and Enterprise JavaBeans (EJB) components, respectively. As already stated, the approach of this work does not aim at the model driven implementation of components. The objective is to compose the invocation of services provided by theses components by means of processes, see 4.5.2. Therefore a lightweight Web platform is favored, which facilitates the use of components (or services). These components could be implemented by using a complex component model such as EJB or Web Services.

[10] Although platform independent implementations for .NET such as Mono or dotGNU exist, still the common runtime libraries needed by ASP.NET applications are available for Windows only

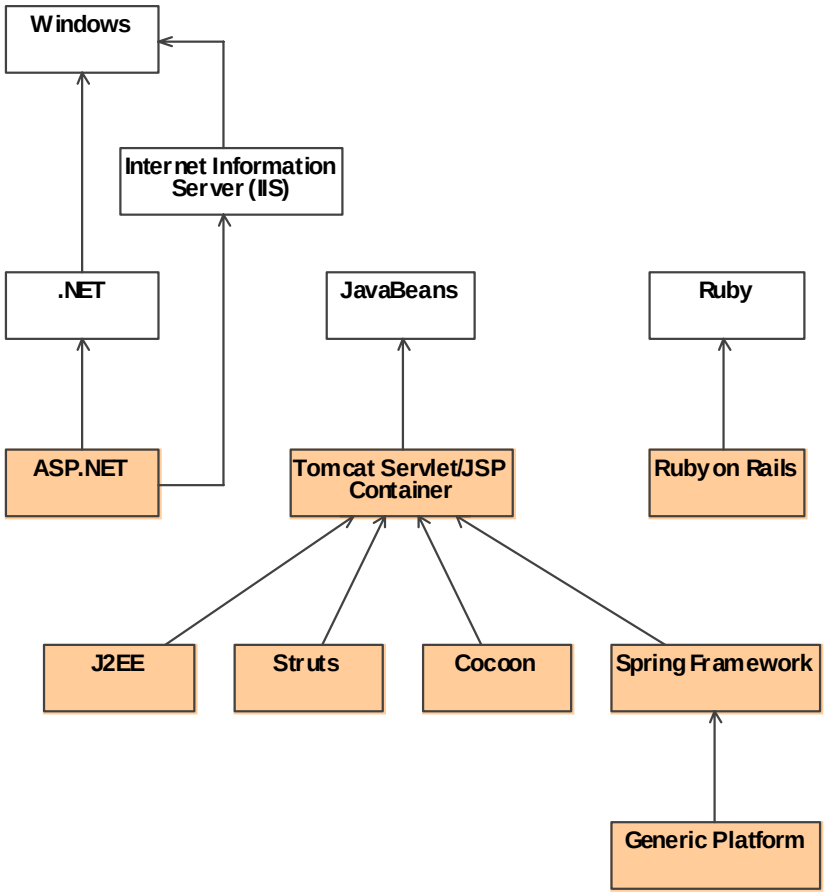


Figure 71. Common Web platforms and the proposed generic platform

The proposed generic platform is based on the Spring framework, which is presented in the next section. Spring provides a Web framework that offers a high degree of flexibility for the combination of different technologies and therefore qualifies as a generic Web platform. The Spring Web framework relies on the Model/View/Controller (MVC) pattern [Reenskaug79], where the concerns of a Web application correspond to the model (content), view (presentation) and controller (navigation and process) roles in the MVC pattern. This allows for a corresponding decomposition of the transformation to the platform specific models as depicted in Figure 72. For a concrete model technology (e.g. JavaBeans) or view technology (e.g. Java Server Pages) corresponding metamodels and transformations have to be defined. An abstraction technique (see next section) for accessing the model and view objects from the controller allows to decouple the concrete model and view technologies from the controller implementation. This is represented as inheritance relationships for the model and view technologies in Figure 72. In the same way does the view technology not depend from the model technology by using another abstraction technique. A generic runtime environment plugged into the Spring framework takes the controller part of a Web

application implementation, see 5.1.2. This controller has to be configured for a specific Web application by configuration data generated from the navigation and the process models. Therefore, a transformation based approach for using the configuration facilities of Spring to configure the runtime environment is presented.

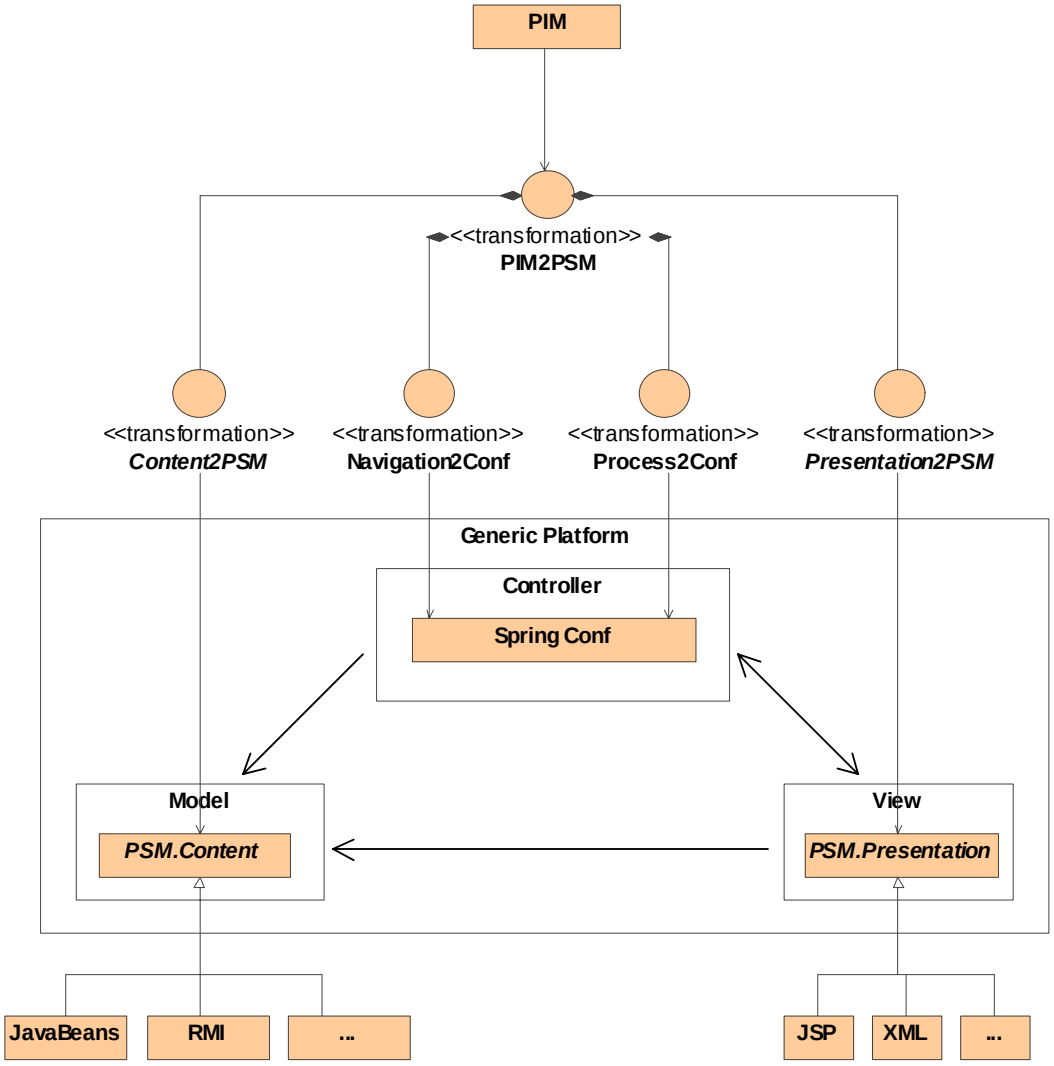


Figure 72. Decomposition of the *PIM2PSM* transformation for the generic platform

It has to be stressed that the results presented in the following depend on the choice of the underlying platform, i.e. the Spring Web framework. While the transformations for the content and presentation concerns could as well be used with other Web frameworks, the transformations for the navigation and process concerns and the runtime environment would have to be adapted accordingly. Additionally, most other Web frameworks are more restricted in the choice of technologies for the content and presentation concern.

5.1.1 Spring Framework

The Spring framework [Spring] is a multi-purpose framework based on the Java platform. Although the important part for this work is the Spring Web framework, it can also be used independently of the Web application context. Integration facilities for different technologies for several domains, such as persistence or transaction management, are provided. A modular architecture facilitates extensibility and reuse. The following modules are comprised:

- Web framework
- Beans (factory, naming services, events, …)
- Support of common middleware technologies like CORBA, SOAP or Web services
- Direct Access Objects (DAO, database abstraction layer)
- Object Relational Mappings (ORM, integration layer for object relational mappings, e.g. JDO, Hibernate, iBatis)
- Transaction management
- Aspect Oriented Programming (AOP, support for aspect oriented programming conform with AOP alliance and AspectJ)

The Spring Web framework is based on the Model/View/Controller (MVC) pattern [Reenskaug79]: the model encapsulates the core application data and functionality, the view presents data from the model to the user and the controller receives requests from the user, modifies the model and updates the view. For Web application development a slightly modified version of the pattern named MVC 2, MVC Version 2 or MVC Model 2 [Sun02] is used resembling the strict HTTP request/response protocol. The view is updated only on each user request and there is no mechanism so that the model can trigger an update in the view actively, for instance by using the Observer pattern [Gamma95]. An example for the MVC 2 control flow is given at the end of this section. As already mentioned in the last section, by using the MVC pattern, the Spring Web framework allows for a high degree of decoupling between the model, view and controller parts.

The Tomcat Web container is configured for using the Spring Web framework by the following code lines in the configuration file *web.xml* of a Web application.

```
<web-app>
```

```
  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>*.uwe</url-pattern>
  </servlet-mapping>
</web-app>
```

The meaning of the entries in the configuration file is that for each request with a URL which ends with *.uwe* the request is dispatched to an instance of the *DispatcherServlet* from the Spring framework as illustrated in Figure 73. When the user of a Web application enters a URL in the Web browser then a HTTP request is sent to the Web server, i.e. the Tomcat Web container. This request is decoded and a corresponding *HttpServletRequest* object *request* is instantiated which can be used to query the decoded parts of the HTTP request, such as for example the parameters of the request. Additionally, a *HttpServletResponse* object *response* is instantiated that has to be used for returning the code of a Web page that should be displayed to the user. See [J2EE] for details about the request and response classes. If the Web server encounters a URL with the ending *.uwe* then the request is dispatched to the corresponding *DispatcherServlet* from the Spring framework by calling the *doService* method. After handling the request within the Spring Web framework (see below) the resulting Web page is displayed to the user.

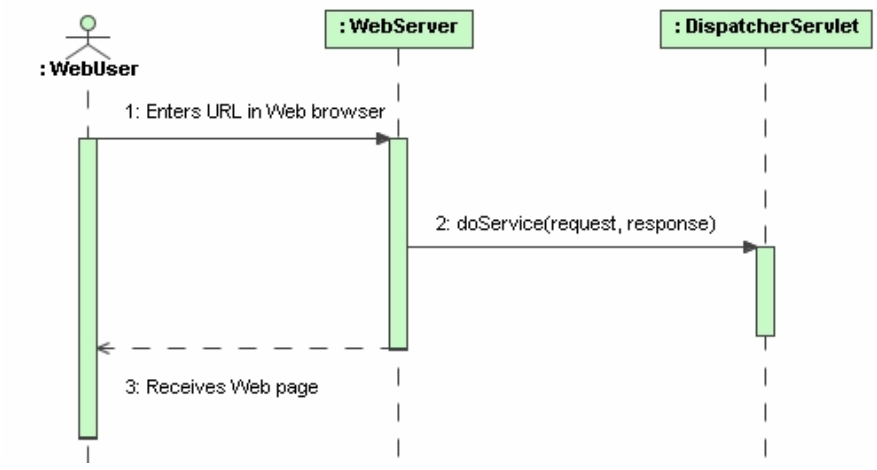


Figure 73. Dispatching of a Web request to the *DispatcherServlet*

In the following, the roles of the model, view and controller parts within the generic platform are presented. Additionally, the employed abstraction technique for the communication between the parts is discussed, which allows to decouple the parts (including the transformations) for the corresponding concerns from each other. While the generic platform is flexible with respect to the concrete technology for the model and the view parts, the controller part is predefined as presented in the next section.

Model: Minimal requirements are imposed on the target technology for transforming the content model to a platform specific implementation. It is not required that any specific superclasses or superinterfaces are extended or implemented, just any kind of Java objects can be used for the model, i.e. Plain Old Java Objects (POJO) [Spring]. The access to the model from the view or the controller parts should only rely on calling the *get*- and *set*-methods corresponding to the properties in the content model. These requirements are fulfilled by most technologies and therefore the model technology is exchangeable to a high degree. In the worst case appropriate proxy classes would have to be generated. Those principles also apply if a database should be used for the persistence of the content objects, i.e. a mapping would have to be defined between POJOs and the database. The easiest way to achieve this is to use the database mapping of Enterprise Java Beans (EJB) [J2EE] or any other database mapping technology.

The Tomcat Web container allows the passing of named variables of arbitrary type between the controller and the view. The main controller from the runtime environment (presented in the next section) provides the content object (i.e. the model) of the current Web page (i.e. the view) to be displayed in a variable with the name *self*. As stated above, a content object can be of arbitrary type, it only has to provide *get*- and *set*- methods for accessing its properties. The most convenient way for implementing the access to content objects from the view is to use the unified expression language, as for example available for Java Server Pages, see [J2EE]. For example, the expression *self.projects* within the Web page for the project manager is resolved to the list of projects by calling the *getProjects* method on the project manager content object. For more detailed examples see 5.6 and 6.2.4.1.

View: The Spring Web framework provides a mechanism for decoupling the concrete view technology, i.e. the target technology for transforming the presentation model to a platform specific implementation, from the controller part (the model part does not depend from the view technology anyway). This allows for example the use of the following view technologies:

- Java Server Pages (JSP)
- Tiles (based on Struts)

- Velocity and Freemarker (template languages)
- XML + XSLT
- Document views (e.g. PDF or excel)
- Jasper reports (report engine)
- Portlets
- JavaServer Faces

Different view technologies can even be combined. Additionally, the Spring framework allows the integration in other Web frameworks such as Struts or JavaServer faces. The following code lines in the configuration file for the dispatcher servlet demonstrate how the framework is configured to use Java Server Pages (JSP) and the Java Standard Tag Library (JSTL) view technology. Within the controller part a concrete view is referenced only by a name. This view name is resolved to a Web page by the Spring framework, using the configuration information. For example, the concrete view name *ProjectManager* would be resolved to the JSP page */WEB-INF/jsp/ProjectManager.jsp*.

```
<bean id="viewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass">
        <value>org.springframework.web.servlet.view.JstlView</value>
    </property>
    <property name="prefix"><value>/WEB-INF/jsp/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>
```

The other way round, URLs embedded in Web pages are used for the communication between the view and the controller. The URL must have the suffix *.uwe* preceded by the name of a node from the navigation model for navigation purposes, such as for example *ProjectManager.uwe*.

Controller: A controller in the Spring Web framework is a Java class that implements the interface *Controller*. An outline to the general control flow for handling a Web request within the Spring framework, including the model, view and controller parts, is illustrated by the sequence diagram depicted in Figure 74. As explained above and illustrated in Figure 73, a Web request is handled within the *doService* method of the dispatcher servlet from the Spring framework. This request is then further delegated to a controller imple-

mentation by calling the method *handleRequest*. Here, the class *MainController* presented in the next section serves to illustrate the basic control flow within a controller. The method *handleRequest* has to return an object of type *ModelAndView*, which is a utility class used to return both the model and the view instances in a single return value. Note that the term model as used here refers to the data required for the presentation of a single Web page, i.e. a single content object. The model is implemented by a map, which contains exactly one entry *self* that holds a reference to the current content object. In the example the content object *rootObject,* which represents the entry point of the application as explained in the next section is put in the map. Then a *ModelAndView* object is constructed with the name of the view to be displayed, in the example the view *ProjectManager*, and a reference to the model map. After the method call has returned, the dispatcher servlet calls its method *render* to render the resulting Web page. Within this method the call is further delegated to the concrete view implementation which receives a reference to the model map. The view implementation, in the example a *JstlView* for rendering Java Server Pages, retrieves the content object to be displayed from the model map and renders the Web page.

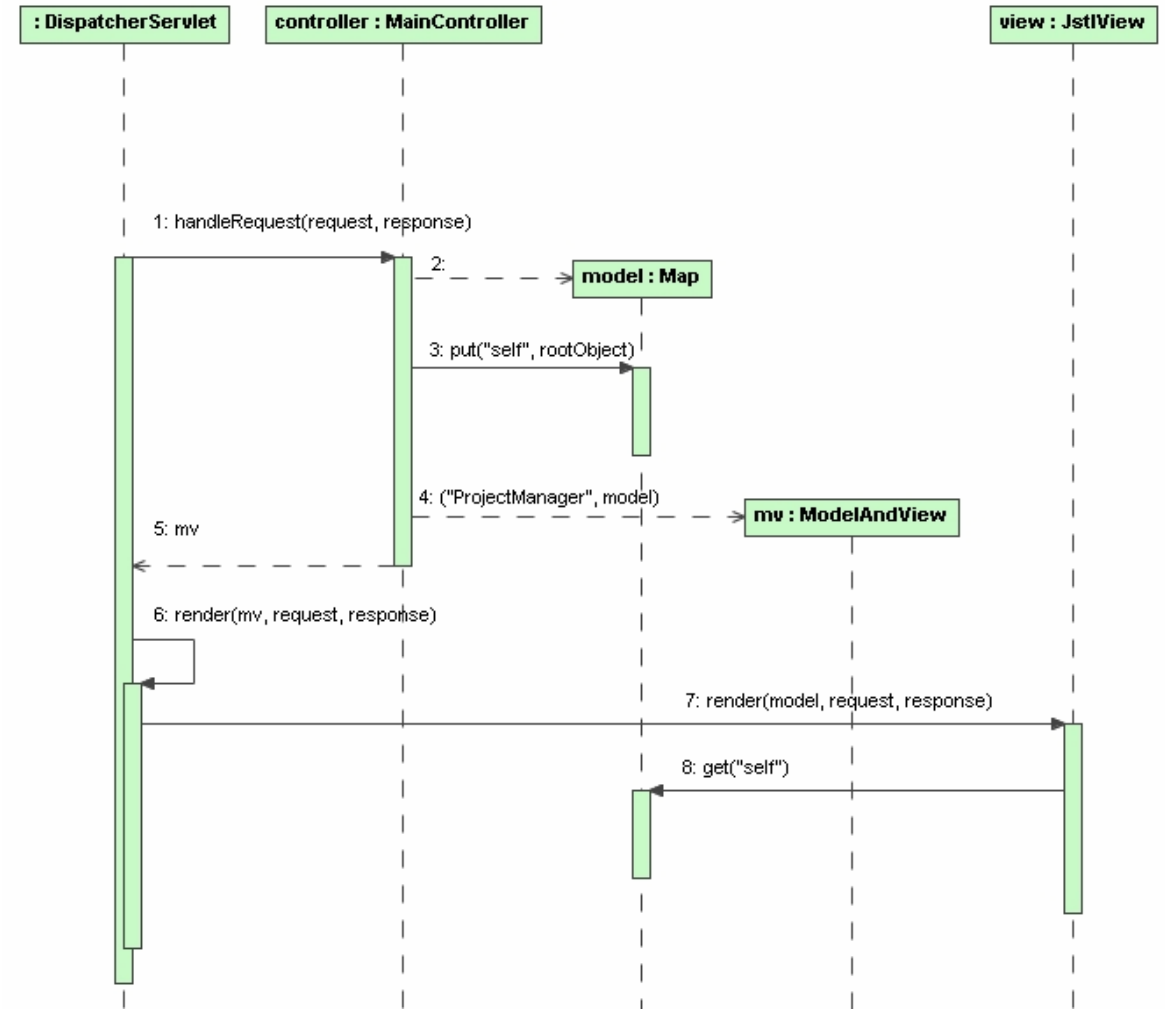


Figure 74. Handling of a Web request within the Spring Web framework

5.1.2 Runtime Environment

The Spring framework is configured to use a specific generic controller implementation named *MainController*. The corresponding configuration technique is described in the next section. Generic means that the same controller can be used for all applications generated by following this approach. The runtime environment developed for the generic platform comprises this controller and all associated classes. It is kept as simple as possible as can be seen in Figure 75. For a specific application the controller is configured to use the artifacts generated for the navigation and the process models as described in the next section.

The main controller has access to one designated root content object that represents the entry point of a Web application. The use of the type *Object,* which is the root type in the Java class hierarchy, indicates that this approach is generic in reference to the concrete

types. Model objects are accessed by calling their *get*- and *set*- methods and the operations defined in the content model. Additionally, the controller manages a set of *NavigationClassInfo* objects which contain information about the navigation structure regarding inheritance between navigation classes, which is required for resolving dynamic navigation. The corresponding configuration data that is used to instantiate these objects is generated from the navigation model as presented in 5.4. A set of *ProcessActivity* objects represents the available Web processes. Similarly, the corresponding configuration data that is used to instantiate these objects is generated from the process model as presented in 5.5.

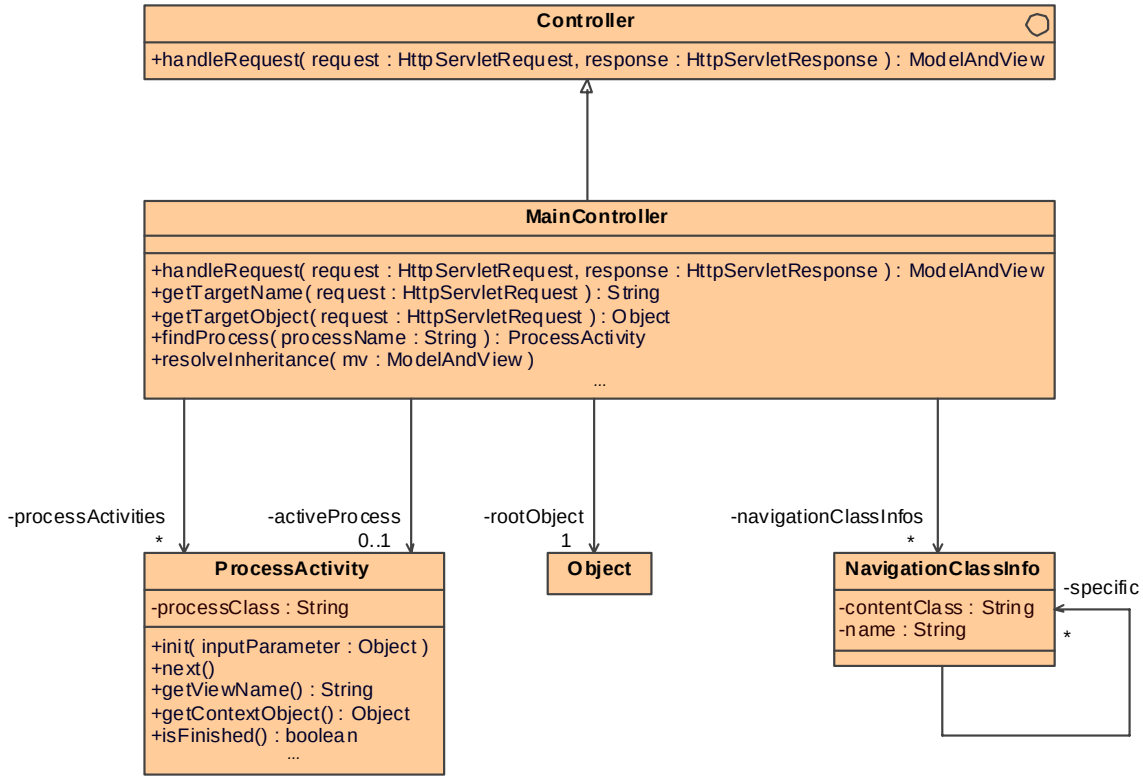


Figure 75. Runtime environment

An outline to the general control flow for handling a Web request within the Spring framework was already given in the last section. The basic control flow within the method *handleRequest* of a controller as depicted in Figure 74 is further refined by the main controller of the runtime environment as illustrated in Figure 76. First, a local variable *targetName* is initialized with the navigation target represented by the Web request which equals to the name of a node from the navigation model, for example *Project*. Another local variable *targetObj* is initialized with the corresponding target content object.

Then, if no process is currently active, it is checked if the navigation target equals to the name of a process class associated to one of the available process activities by calling the method *findProcess*. If a corresponding process exists, then this process is started by calling the method *init* (further details are presented in 5.5.1).

Afterwards, if a process is currently active, which includes the case that it has just been started, then the next step of the process is executed by calling the method *next* on the corresponding process activity (again, further details are presented in 5.5.1). Following, the view name to be displayed and the corresponding context object is queried from the process activity. In the case that the process is finished after execution of the next step, then the association to the currently active process is removed.

Otherwise, if no process is currently active, then the view name to be displayed is determined by the navigation target derived from the Web request. The same holds for the target content object. Finally, in the same way as described in the last section a *ModelAndView* object is constructed. Before this object is returned to the dispatcher servlet, inheritance between navigation classes is resolved by calling the method *resolveViewInheritance*. Within this method the set of *NavigationClassInfo* objects is searched for the most specialized navigation subclass which is compatible with the actual content object type. Compatible means that the content class type associated to the navigation class is type compatible with the content object.

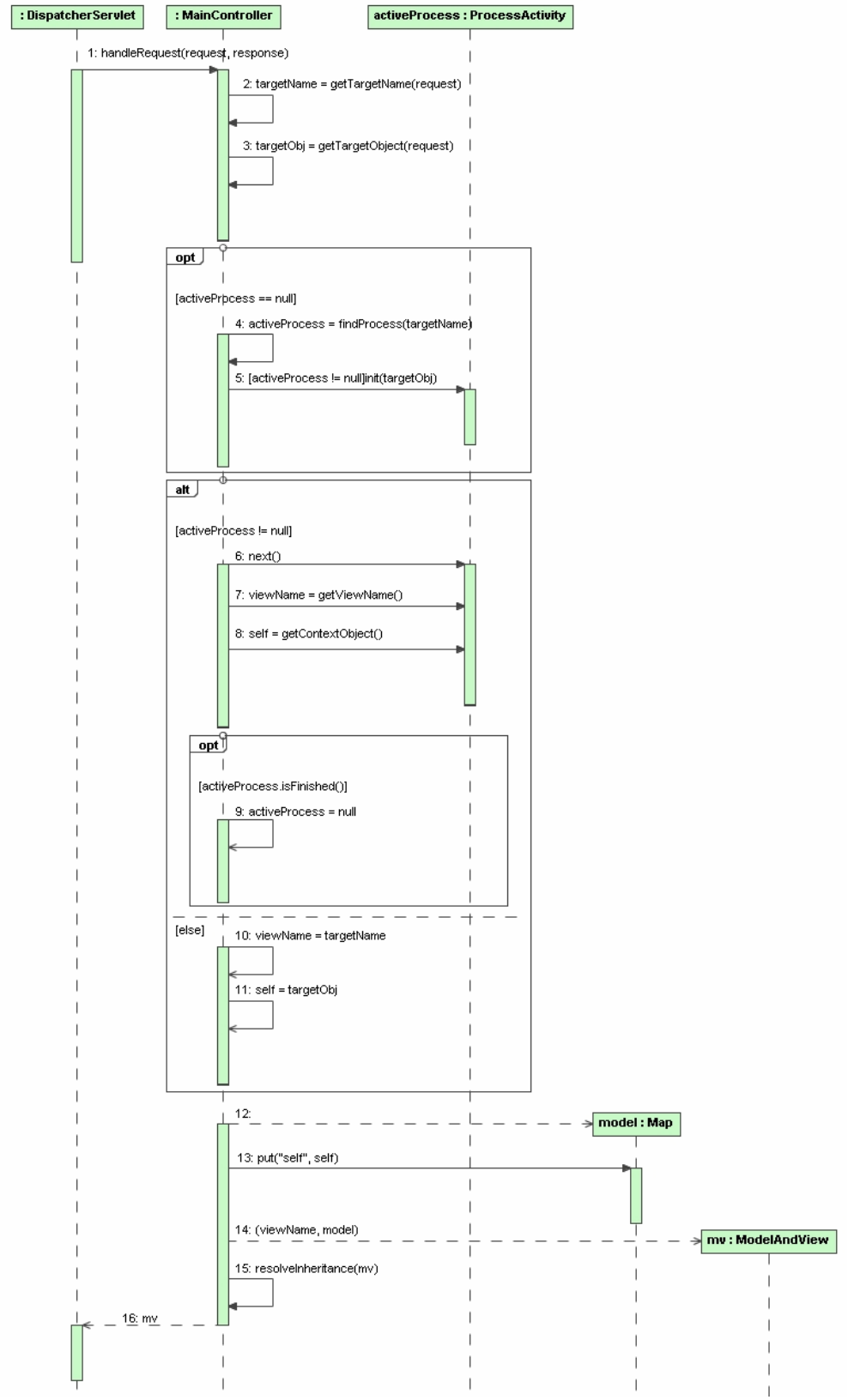


Figure 76. Control flow within the runtime environment

5.1.3 Configuration

The Spring framework provides a simple but powerful configuration mechanism based on the Inversion of Control (IoC) or Dependency Injection (DI) principle [Fowler04a]. The Spring IoC container provides the functionality to instantiate, assemble and manage the objects of a Spring application, i.e. an application that uses the Spring framework. Those objects which are managed by the IoC container can be of arbitrary type and are called beans or Plain Old Java Objects (POJOs). The IoC container, also called bean factory, is initialized by reading an XML bean definition document which comprises the definition of the beans of the application and the dependencies between them. The XML format for bean definitions is well defined by a corresponding DTD[11]. As already stated, the type of a bean can be arbitrary as for example data access objects (DAO) or other infrastructure objects to access databases. Most important, the Spring framework itself uses the bean factory mechanism for the configuration of its modules, for instance for enabling the use of aspect oriented techniques using AspectJ.

In this approach beans will be used for the configuration of the runtime environment, including data about the navigation and the process concerns. The following XML bean definition document demonstrates how a Web application using the proposed generic platform is configured. This document with the name *dispatcher-servlet.xml* is read by the dispatcher servlet of the Spring framework, which is responsible for handling Web requests delegated by the Tomcat Web container as described in 5.1.1.

```
<beans>
    <import resource="content-conf.xml" />
    <import resource="navigation-conf.xml" />
    <import resource="process-conf.xml" />

    <bean id="urlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/*.uwe">mainController</prop>
            </props>
        </property>
    </bean>
```

[11] The Spring bean definition DTD can be found at http://www.springframework.org/dtd/spring-beans.dtd

```
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass">
            <value>org.springframework.web.servlet.view.JstlView</value>
        </property>
        <property name="prefix"><value>/WEB-INF/jsp/</value></property>
        <property name="suffix"><value>.jsp</value></property>
    </bean>

    <bean id="mainController" class="uwe.runtime.MainController">
       <property name="rootObject">
          <ref bean="rootObject"/>
        </property>
        <property name="processActivities">
            <bean id="processActivities.list"
                class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
        </property>
        <property name="navigationClassInfos">
            <bean id="navigationClassInfos.list"
                class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
        </property>
    </bean>
</beans>
```

All bean definitions have to be enclosed by a *beans* tag. A bean definition starts with the *bean* tag followed by a list of *property* tags for the properties of a bean. The *ref* tag is used to reference a bean from another bean by means of an identifier defined by the *id* attribute of the *bean* tag. The *class* attribute indicates which class should be instantiated. Note that the Spring framework provides much more possibilities for working with beans, see [Spring].

The global bean definition for the configuration of a Web application is spread over several bean definition files which are imported by using the *import* tag within the main bean definition file read by the dispatcher servlet of the Spring framework. The main bean definition file comprises the application independent part of the configuration, including the configuration of the view technology. The imported bean definition files comprise the application dependent parts of the configuration. The bean definition file *content-conf.xml* has to be provided manually and it comprises the configuration of the content part of the Web application. The only requirement is that a bean with the id *rootObject* is defined within this file that serves as the entry point of the application. The other two imported bean definition files *navigation-conf.xml* and *process-conf.xml* represent the navigation and process parts

of a Web application. These files are generated automatically as presented in 5.4 and 5.5 and should not be modified manually.

The dispatcher servlet from the Spring framework uses two specific beans with the ids *urlMapping* and *viewResolver* for its configuration. The bean with the id *urlMapping* is used to map URLs that the user has entered in the browser to controller implementations. Exactly one URL mapping is defined for mapping each URL with the ending *.uwe* to a controller bean with the id *mainController* (see below). On the other hand, and as already described in 5.1.1, the bean with the id *viewResolver* defines which concrete view technology should be used for the application. In the example configuration Java Server Pages (JSP) and the Java Standard Tag Library (JSTL) should be used as view technology.

Finally, the main bean definition file also comprises the configuration of the main controller from the runtime environment presented in the last section. The property *rootObject* is set to the bean with the id *rootObject* which should be defined in the file *content-conf.xml* and which represents the entry point content object of the application. The property *navigationClassInfos* is set to the list of navigation class info objects defined in the file *navigation-conf.xml* by using the *PropertyPathFactoryBean* (for the technical details see [Spring]). In the same way the property *processActivities* is set to the list of process activities defined in the file *process-conf.xml*.

The instantiated objects for the example bean definition file listed above are illustrated in Figure 77. The dispatcher servlet from the Spring framework is linked with an object for the URL mapping and with another object for the concrete view technology. For further technical details about the dispatcher servlet and its configuration see [Spring]. Most important, the dispatcher servlet is also linked with an instance of the main controller which receives all Web requests determined by the URL mapping. The main controller object has a link to the root object, i.e. the entry point content object of the application. It is further linked with the set of the application specific navigation class info objects and with the set of the application specific process activity objects.

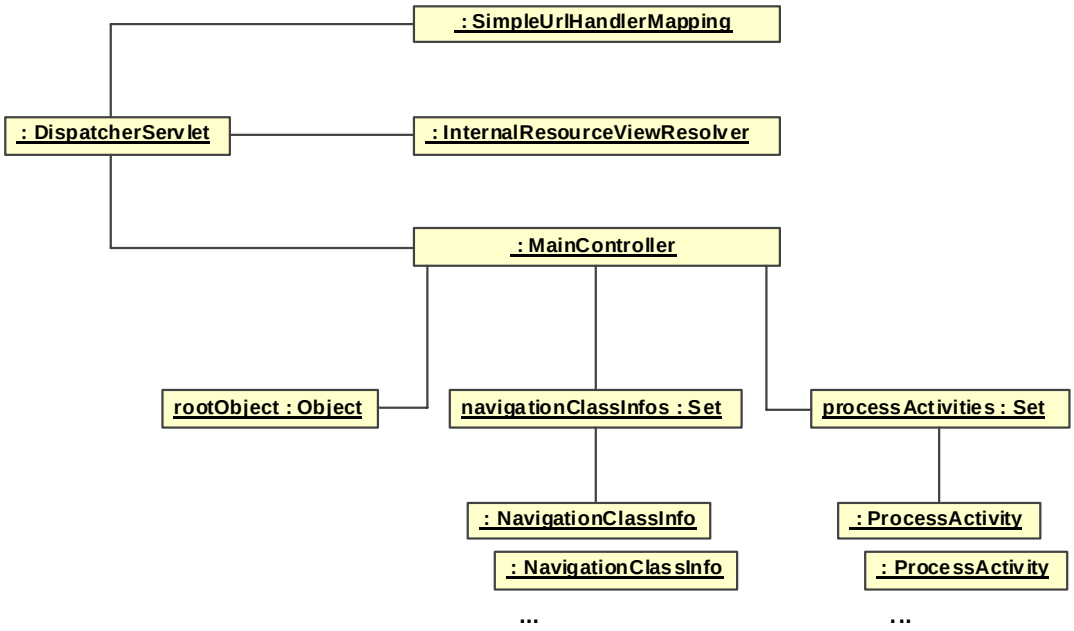


Figure 77. Example configuration of the runtime environment

In the following subsections the general technique for generating configuration data for the runtime environment is presented, which is used for generating the bean definition files *navigation-conf.xml* and *process-conf.xml* from the platform independent models. Therefore, first a metamodel for XML is defined. This is followed by the definition of a set of rules that are used in the transformations *Navigation2Conf* and *Process2Conf* presented in 5.4 and 5.5, respectively. Finally, an ATL query for the serialization of an XML model to an XML document, i.e. code, is presented.

5.1.3.1 XML Metamodel

A metamodel for simple XML documents is depicted in Figure 78. Some details of XML not required in this work, such as for example processing instructions, have been omitted. The root element in the inheritance hierarchy for all elements is the abstract class *Node*. Each node has a name and a value. The concrete class *Element* represents an XML tag which can contain other nodes, thus it is used for nesting nodes. The special element *Root* is the root node of a node hierarchy. Each root node represents an XML document, thus an XML model represents a set of XML documents. The attribute *documentName* is used for writing a node hierarchy to a file as explained in 5.1.3.3. An attribute node represents attributes of an XML tag and a text node arbitrary text (XML CDATA) nested within a tag.

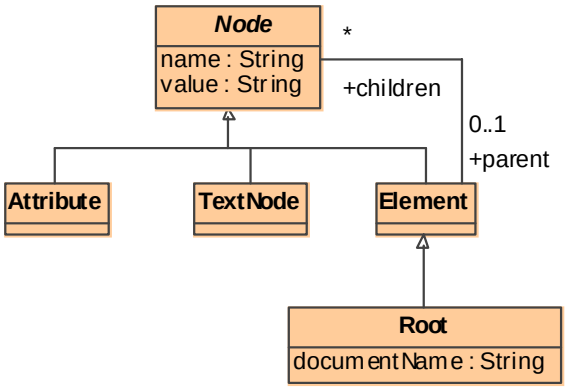


Figure 78. XML metamodel

Constraints

Only root nodes have no parent node and root nodes may not be nested within other elements

```
context Node inv RootParent :
    self.parent->isEmpty() implies self.oclIsTypeOf( Root ) and
    self.oclIsKindOf( Root ) implies self.parent->isEmpty()
```

5.1.3.2 Transformation Rules

The transformation rules presented in this section comprise an abstract matched rule *NamedElement2Conf* which is specialized by sub rules in 5.4 and 5.5 in order to generate an XML node that represents a bean entry in a configuration file. Further, a set of called rules is used to generate subordinated property entries. For the detailed description of the rules see B.4.1. Note that the rules of this section cannot be used stand-alone because neither abstract matched rules nor called rules are triggered automatically. An example of using these rules is given in 5.4.2 and 5.5.3.

Rule NamedElement2Conf

This abstract rule maps a named element from the metamodel for platform independent analysis and design to a bean XML node. The *id* attribute of the bean derived by the *getId* helper is either automatically derived from the qualified name of the element or, if a qualified name is not available, automatically generated by using a global id counter. The resulting id is stored in a global map that is used for resolving references between elements to references between beans when calling the rule *CreateConfPropertyRefValue*.

```
abstract rule NamedElement2Conf
{
    from el : UWE!NamedElement
    to beanEl : XML!Element
    (
        name <- 'bean'
    ),
    idAttr : XML!Attribute
    (
        name <- 'id',
        value <- el.getId(),
        parent <- beanEl
    )
}
```

Rule CreateConfProperty

This rule is called to create a configuration *property* XML node for a given *name* and *value* to be represented. The call is further delegated to the called rule *CreateConfPropertyValue* to generate the contained *value* XML node.

```
rule CreateConfProperty( parent : XML!Element, name : String, value : OclAny )
{
    to propertyEl : XML!Element
    (
        name <- 'property',
        parent <- parent
    ),
    nameAttr : XML!Attribute
    (
        name <- 'name',
        value <- name,
        parent <- propertyEl
    )
    do
    {
        thisModule.CreateConfPropertyValue( propertyEl, value );
    }
}
```

Rule CreateConfPropertyValue

This rule is called to create a *value* XML node for a given *value* to be represented. The call is further delegated to different called rules presented in the following depending on the type of the value allowing thereby even the handling of nested values, such as lists of lists of values.

```
rule CreateConfPropertyValue( parent : XML!Element, value : OclAny )
{
    do
    {
        if( value.isPrimitive() )
        {
            thisModule.CreateConfPropertyPrimitiveValue( parent, value );
        }
        else
        {
            if( value.oclIsKindOf( UWE!NamedElement ) )
            {
                thisModule.CreateConfPropertyRefValue( parent, value );
            }
            else
            {
                if( value.oclIsKindOf( Set( OclAny ) ) )
                {
                    thisModule.CreateConfPropertySetValue( parent, value );
                }
                else
                {
                    if( value.oclIsKindOf( Sequence( OclAny ) ) )
                    {
                        thisModule.CreateConfPropertySequenceValue( parent, value );
                    }
                    else
                    {
                        value.debug( 'Property value cannot be converted to conf' );
                    }
                }
            }
        }
    }
}
```

Rule CreateConfPropertyPrimitiveValue

This rule is called for creating the XML representation of primitive values such as numbers or strings.

```
rule CreateConfPropertyPrimitiveValue( parent : XML!Element, value : OclAny )
{
    to valueEl : XML!Element
    (
        name <- 'value',
        parent <- parent
    ),
    stringValue : XML!TextNode
    (
        value <- value.toString(),
        parent <- valueEl
    )
}
```

Rule CreateConfPropertyRefValue

This rule is called for creating the XML representation of reference values to other model elements. Therefore the reference ids are used that were put in the global map by the helper *getId*.

```
rule CreateConfPropertyRefValue( parent : XML!Element, value : UWE!NamedElement )
{
    to refEl : XML!Element
    (
        name <- 'ref',
        parent <- parent
    ),
    beanAttr : XML!Attribute
    (
        name <- 'bean',
        value <- value.getId(),
        parent <- refEl
    )
}
```

Rule CreateConfPropertySetValue

This rule is called for creating the XML representation of sets of elements. Therefore the rule *CreateConfPropertyValue* is called for each element in the set.

```
rule CreateConfPropertySetValue( parent : XML!Element, value : Set( OclAny ) )
{
    to setEl : XML!Element
    (
        name <- 'set',
        parent <- parent
    )
    do
    {
        for( v in value )
        {
            thisModule.CreateConfPropertyValue( setEl, v );
        }
    }
}
```

Rule CreateConfPropertySequenceValue

This rule is called for creating the XML representation of sequences of elements. Therefore the rule *CreateConfPropertyValue* is called for each element in the sequence.

```
rule CreateConfPropertySequenceValue( parent : XML!Element, value : Sequence( OclAny ) )
{
    to listEl : XML!Element
    (
        name <- 'list',
        parent <- parent
    )
    do
    {
        for( v in value )
        {
            thisModule.CreateConfPropertyValue( listEl, v );
        }
    }
}
```

5.1.3.3 Serialization to Code

An XML model is transformed to executable code (i.e. an XML document) with the ATL query *XML2Code* listed below (cf. 2.3.3.2). This is done by calling the helper method *toCode* on all children of root elements, concatenating the results and writing it to a file given by the attribute *documentName* of the root element. For writing strings to a file the predefined method *writeTo* of type String is used.

```
query XML2Code = XML!Root.allInstances()->collect( n | n.getChildren()->
    iterate( n; acc : String = '' | acc + n.toCode() ).writeTo( n.documentName ) );

helper context XML!Element def : getAttributes() : Sequence( XML!Attribute ) =
    self.children->select( cn | cn.oclIsKindOf( XML!Attribute ) );

helper context XML!Element def : getChildren() : Sequence( XML!Node ) =
    self.children->select( cn | not cn.oclIsKindOf( XML!Attribute ) );

helper context XML!Element def : toCode() : String =
    '<' + self.name + self.getAttributes()->iterate( n; acc : String = '' | acc + ' ' + n.name + '=\"' +
    n.value + '\"' ) + '>\n' + self.getChildren()->iterate( n; acc : String = '' | acc + n.toCode() )
    + '</' + self.name + '>\n';

helper context XML!TextNode def : toCode() : String =
    self.value;
```

5.2 Content via JavaBeans

This section presents the first of the two investigated alternatives for the transformation of the content model to the platform specific implementation for the platform as described in the previous section using JavaBeans.

JavaBeans [Sun06a] are lightweight software components for the programming language Java. The development initially stemmed from the need for a simple way to instantiate and transfer (desktop) GUI components for the use in GUI builders. In this work not all features from the JavaBeans specification are needed.

JavaBeans are essentially Java classes, or Plain Old Java Objects (POJOs), that are subject to certain constraints [Sun06a]: all fields should have private visibility and be accessible only by public getters and setters, e.g. for a field named *x* the corresponding getter has to be named *getX* and the setter *setX*; a public default constructor must be provided; and the

class has to be serializable, thus enabling persistence and data transfer technologies. Further general properties of JavaBeans are not in the scope of this work, for more details see [Sun06a]. Within the context of this work JavaBeans are used as a simple yet powerful implementation technology for the content model. The Spring framework provides a broad support for using JavaBeans. A *BeanFactory* instance (provided by the Spring framework) is responsible for instantiating, configuring and managing a number of beans. This includes resolving dependencies (i.e. associations) between beans. Additionally, persistence techniques are provided. The simplest way to handle beans is the XML variant of the *BeanFactory* which allows reading them from an XML document.

JavaBeans allow for fast prototyping and testing of a Web application and in some cases may even be a sufficient "component technology". In the case of DANUBIA it fulfils the special requirements for the component technology as discussed in 6.2.1. The example bean definition listed in the following is stored in the configuration file *content-conf.xml* for the configuration of the content part within the runtime environment configuration as discussed in 5.1.3. A designated bean with the id *rootObject*, i.e. an instance of the project manager, represents the entry point of the application.

```
<beans>
   …
  <bean id="rootObject" class="ProjectManager">
      <property name="projects">
         <list>
            <ref bean="project1"/>
             …
         </list>
       </property>
  </bean>

  <bean id="project1" class="UserProject">
      <property name="id"><value>1</value></property>
      <property name="name"><value>Project 1</value></property>
      <property name="description"><value>Description of project 1</value></property>
      <property name="scenarios">
         <list>
            <ref bean="scenario1"/>
             …
         </list>
       </property>
      …
  </bean>
   …
```

```
  <bean id="scenario1" class="Scenario">
     <property name="id"><value>1</value></property>
     <property name="name"><value>Scenario 1</value></property>
     <property name="description"><value>Description of scenario 1</value></property>
      …
  </bean>
   …
</beans>
```

In the following, first a metamodel for Java is presented that is used for both Java-based transformation alternatives. This is followed by an example and the description of the transformation from the content model to JavaBeans. Finally, the serialization to Java code is presented. The resulting Java code has to be manually refined for all content classes that contain at least one operation.

5.2.1 Java Metamodel

A metamodel for Java that covers all features of Java needed by the transformations in this work is depicted in Figure 79. All classes of the Java metamodel are organized in an inheritance hierarchy with the class *JavaElement* as root element. Java classes, primitive types and enumerations are distinguished. Java classes and enumerations are assigned to a package which may be declared as imported. Imported packages are not serialized to code, see 5.2.4. Java interfaces are represented by Java classes with the attribute *isInterface* set to true. Java classes (and interfaces) can be organized in an inheritance hierarchy and they can be parameterized to support Java 1.5 Generics [Mahmoud04]. Members of a Java class are either methods or fields. Methods support an ordered list of parameters and can throw exceptions.

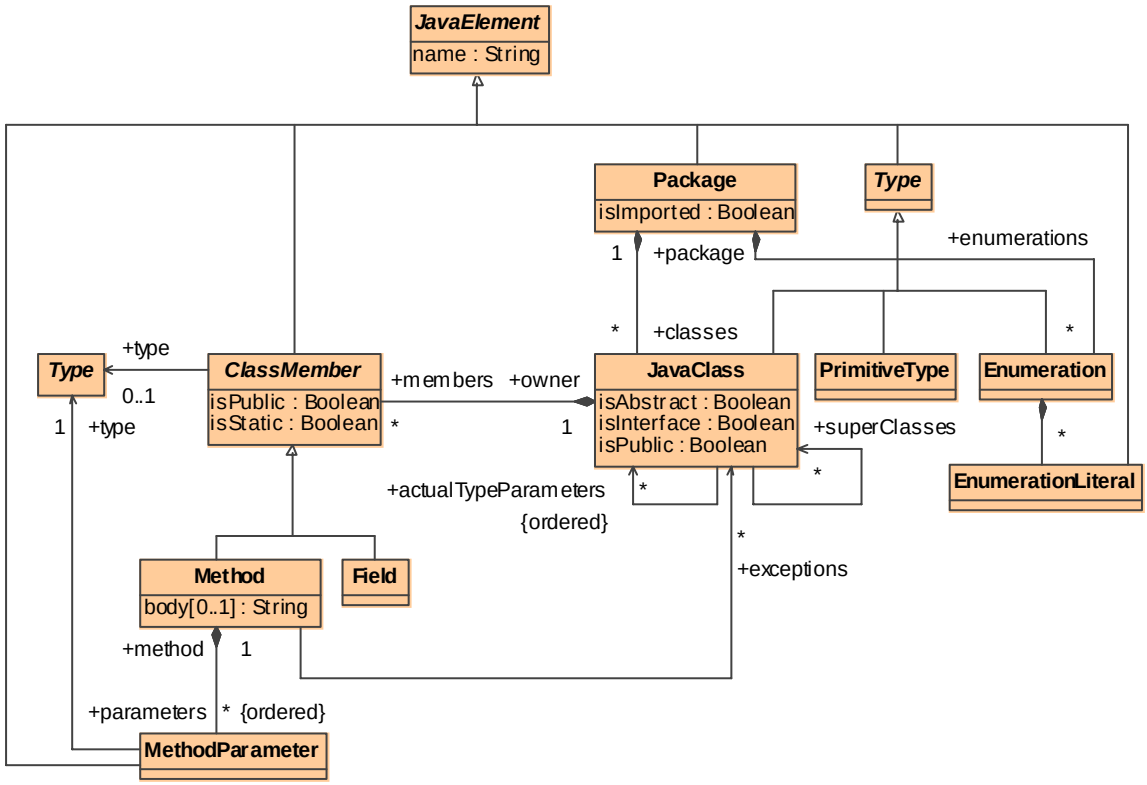


Figure 79. Java metamodel

Constraints

An interface must not contain fields and must not define a method body.

```
context JavaClass inv InterfaceMembersAndBody :
    self.isInterface implies self.members->forAll( m |
        not m.oclIsTypeOf(Field ) and
        m.oclIsTypeOf( Method ) implies m.body->isEmpty() )
```

An interface can only have super interfaces.

```
context JavaClass inv InterfaceSuperClasses :
    self.isInterface implies self.superClasses->forAll( sc | sc.isInterface )
```

A class can have at most one super class, but may implement several interfaces.

```
context JavaClass inv ClassSuperClasses :
    not self.isInterface implies self.superClasses->select( sc | not sc.isInterface )->size() <= 1
```

A field must have a type.

```
context Field inv FieldType : self.type->notEmpty()
```

5.2.2 Example

The following code listing shows the JavaBean code generated for the content class *ProjectManager* by the transformation presented in the next section after serialization to code. The property *projects* and the corresponding getter and setter methods are already fully implemented. The body of the other operations used by the Web processes has to be completed by the developer. For a more detailed example see also 6.2.1.1.

```
public class ProjectManager
{
    private List<Project> projects;

    public List<Project> getProjects()
    {
        return projects;
    }

    public void setProjects( List<Project> projects )
    {
        this.projects = projects;
    }

    public UserProject addUserProject( UserProject userProject )
    {
        // to be implemented manually
    }

    public ValidationProject addValidationProject( ValidationProject validationProject )
    {
        // to be implemented manually
    }

    public void removeProject( Project project )
    {
        // to be implemented manually
    }

}
```

5.2.3 Transformation Content2JavaBeans

The transformation *Content2JavaBeans* depicted in Figure 80 maps the content model to a Java model for JavaBeans, which is used for the case study as described in 6.2.1. It comprises four transformation rules which are outlined below and detailed in B.4.2. This re-

sembles the code generation facilities that are provided by most UML CASE tools. But, in contrast to the flexible and completely customizable approach presented here, transformations implemented in CASE tools are usually hard-coded.

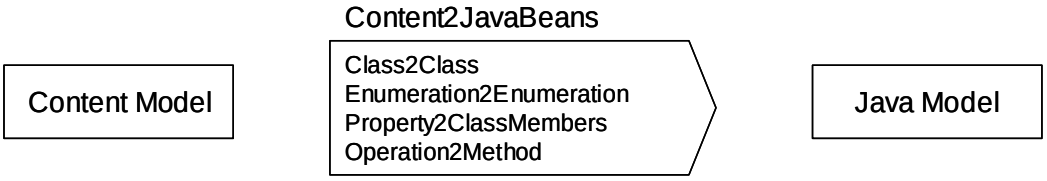


Figure 80. Transformation *Content2JavaBeans*

The downside of using the JavaBeans code resulting from this transformation is that the bodies of the operations (except for getter and setter methods) still have to be completed by the developer and that these modifications in the source code are not preserved upon re-generation from a modified content model.

Rule Class2Class

Each content class from the content model is mapped to a JavaBean class. The superclass relationship in the source model is mapped to a corresponding superclass relationship in the target model.

```
rule Class2Class
{
    from c : UWE!Class ( c.oclIsTypeOf( UWE!Class ) )
    to jc : JAVA!JavaClass
    (
        name <- c.name,
        package <- c.package,
        superClasses <- c.generalization->collect( g | g.general ),
        isAbstract <- false,
        isPublic <- true,
        isInterface <- false
    )
}
```

Rule Enumeration2Enumeration

Each enumeration with its enumeration literals is mapped to a corresponding Java enumeration by the rules *Enumeration2Enumeration* and *EnumerationLiteral2EnumerationLiteral*.

```
rule Enumeration2Enumeration
{
    from e : UWE!Enumeration
    to je : JAVA!Enumeration
    (
        name <- e.name,
        package <- e.package,
        enumerationLiterals <- e.ownedLiteral->collect( el |
        thisModule.EnumerationLiteral2EnumerationLiteral( el ) )
    )
}

lazy rule EnumerationLiteral2EnumerationLiteral
{
    from el : UWE!EnumerationLiteral
    to jel : JAVA!EnumerationLiteral
    (
        name <- el.name
    )
}
```

Rule Property2ClassMembers

Each content property owned by a content class is mapped to a corresponding Java field and getter and setter methods for that field. The (trivial) code for the method body is generated by string concatenation. Properties owned by a class comprise attributes as well as owned association ends. Multi-valued properties are mapped to the parameterized Java collection interfaces *java.util.List<E>* for ordered properties and *java.util.Set<E>* for unordered properties by the unique lazy rules *Class2ParameterizedList* and *Class2ParameterizedSet*, respectively. Note that derived properties are mapped by the rule *DerivedProperty2ClassMembers*.

```
rule Property2ClassMembers
{
    from p : UWE!Property ( p.class_.oclIsTypeOf( UWE!Class ) and
        ( p.type.oclIsKindOf( UWE!DataType ) or p.type.oclIsTypeOf( UWE!Class ) ) and
```

```
        not p.isDerived )
    to field : JAVA!Field
    (
        owner <- p.class_,
        name <- '_' + p.name,
        type <- if p.isMultivalued() then
                if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
                thisModule.Class2ParameterizedSet( p.type ) endif
            else p.type endif,
        isPublic <- false,
        isStatic <- false,
        initializer <- …
    ),
    getter : JAVA!Method
    (
        owner <- p.class_,
        name <- 'get' + p.name.stringFirstToUpper(),
        type <- if p.isMultivalued() then
                if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
                thisModule.Class2ParameterizedSet( p.type ) endif
            else p.type endif,
        isPublic <- true,
        isStatic <- false,
        body <- 'return ' + '_' + p.name + ';'
    ),
    setter : JAVA!Method
    (
        owner <- p.class_,
        name <- 'set' + p.name.stringFirstToUpper(),
        isPublic <- true,
        isStatic <- false,
        parameters <- Sequence { setterParameter },
        body <- 'this.' + '_' + p.name + ' = ' + '_' + p.name + ';'
    ),
    setterParameter : JAVA!MethodParameter
    (
        name <- '_' + p.name,
        type <- if p.isMultivalued() then
                if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
                thisModule.Class2ParameterizedSet( p.type ) endif
            else p.type endif
    )
}
```

Rule Operation2Method

Each operation in the content model is mapped to Java method of the corresponding class. A default method body is generated so that the generated class can be compiled by the Java compiler.

```
rule Operation2Method
{
    from o : UWE!Operation ( o.class_.oclIsTypeOf( UWE!Class ) )
    using
    {
        formalParameters : Sequence ( UWE!Parameter ) = o.ownedParameter->select( op |
            op.direction <> #return );
    }
    to m : JAVA!Method
    (
        name <- o.name,
        owner <- o.class_,
        isPublic <- true,
        isStatic <- false,
        parameters <- parameters,
        type <- i o.type,
        body <- if o.type.oclIsUndefined() then '' else
            if o.type.oclIsKindOf( UWE!DataType ) then
                if o.type.name = 'void' then '' else
                    if o.type.name = 'Boolean' then 'return false;' else
                        'return (' + o.type.name + ')0;'
                    endif
                endif
            else
                'return null;'
            endif
        endif
    ),
    parameters : distinct JAVA!MethodParameter foreach ( p in formalParameters )
    (
        name <- '_' + p.name,
        type <- p.type
    )
}
```

Rule Class2ParameterizedSet

This unique lazy rule is explicitly invoked (hence lazy) from other matched rules and always returns the same (hence unique) Java set class parameterized by the given content class type.

```
unique lazy rule Class2ParameterizedSet
{
    from c : UWE!Class
    to s : JAVA!JavaClass
    (
        name <- 'Set',
        package <- thisModule.utilPck,
        isAbstract <- false,
        isPublic <- true,
        isInterface <- true,
        actualTypeParameters <- Sequence { c }
    )
}
```

Rule Class2ParameterizedList

Like the previous rule but returning a parameterized Java list.

```
unique lazy rule Class2ParameterizedList
{
    from c : UWE!Class
    to s : JAVA!JavaClass
    (
        name <- 'List',
        package <- thisModule.utilPck,
        isAbstract <- false,
        isPublic <- true,
        isInterface <- true,
        actualTypeParameters <- Sequence { c }
    )
}
```

5.2.4 Serialization to Code

A Java model is transformed to executable code (i.e. text) with the ATL query *Java2Code* outlined below. This is done by calling the helper method *toString* on all Java classes and

enumerations whose package is not imported and by writing it to files given by the name of the Java class preceded by the full file path resulting from replacing all '.' characters in the name of the owning package with the file separator. For writing strings to a file the predefined method *writeTo* of the type String is used. Only the helpers for serializing classes, methods and fields are listed here for brevity. For the technical details see B.2.2.3.

```
query Java2Code = JAVA!Type.allInstances()->
    select( e | if e.oclIsTypeOf( JAVA!JavaClass ) or e.oclIsTypeOf( JAVA!Enumeration ) then
        e.package.isImported else false endif )->
        collect(x | x.toString().writeTo( 'src/' + x.package.name.replaceAll('.', '/') + '/' +
            x.name + '.java'));

helper context JAVA!JavaClass def: toString() : String =
    self.package.toString() + self.visibility() + self.modifierAbstract() +
    if self.isInterface then 'interface ' else 'class ' endif + self.name +
    self.superClasses->select( sc | not sc.isInterface or self.isInterface )->
        iterate( sc; acc : String = ''| acc + if acc='' then ' extends ' else ', ' endif + sc.fullName() ) +
    self.superClasses->select( sc | not self.isInterface and sc.isInterface )->
        iterate( sc; acc : String = ''| acc + if acc='' then ' implements ' else ', ' endif + sc.fullName() ) +
    ' {\n' +
        self.members->iterate(i; acc : String = '' | acc + i.toString()  ) +
    '\n}\n\n';

helper context JAVA!Field def: toString() : String =
    '\t' + self.visibility() + self.scope() + self.type.fullName() + ' ' + self.name + ';\n';

helper context JAVA!Method def: toString() : String =
    '\t' + self.visibility() + self.scope() +
    if self.type.oclIsUndefined() then 'void' else self.type.fullName() endif
    + ' ' + self.name + '(' +
    self.parameters->iterate( p; acc : String = '' | acc + if acc = '' then '' else ', ' endif + p.toString() )
    ')' + if self.exceptions->size() > 0 then ' throws ' +
        self.exceptions->iterate( e; acc : String = '' | acc + if acc = '' then '' else ', ' endif + e.name )
    else '' endif + if self.body.oclIsUndefined() then ';\n' else ' {\n\t\t' + self.body + '\n\t}\n' endif;

…
```

5.3 Content via RMI

In addition to using JavaBeans as described in the last section, this section presents a second alternative technology for the model driven implementation of the content concern using RMI interfaces.

RMI is a Java technology to allows the invocation of methods on remote objects, i.e. objects that reside in a different Java Virtual Machine than the caller [RMI]. The different virtual machines may reside on different hosts. The RMI protocol handles the serialization and deserialization of objects preserving thereby the type of these objects. RMI stands in contrast to the non platform specific technologies such as CORBA or Web Services. Of course, RMI is just one technology amongst many. As already stated in the previous section, the platform does not impose special requirements on the technology used for the content model as long as its instances can be treated as Plain Old Java Objects (POJOs).

The transformation presented in this section uses the same metamodel and model-to-code transformation for Java as presented in the last section. Thus only a corresponding example for the generated implementation for RMI interfaces and the transformation itself are presented in the following.

5.3.1 Example

The following code listing shows the RMI interface code generated for the content class *ProjectManager* by the transformation presented in the next section after serialization to code. The interface is a specialization of the *Remote* interface and each method is declared to throw a *RemoteException*.

```
public interface ProjectManager extends Remote
{
    public List<Project> getProjects() throws RemoteException;
    public void setProjects( List<Project> _projects ) throws RemoteException;

    public UserProject addUserProject( String _name, String _description,
        ValidationProject _validationProject )  throws RemoteException;
    public ValidationProject addValidationProject(String _name, String _description )
        throws RemoteException;
    public void removeProject( Project _project ) throws RemoteException;
}
```

5.3.2 Transformation Content2RMIInterfaces

The transformation *Content2RMIInterfaces* depicted in Figure 81 maps the content model to a Java model for RMI interfaces that is used to access remotely implemented interface implementations. As already stated for the transformation to JavaBeans, the result of this transformation resembles the code generation facilities that are provided by most UML CASE tools. In contrast to the flexible and completely customizable approach presented here, transformations implemented in CASE tools are usually hard-coded. As this trans-

formation is very similar to the transformation to JavaBeans presented in 5.2.3 only the differences are detailed here. For the technical details about this transformation see B.4.3.

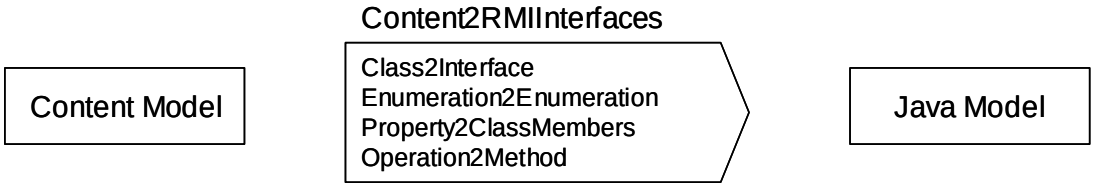


Figure 81. Transformation *Content2JavaInterfaces*

Rule Class2Interface

Each content class from the content model is mapped to a Java RMI interface. The superclass relationship in the source model is mapped to a corresponding superclass relationship (between interfaces) in the target model. Additionally, the super interface for all RMI interfaces, which is generated in the entrypoint rule of the transformation, is included in the list of super classes.

```
rule Class2Interfaces
{
    from c : UWE!Class ( c.oclIsTypeOf( UWE!Class ) )
    to jc : JAVA!JavaClass
    (
        name <- c.name,
        package <- c.package,
        superClasses <- c.generalization->collect( g | g.general )->
            including( thisModule.remoteClass ),
        isAbstract <- false,
        isPublic <- true,
        isInterface <- true
    )
}
```

Rule Property2ClassMembers

Each content property owned by a content class is mapped to corresponding Java getter and setter methods. Properties owned by a class comprise attributes as well as owned association ends. Multi-valued properties are mapped to the parameterized Java collection interfaces *java.util.List<E>* for ordered properties and *java.util.Set<E>* for unordered properties by the unique lazy rules *Class2ParameterizedList* and *Class2ParameterizedSet*, re-

spectively. Note that derived properties are mapped to a getter method by the rule *DerivedProperty2ClassMembers*.

```
rule Property2ClassMembers
{
    from p : UWE!Property ( p.class_.oclIsTypeOf( UWE!Class ) and
        ( p.type.oclIsKindOf( UWE!DataType ) or p.type.oclIsTypeOf( UWE!Class ) ) and
        not p.isDerived )
    to getter : JAVA!Method
    (
        owner <- p.class_,
        name <- 'get' + p.name.stringFirstToUpper(),
        type <- if p.isMultivalued() then
                if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
                thisModule.Class2ParameterizedSet( p.type ) endif
            else p.type endif,
        isPublic <- true,
        isStatic <- false,
        exceptions <- Set { thisModule.remoteException }
    ),
    setter : JAVA!Method
    (
        owner <- p.class_,
        name <- 'set' + p.name.stringFirstToUpper(),
        isPublic <- true,
        isStatic <- false,
        parameters <- Sequence { setterParameter },
        exceptions <- Set { thisModule.remoteException }
    ),
    setterParameter : JAVA!MethodParameter
    (
        name <- '_' + p.name,
        type <- if p.isMultivalued() then
                if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
                thisModule.Class2ParameterizedSet( p.type ) endif
            else p.type endif
    )
}
```

Rule Operation2Method

Each operation in the content model is mapped to a Java method of the corresponding interface. This rule is similar to the rule with the same name in the transformation *Content2JavaBeans*, only that no method body is generated.

```
rule Operation2Method
{
    from o : UWE!Operation ( o.class_.oclIsTypeOf( UWE!Class ) )
    using
    {
        formalParameters : Sequence ( UWE!Parameter ) = o.ownedParameter->select( op |
            op.direction <> #return );
    }
    to m : JAVA!Method
    (
        name <- o.name,
        owner <- o.class_,
        isPublic <- true,
        isStatic <- false,
        parameters <- parameters,
        type <- o.type,
        exceptions <- Set { thisModule.remoteException }
    ),
    parameters : distinct JAVA!MethodParameter foreach ( p in formalParameters )
    (
        name <- '_' + p.name,
        type <- p.type
    )
}
```

5.4 Navigation

The navigation model does not have to be directly transformed to code because in the transformation of the presentation model to Web pages references to elements from the navigation model are resolved: references to nodes, i.e. navigation classes, access primitives and process classes, are resolved to Web pages; and references to properties of nodes are resolved to directly access the content model. Nevertheless, a minimum knowledge about the navigation model is needed in the runtime environment to handle dynamic navigation. For instance, in the example navigation model a navigation link leads from the project index to the abstract navigation class *Project* with the two navigation sub classes *UserProject* and *ValidationProject* as depicted in Figure 82. Thus, when following the link from the project index to a specific project then not the presentation class for the abstract navigation class *Project* should be displayed, but the presentation class for the most specialized navigation subclass which is compatible with the actual content object type. Com-

patible means that the content class type associated to the navigation class is type compatible with the content object. It is important to stress that dynamic navigation is resolved within the runtime environment, i.e. the main controller as described in 5.1.2, and not within the code of a Web page. Therefore, an anchor in a generated Web page always references the target node of a link, such as for example the abstract super navigation class *Project*.

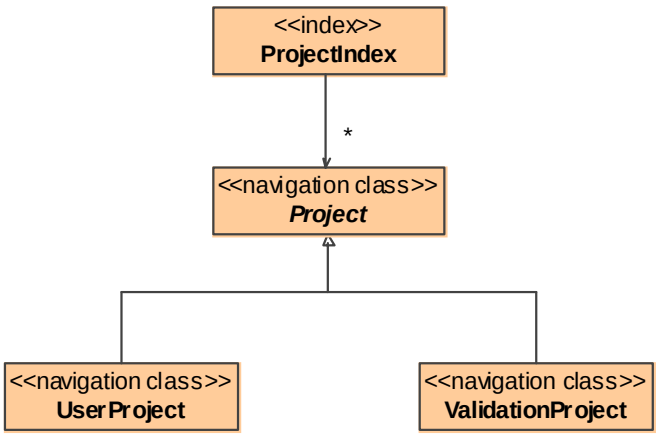


Figure 82. Dynamic navigation structure

For providing the dynamic navigation structure to the runtime environment, the information about the available navigation classes and their inheritance relationships are transformed to configuration data. As discussed in 5.1.2 and in 5.1.3, this configuration data has the form of XML nodes which represent *NavigationClassInfo* objects and their properties in the runtime environment. On instantiation of these configuration beans the runtime environment gets initialized with information about the available navigation classes as explained in 5.1.2.

In the following, first an example for the representation of navigation info classes is given, followed by the transformation from the navigation model to configuration data for the runtime environment.

5.4.1 Example

The following example XML code lines show the generated configuration data for the navigation classes from Figure 82 after serialization to the file *navigation-conf.xml*. Each navigation class is mapped to an XML bean node by the transformation presented in the next section. For each such bean node the Java class *NavigationClassInfo* from the runtime environment is instantiated when the Web application is configured by the Spring bean

factory, see also Figure 75. The property *specific* reflects the inheritance relationship between navigation classes.

```
<bean id="DANUBIA_Navigation_Project" class="uwe.runtime.NavigationClassInfo">
    <property name="name"><value>DANUBIA_Navigation_Project</value></property>
    <property name="specific">
        <list>
            <ref bean="DANUBIA_Navigation_ValidationProject"></ref>
            <ref bean="DANUBIA_Navigation_UserProject"></ref>
        </list>
    </property>
    <property name="contentClass">
        <value>danubia.content.beans.Project</value>
    </property>
</bean>

<bean id="DANUBIA_Navigation_UserProject" class="uwe.runtime.NavigationClassInfo">
    <property name="name"><value>DANUBIA_Navigation_UserProject</value></property>
    <property name="specific">
        <list>
        </list>
    </property>
    <property name="contentClass">
        <value>danubia.content.beans.UserProject</value>
    </property>
</bean>

<bean id="DANUBIA_Navigation_ValidationProject" class="uwe.runtime.NavigationClassInfo">
    <property name="name"><value>DANUBIA_Navigation_ValidationProject</value></property>
    <property name="specific">
        <list>
        </list>
    </property>
    <property name="contentClass">
        <value>danubia.content.beans.ValidationProject</value>
    </property>
</bean>
```

5.4.2 Transformation Navigation2Conf

The transformation *Navigation2Conf* depicted in Figure 83 maps the navigation model to an XML model which is then serialized to an XML document *navigation-conf.xml* as presented in 5.1.3. The transformation comprises one rule which is outlined below and detailed in B.4.3. Each navigation class is mapped by this rule *NavigationClass2Conf* to an

XML bean node. This node is used for the instantiation of the Java class *NavigationClass-Info* when the Web application is configured by the Spring bean factory. The rule is a specialization of the rule *NamedElement2Conf* presented in 5.1.3.2 for mapping model elements to bean nodes.

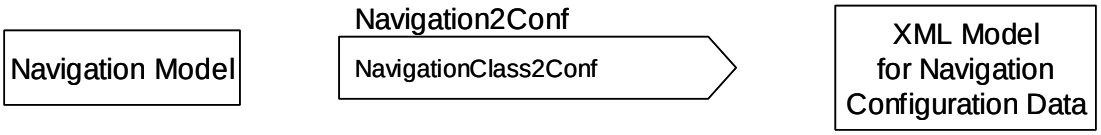


Figure 83. Transformation *Navigation2Conf*

```
rule NavigationClass2Conf extends NamedElement2Conf
{
    from el : UWE!NavigationClass
    to classAttr : XML!Attribute
    (
        name <- 'class',
        value <- 'uwe.runtime.NavigationClassInfo',
        parent <- beanEl
    )
    do
    {
        thisModule.CreateConfProperty( beanEl, 'name', el.qualifiedId() );
        thisModule.CreateConfProperty( beanEl, 'specific',
            UWE!Generalization.allInstances()->select( g | g.general = el )->collect( g | g.specific ) );
        thisModule.CreateConfProperty( beanEl, 'contentClass', el.contentClass.fullJavaName() );
    }
}
```

5.5 Process

As presented in 4.5.2, process flows are modeled with extended UML activities allowing the composition of complex workflows. And, in contrast to other Web approaches, the process flow model is not dissolved at design level into modeling primitives of the navigation model. The drawback of using activities for process modeling reveals when the process flow model has to be transformed to the platform specific level. Because of the complex execution semantics of activities based on token flows as described in [OMG05a] the mapping to an executable implementation is difficult.

The proposed solution is to use a platform specific implementation of the platform independent process metamodel presented in 4.5.2. A transformation maps a process model to XML nodes that represent the corresponding configuration of the process runtime environment presented in the following. The process runtime environment is part of the generic runtime environment presented in 5.1, allowing the execution of process activities. The basic structure and behavior of the process runtime environment corresponds to the abstract definition of syntax and semantics of UML activities. Then an example for the generated process configuration is given and finally the corresponding transformation to the configuration data is presented.

5.5.1 Process Runtime Environment: The Web Process Engine

The process runtime environment, or Web process engine, is a part the generic runtime environment presented in 5.1.2. The runtime environment contains a list of available process activities and further holds a reference to the currently active process activity, if any. Within the method *handleRequest* of the main controller depicted in Figure 75 the control flow is delegated to the Web process engine if either a new process should be started, or the next step of the currently active process should be executed. For more details about the integration of processes in the runtime environment see 5.1.2.

The process runtime environment is represented by the collection of process activities of a Web application. In Figure 84 the implementation classes for the execution of a process activity are outlined. The name of the process class representing a process is comprised as an attribute of the corresponding process activity. This reference is needed in the runtime environment to identify the invocation of a process. A process activity comprises a list of activity nodes and set of activity edges. Activity nodes can hold a token which is either a control token, indicating that a flow of control is currently at a specific node, or an object token which indicates that an object flow is at a specific node. Activity edges represent the possible flow of tokens from one activity node to another. Multiple tokens may be present at different activity nodes at a specific point in time. The method *acceptsToken* of an activity node or an activity edge is used to query if a specific token would currently be accepted which then could be received by the method *receiveToken*. An activity has an input parameter node and optionally an output parameter node which serve to hold input and output object tokens.

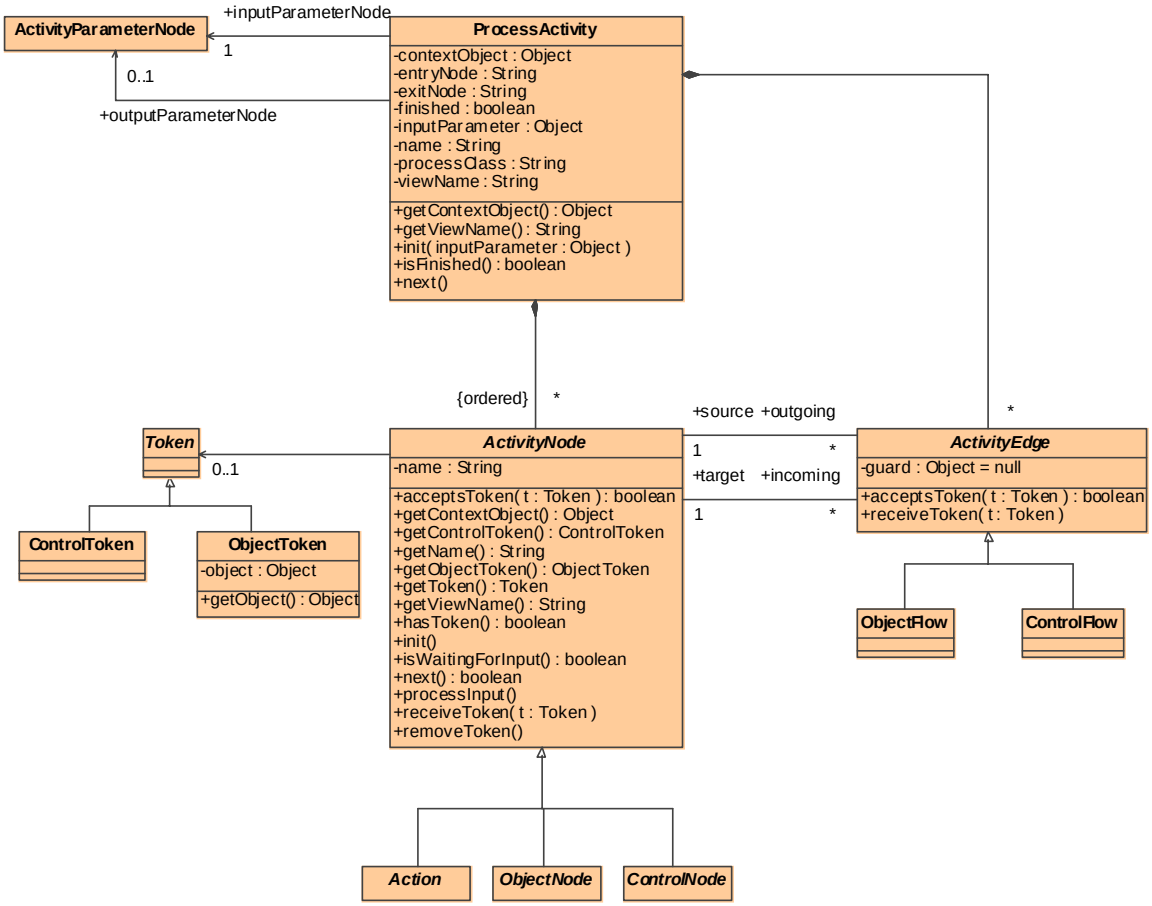


Figure 84. Runtime process activity

In Figure 85 the different kind of control nodes supported by the process engine are depicted. The implementation of these nodes corresponds to the UML specification as defined in [OMG05a]. In comparison to Figure 48, decision and merge nodes are implemented by a common class *DecisionMergeNode*, and fork and join nodes by a common class *ForkJoinNode*, because most modeling tools do not clearly differentiate the corresponding node types. Figure 86 comprises the different kind of object nodes of a process activity supported here which are also compliant with the UML specification, cf. 4.5.2. Pins represent input and output of actions and activity parameter nodes the input and output of process activities, respectively. A central buffer node is used for intermediate buffering of object tokens while a datastore node represents a permanent (i.e. during the execution of the activity) buffer.

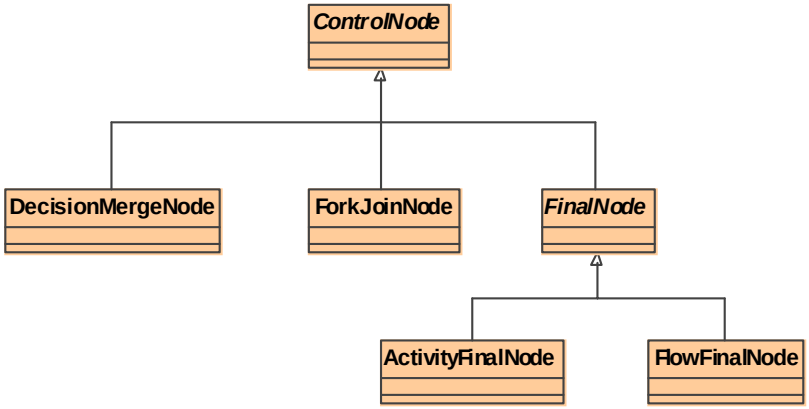


Figure 85. Runtime control nodes

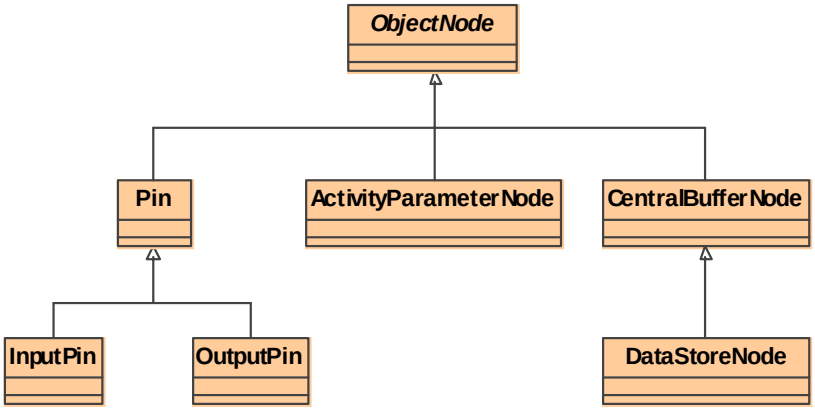


Figure 86. Runtime object nodes

The different kinds of actions supported here are depicted in Figure 87. Input and output pins are associated to actions. An action starts its execution when tokens are available at all input pins. If optionally a control flow is entering the action an additional control token is required. After completion of an action the result data is available at the output pins, and an additional control token is available at the corresponding outgoing control flow, if a control flow is leaving from the action. The call operation action executes fully automatically by invoking a method on the target object. The call behavior action is used to compose process activities and controls the execution of a subordinated process activity (other kinds of subordinated behavior are not supported, hence the difference to Figure 47). Finally, a user action represents an interaction with the user. When it is ready to be executed, i.e. all required input and control tokens are available, then it indicates that it is waiting for input. The corresponding user interaction object for the input is returned by calling the method *getContextObject* which is specified in the super class *ActivityNode*.

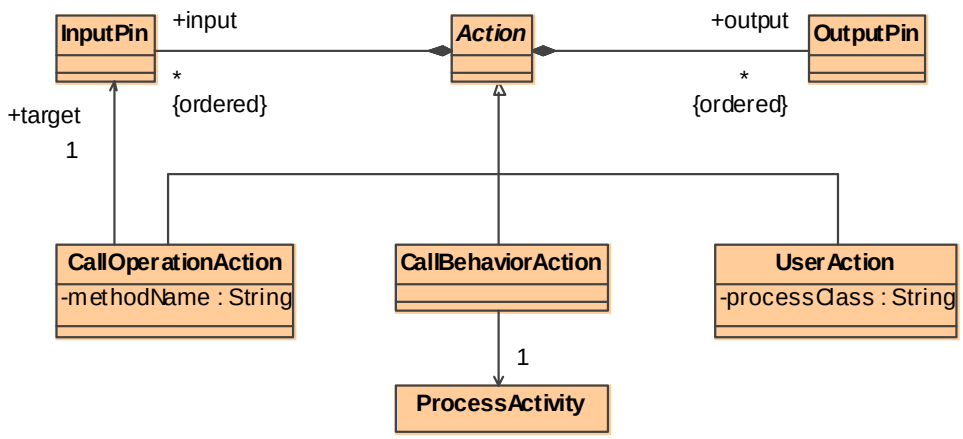


Figure 87. Runtime actions

Before starting the execution of a process activity it has to be initialized by calling the method *init*. This results in initializing all activity nodes and placing an object token in the input parameter node as illustrated by the following simplified Java code lines:

```
public void init( Object inputParameter )
{
    // initialize all activity nodes
    for( ActivityNode n : activityNodes ) n.init();

    // place new object token in input parameter node
    inputParameterNode.receiveToken( new ObjectToken( inputParameter ) );
    this.inputParameter = inputParameter;

    finished = false;
}
```

The complete execution of a process activity comprises the handling of user interactions. Thus, when a process activity contains at least one user interaction then it cannot be executed completely in one step. This is the case if a process activity contains at least one user action either directly, or indirectly by containing a call behavior action that calls another process activity that contains a user interaction. The method *next* of a process activity is called from the runtime environment to execute the process activity until the next user interaction is encountered or the process activity has finished its execution, see 5.1.2. Moreover, either the next user interaction object to be presented to the user is saved in the attribute *contextObj*, or the output parameter object if the activity has finished with a return value. The following code lines give an outline to the implementation of the method *next*:

```
public void next()
{
    // process input requested after last method call
    for( ActivityNode n : activityNodes )
    {
        if( n.isWaitingForInput() )
        {
            n.processInput();
            break;
        }
    }

    // token passing loop
    while( true )
    {
        boolean progress = false;
        for( ActivityNode n : activityNodes )
        {
            progress |= n.next();

            // return in case of waiting for user input
            if( n.isWaitingForInput() )
            {
                contextObject = n.getContextObject();
                viewName = n.getViewName();
                return;
            }

            // finish activity and return in case that the output parameter node has an object token
            else if( n == outputParameterNode && n.hasToken() )
            {
                finished = true;
                contextObject = outputParameterNode.getObjectToken().getObject();
                viewName = exitNode;
            }

            // finish activity and return in case of reached activity final node
            else if( n instanceof ActivityFinalNode && n.hasToken() )
            {
                finished = true;
                contextObject = inputParameter;
                viewName = entryNode;
                return;
            }
        }
```

```
        // throw an exception if no progress has been made
        if( !progress ) throw new ProcessActivityStallException();
    }
}
```

First the method *processInput* of the first activity node that was waiting for input in the last step is called to process the user input that is now available in the user interaction object. Then all activity nodes are notified to execute its behavior by calling the method *next*. If a node then indicates that is waiting for input the method returns with the user interaction object returned by this node. If a token arrives either at an activity output parameter node or at an activity final node the execution of the process activity terminates and the method returns. After a full loop over all activity nodes a progress must have been made. Then the loop is repeated. Each activity node therefore has to indicate on returning from the method *next* if it made a progress. If no progress has been made an exception is thrown to indicate that the progress of the process activity has stalled.

A detailed example for the execution of processes in the runtime environment is given in 6.2.3.

5.5.2 Example

As already stated and in contrast to the content and presentation concerns of a Web application, the process model is not transformed to code in a specific programming language but to configuration data of the runtime environment. As explained in 5.1.3 this configuration data has the form of XML bean nodes which represent the objects of the process runtime environment and their properties as depicted in Figure 84 to Figure 87. On instantiation of these configuration beans the runtime environment gets initialized with the available process activities which correspond to the process model. The following example shows an excerpt from the serialized XML bean definition document *process-conf.xml* for the process activity and the input activity parameter node of the process *RemoveProject* depicted in Figure 57.

```
<bean class="uwe.runtime.process.ProcessActivity"
    id="ProcessActivity_DANUBIA_Process_RemoveProject_RemoveProject">
    <property name="name"><value>RemoveProject</value></property>
    <property name="processClass">
        <value>DANUBIA_Process_RemoveProject</value>
    </property>
    <property name="entryNode">
        <value>DANUBIA_Navigation_ProjectManager</value>
```

```
    </property>
    <property name="activityNodes">
        <list>
            <ref bean="ActivityParameterNode _DANUBIA_Process
                _RemoveProject_RemoveProject_ProjectManager"></ref>
            …
        </list>
    </property>
    <property name="activityEdges">
        <list>
            …
        </list>
    </property>
    <property name="inputParameterNode">
        <ref bean="ActivityParameterNode_DANUBIA_Process
            _RemoveProject_RemoveProject_ProjectManager"></ref>
    </property>
</bean>

<bean class="uwe.runtime.process.ActivityParameterNode" id="ActivityParameterNode_DANUBIA
    _Process_RemoveProject_RemoveProject_ProjectManager">
    <property name="name"><value>ProjectManager</value></property>
    <property name="activity">
        <ref bean="ProcessActivity_DANUBIA_Navigation_RemoveProject_RemoveProject"></ref>
    </property>
    <property name="incoming"><list></list></property>
    <property name="outgoing"><list>…</list></property>
</bean>
…
```

5.5.3 Transformation Process2Conf

The transformation *Process2Conf* depicted in Figure 88 maps the process model to configuration data for the process runtime environment. Therefore the transformation rules defined in 5.1.3.2 are reused. For each class of the process runtime environment depicted in Figure 84 to Figure 87 a transformation rule which specializes the rule *NamedElement2Conf* defined in 5.1.3.2 is responsible for mapping the corresponding model elements from the process model to a bean node in the XML configuration model. Two basic rules for mapping process activities and activity nodes are outlined in the following. The later is an abstract rule that is specialized by sub rules. The rule inheritance hierarchy corresponds to the class inheritance hierarchy of the classes for the process runtime environment depicted in Figure 84 to Figure 87. Finally, the resulting XML model is serialized to the

XML bean definition document *process-conf.xml* as explained in 5.1.3. For the technical details of this transformation see B.4.5.

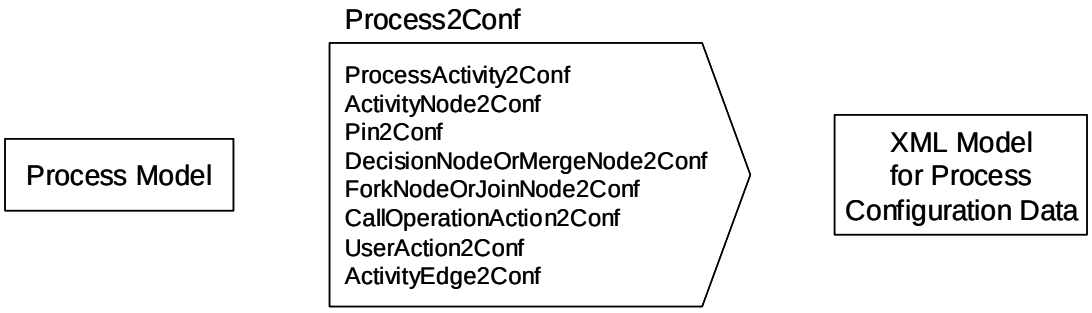


Figure 88. Transformation *Process2Conf*

```
rule ProcessActivity2Conf extends NamedElement2Conf
{
    from el : UWE!ProcessActivity
    using
    {
        inputParameterNode : UWE!ActivityParameterNode =
            el.node->select( n | n.oclIsTypeOf( UWE!ActivityParameterNode ) and
            n.incoming->size() = 0 )->asSequence()->first();
        outputParameterNode : UWE!ActivityParameterNode =
            let ns : Set( UWE!ActivityParameterNode ) =
            el.node->select( n | n.oclIsTypeOf( UWE!ActivityParameterNode ) and
            n.outgoing->size() = 0 ) in if ns->size() = 0 then OclUndefined else
            ns->asSequence()->first() endif;
        …
    }
    to classAttr : XML!Attribute
    (
        name <- ‘class’,
        value <- ‘ProcessActivity’,
        parent <- beanEl
    )
    do
    {
        thisModule.CreateConfProperty( beanEl, 'processClass', el.owner.qualifiedId() );
        thisModule.CreateConfProperty( beanEl, ‘name’, el.name );
        thisModule.CreateConfProperty( beanEl, ‘activityNodes’, el.node->asSequence() );
        thisModule.CreateConfProperty( beanEl, ‘activityEdges’, el.activityEdges );
        thisModule.CreateConfProperty( beanEl, ‘inputParameterNode’, inputParameterNode );
        if( not outputParameterNode.oclIsUndefined() )
        {
```

```
            thisModule.CreateConfProperty( beanEl, 'outputParameterNode',
                outputParameterNode );
        }
    }
    …
}

abstract rule ActivityNode2Conf extends NamedElement2Conf
{
    from el : UWE!ActivityNode
    to classAttr : XML!Attribute
    (
        name <- 'class',
        value <- 'ActivityNode',
        parent <- beanEl
    )
    do
    {
        thisModule.CreateConfProperty( beanEl, 'name', el.name );
        thisModule.CreateConfProperty( beanEl, 'activity', el.activity );
        thisModule.CreateConfProperty( beanEl, 'outgoing', el.outgoing );
        thisModule.CreateConfProperty( beanEl, 'incoming', el.incoming' );
    }
}
```

5.6 Presentation

In this section the use of the Java Server Pages (JSP) technology for the presentation concern is presented. Although JSPs are just one out of many possible technologies for the presentation concern in combination with the Spring framework (see 5.1.1), JSPs are the default presentation technology in Java Web platforms provided by the Tomcat JSP/Servlet container.

Java Server Pages are a technology for dynamic Web pages which are processed in a JSP/Servlet Web container. They allow the embedding of Java code into Web pages and the use of special and possibly customized XML tags within Web pages which are defined in tag libraries. Here the standard tag library named JavaServer Pages Standard Tag Library (JSTL) is used to access the content objects without the need for explicit Java code. JSTL provides tags for common tasks needed for the implementation of Web applications such as iterations or conditional constructs. Additionally, it provides an expression language which is used for accessing the content objects without the need for explicit Java

code, cf. 4.1.3. The following JSP page fragment gives an example about how JSTL can be used. The current content object that should be displayed, i.e. the actual context, is accessible in a variable with the name *self* provided by the Web container as described in 5.1. Expressions in the unified expression language allow to access properties of the content objects. The example page fragment produces an unnumbered list tag with a list item entry containing the name for each project of the project list of a project manager content object. The JSTL tag *forEach* ("c:" is just an XML namespace prefix) represents an iteration over the collection given by the expression *self.projects*. For the tag *out* the Web container evaluates the expression *project.name* and embeds the result into the Web page. For more details about JSP and JSTL see [JSP].

```
<ul>
    <c:forEach var="project" items="${self.projects}">
        <li>
            <c:out value="${project.name}" />
        </li>
    </c:forEach>
</ul>
```

In the following sections first a simple metamodel for JSP pages is specified followed by the results for the running example of this work. Then the transformation from the presentation model to JSP pages and the corresponding serialization transformation to JSP code are presented.

5.6.1 JSP Metamodel

A simple metamodel for Java Server Pages (JSP) is depicted in Figure 89. It is an extension of the XML metamodel presented in 5.1.3.1. The only JSP specific class is *JSPDirective* covering JSP directives of the (serialized) form "<%@ … %>".

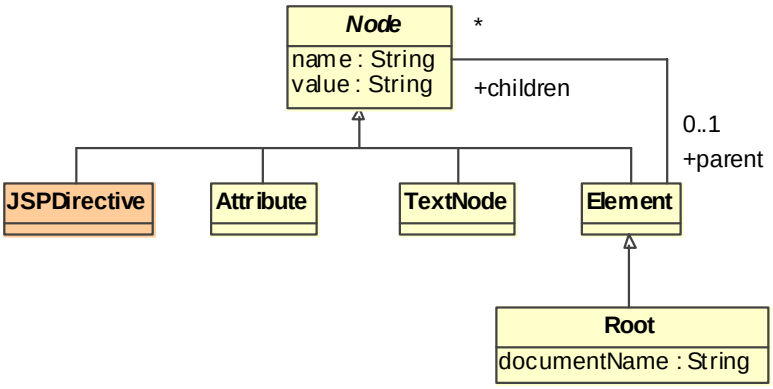


Figure 89. JSP metamodel

5.6.2 Example

The following code sample shows an extract from the generated JSP model after serialization to code for the presentation class *UserProject*. The name of the presentation class was mapped to the content of the *title* and the *h2* tags. Further, for each text element a *c:out* tag was generated to dynamically embed the value of the corresponding expression in the page, such as for example the expression *self.name* which delivers the name of the corresponding user project. For a more detailed example see 6.2.4.

```
<html>
    <head>
        <title>User Project</title>
    </head>
    <body>
        <div>
            <h2>User Project</h2>
            <table>
                <tr>
                    <td>Name:</td>
                    <td><span><span class="" style="">
                        <c:out value="${self.name}"></c:out>
                    </span></span></td>
                </tr>
                <tr>
                    <td>Id:</td>
                    <td><span><span class="" style="">
                        <c:out value="${self.id}"></c:out>
                    </span></span></td>
                </tr>
                <tr>
```

```
                <td>Description:</td>
                <td><span><span class="" style="">
                    <c:out value="${self.description}"></c:out>
                </span></span></td>
            </tr>
        </table>
    </div>
  </body>
</html>
```

5.6.3 Transformation Presentation2JSP

The transformation *Presentation2JSP* depicted in Figure 90 transforms the presentation model to a JSP model representing Java Server Pages. It comprises three main rules which are outlined in the following. The rule *PresentationClass2JSP* maps presentation classes to the JSP model. Sub rules of this rule are responsible for mapping presentation classes for specific associated node types, such as for example presentation classes that are associated to navigation classes. The presentation properties owned by a presentation class are mapped by the rule *PresentationProperty2JSP*. User interface elements are mapped by the rule *UIElement2JSP*. Again, sub rules are responsible for mapping specific user interface element types, such as for example text elements which are transformed by the rule *Text2JSP* also outlined here. The resulting JSP model is then serialized to JSP pages which can directly be executed in the proposed runtime environment without any modification by the developer. For more details about this transformation see B.4.6.

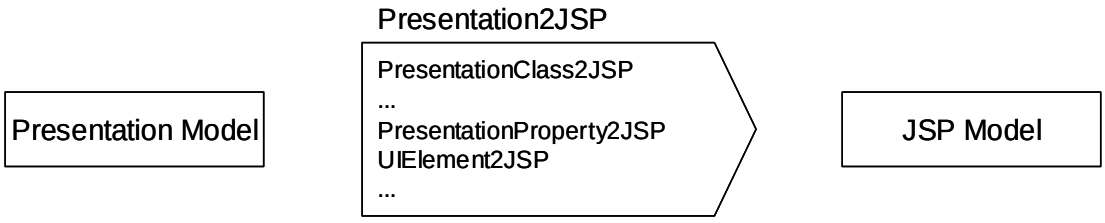


Figure 90. Transformation *Presentation2JSP*

Rule PresentationClass2JSP

Each presentation class is mapped to a *div* element with two attributes *class* and *style* for the specified CSS style for the presentation class. Within the *div* element first a node for the caption of the presentation class is embedded. The corresponding tag name is derived from the containment depth of the presentation class. For a root presentation class the tag

h2 is generated, and for example for a presentation class that is contained within a root presentation class the tag *h3* is generated. The path of containing properties is queried by the helper *containingPropertyPath*. Another helper *formatTypeName* is used to format the name of a type for a better readability on the user interface. The caption for the presentation class *ProjectManager* is for example “Project Manager”. Following the caption node, the transformation targets for all owned attributes as generated by the rule *Presentation-Property2JSP* are embedded in the div tag. If the presentation class is a root presentation class, i.e. if it is not contained in another presentation class, then the parent of the *div* tag is assigned to be the result of the lazy rule *RootPresentationClass2JSP*. The rule *PresentationClass2JSP* is extended for specific node types in order to generate additional tags.

```
rule PresentationClass2JSP
{
    from pc : UWE!PresentationClass
    to pcBody : JSP!Element
    (
        name <- 'div',
        children <- Sequence { cssClassAttr, cssStyleAttr, captionNode, pc.ownedAttribute },
        parent <- if pc.containingClass().oclIsUndefined() then
            thisModule.RootPresentationClass2JSP( pc ) else OclUndefined endif
    ),
    cssClassAttr : JSP!Attribute
    (
        name <- 'class',
        value <- if pc.cssClass.oclIsUndefined() then '' else pc.cssClass endif
    ),
    cssStyleAttr : JSP!Attribute
    (
        name <- 'style',
        value <- if pc.cssStyle.oclIsUndefined() then '' else pc.cssStyle endif
    ),
    captionNode : JSP!Element
    (
        name <- 'h' + ( pc.containingPropertyPath()->size() + 2 ).toString(),
        children <- Sequence { captionTextNode }
    ),
    captionTextNode : JSP!TextNode
    (
        value <- pc.name.formatTypeName()
    )
}
```

Rule PresentationProperty2JSP

Each presentation property is mapped to a *span* node which serves as a container for mapping the type of the presentation property. The type of a presentation property is either a user interface element or a presentation class. Thus, either the generated nodes for a user interface element (see rule *UIElement2JSP*) or for a presentation class (see rule *PresentationClass2JSP*) are embedded in the *span* node.

```
rule PresentationProperty2JSP
{
    from pp : UWE!PresentationProperty
    to spanNode : JSP!Element
    (
        name <- 'span',
        children <- Sequence { pp.type }
    )
}
```

Rule UIElement2JSP

Each user interface element is mapped to a *span* node with two attributes *class* and *style* for the specified CSS style for the user interface element. This rule serves as a base rule for specific user interface types, see for example the rule *Text2JSP*.

```
rule UIElement2JSP
{
    from ui : UWE!UIElement
    to uiBody : JSP!Element
    (
        name <- 'span',
        children <- Sequence { cssClassAttr, cssStyleAttr }
    ),
    cssClassAttr : JSP!Attribute
    (
        name <- 'class',
        value <- if ui.cssClass.oclIsUndefined() then '' else ui.cssClass endif
    ),
    cssStyleAttr : JSP!Attribute
    (
        name <- 'style',
        value <- if ui.cssStyle.oclIsUndefined() then '' else ui.cssStyle endif
    )
}
```

Rule Text2JSP

Each text element is mapped to a JSTL *out* tag (*"c:"* is the XML namespace prefix) for dynamically retrieving the value of a navigation property. For more details about how the expression *elExpression* is calculated see B.4.6.

```
rule Text2JSP extends UIElement2JSP
{
    from ui : UWE!Text
    using
    {
        elExpression : String = …;
    }
    to uiBody : JSP!Element
    (
        children <- Sequence { cssClassAttr, cssStyleAttr, cOutEl }
    ),
    cOutEl : JSP!Element
    (
        name <- 'c:out',
        children <- Sequence { valueAttr }
    ),
    valueAttr : JSP!Attribute
    (
        name <- 'value',
        value <- '${' + elExpression + '}'
    )
}
```

5.6.4 Serialization to Code

The JSP model is transformed to executable code (i.e. text) with the ATL query *JSP2Code* listed which is an extension of the query XML2Code presented in 5.1.3.3.

```
query JSP2Code = JSP!Root.allInstances()->collect( n | n.getChildren()->
    iterate( n; acc : String = '' | acc + n.toCode() ).writeTo( 'jsp/' + n.documentName ) );

helper context JSP!Element def : getAttributes() : Sequence( JSP!Attribute) =
    self.children->select( cn | cn.oclIsKindOf( JSP!Attribute ) );
```

```
helper context JSP!Element def : getChildren() : Sequence( JSP!Node) =
    self.children->select( cn | not cn.oclIsKindOf( JSP!Attribute ) );

helper context JSP!Element def : toCode() : String =
    '<' + self.name + self.getAttributes()->iterate( n; acc : String = '' | acc + ' ' + n.name + '=\"' +
    n.value + '\"' ) + '>\n' + self.getChildren()->iterate( n; acc : String = '' | acc + n.toCode() )
    + '</' + self.name + '>\n';

helper context JSP!TextNode def : toCode() : String =
    self.value;

helper context JSP!JSPDirective def : toCode() : String =
    '<%@ ' + self.name + ' ' + self.value + ' %>\n';
```

6 CASE STUDY

This chapter demonstrates the results of the previous chapters by means of the DANUBIA case study which was introduced in 1.3. Therefore, first the platform independent analysis and design of the case study is presented which comprises the automatic and manual construction of the analysis and design models. Then, the transition to the platform specific implementation, which results in executable code, is described.

6.1 Platform Independent Analysis and Design

In this section the platform independent analysis and design activities for the development of the case study, as described in chapter 4, are presented. At some places in the text of the following sections screenshots are used to demonstrate the effect of design decisions and model transformations, anticipating the final resulting Web pages.

6.1.1 Requirements

The development of the DANUBIA Web application introduced in 1.3 starts with the construction of the requirements model as described in 4.2. The requirements model comprises the analysis content model and the Web use case model. The analysis content model captures structure and data of the application, while the Web use case model captures the functionality of the application. It is suggested to construct the analysis content model first, although the appropriate order may depend on the concrete Web application type. Following, for each analysis content class of the analysis content model the corresponding Web use cases, which represent the functionality of the analysis content class, are developed.

6.1.1.1 Analysis Content

The analysis content model is a class model that captures structure and data of a Web application. The functionality of the application is represented by Web use cases presented in the next section, hence no operations should be present in the analysis content model. In

the following, first a textual description of the case study is given. Phrases written in *italics* serve then to extract the analysis content model.

The main objective of the DANUBIA Web user interface is the *management* of environmental *projects*, in short named projects in the following. For a project in general a set of *documents* gives detailed information about the project, such as the objective of the project, assumptions or results. Further, two different kinds of projects are distinguished, *user projects* and *validation projects*.

A *user project* serves to examine certain questions, e.g. "how will the expected frequency of the occurrence of extreme discharge at a gage P change within the next 100 years?". For further examination of such questions a collection of *scenarios* is managed by a user project. Before the realization of scenarios the participating simulation components have to be validated. For this purpose a *validation project* is used which comprises a collection of *simulations runs*.

A *scenario* is based on a specific assumption in the context of the question of a *user project*, e.g. "the mean temperature will increase by 3°C with a constant temperature gradient within the next 100 years". Please note that assumptions are only represented informally by (textual) scenario descriptions. For each scenario either exactly one *simulation* can be *run* or a set of simulations, a so called *simulation ensemble*. Additionally, a collection of *documents* is managed for a scenario for documentation purposes.

For a *simulation ensemble* a set of statistically equivalent *simulation runs* can be executed. For example a temperature increase of 3°C in 100 years can be realized by different consistent meteorological data sets. Finally, a *simulation run* represents the executable unit of a simulation.

The analysis content model is then constructed from this textual description. Additionally, for the two different kinds of projects an inheritance relationship has been introduced. Further, attributes (without type information) have been added, such as *name* and *description* of a project and a scenario and *author*, *title* and *abstract* of a document. The resulting analysis content model is depicted in Figure 91.

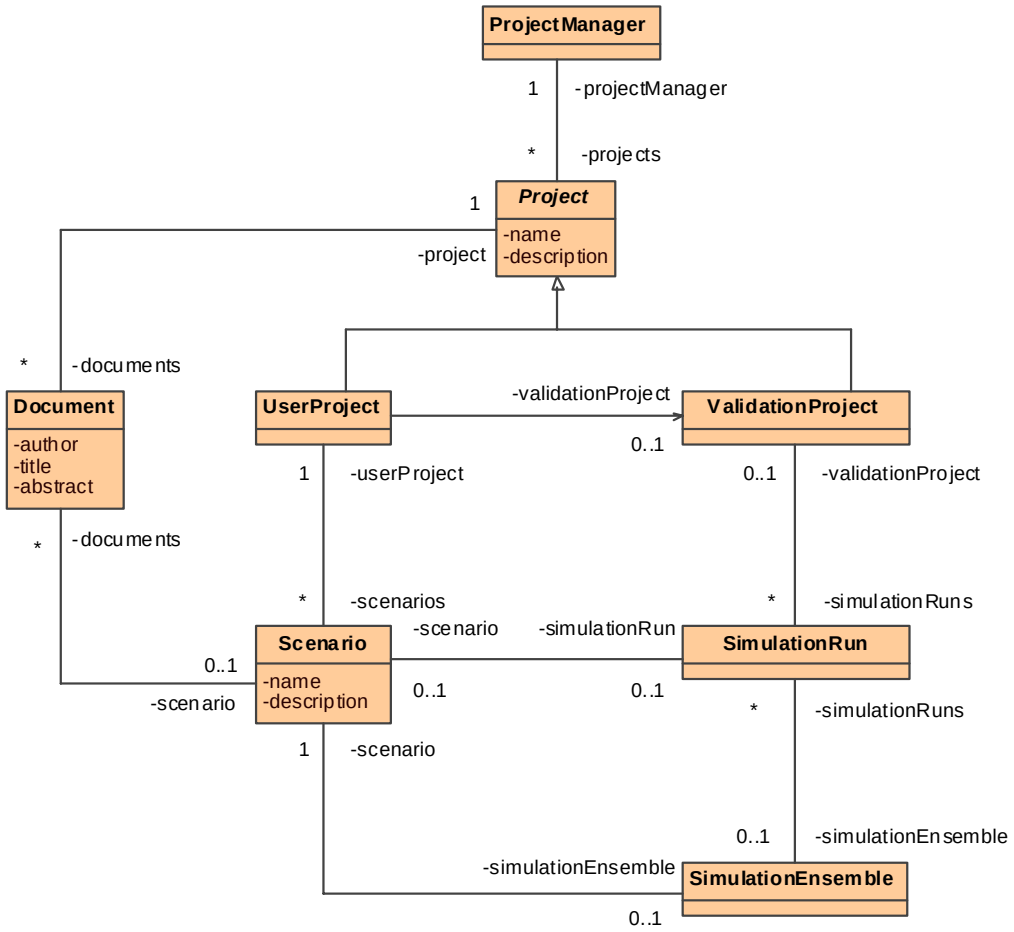


Figure 91. Analysis content model

6.1.1.2 Web Use Cases

Web use cases, i.e. specialized UML use cases, are used for modeling the required functionality of a Web application, see 4.2. For each class from the analysis content model presented in the last section a use case diagram is constructed that comprises all of the corresponding Web use cases which are placed inside the box representing the analysis content class. Navigation and Web process use cases are distinguished. Navigation use cases represent navigation functionality. The target of the navigation functionality is represented by an association between the navigation use case and the corresponding target class of the analysis content model. Two specialized kinds Web processes are distinguished which later allows the automatic derivation of the corresponding trivial workflows: simple processes and edit processes.

The Web use cases for the project manager are depicted in Figure 92. The corresponding analysis content class is represented as a box that contains its use cases. The Web process *Add Project* expresses that the user can add new projects. The resulting project after executing the Web process is notated as the content class *Project* that is associated to the use case. The Web process *Add Project* is not a simple process because it requires a dedicated workflow in which the user first to has to decide which kind of project he wants to add. Then he should enter exactly the information necessary for the selected kind of project. The simple process *Remove Project* expresses that the user can remove a project. The navigation use case *View Projects* means that the user can navigate to the list of projects, represented by the association to the corresponding content class.

For projects in general the user wants to add, remove and view the corresponding documents as depicted in Figure 93. For user projects scenarios can be added, removed and viewed. Additionally, the user can navigate to the corresponding validation project and the user can edit the user project as depicted in Figure 94.

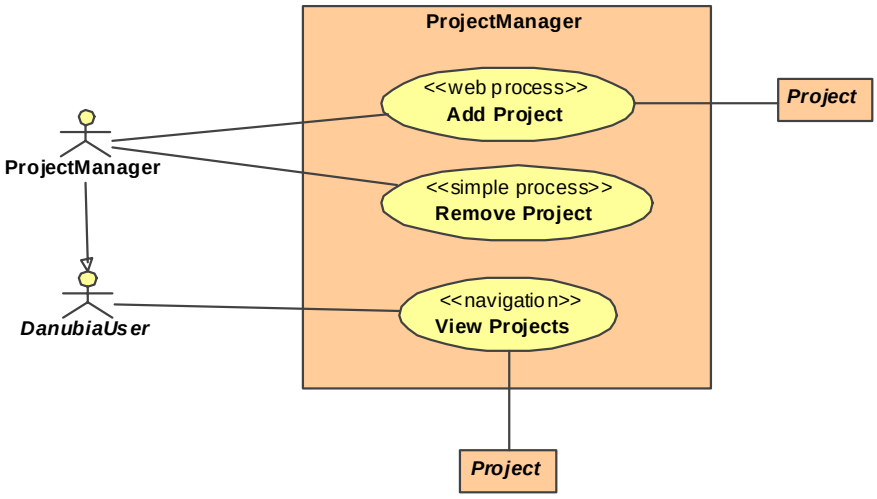


Figure 92. Web use cases for content class *ProjectManager*

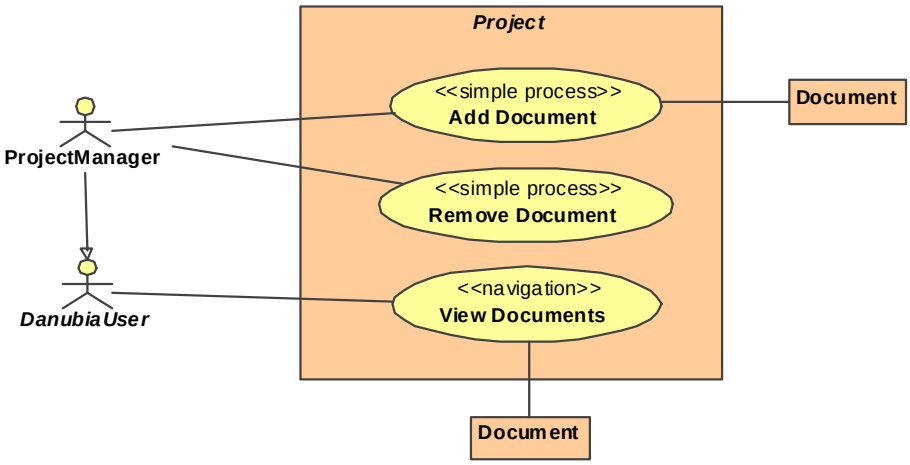


Figure 93. Web use cases for content class *Project*

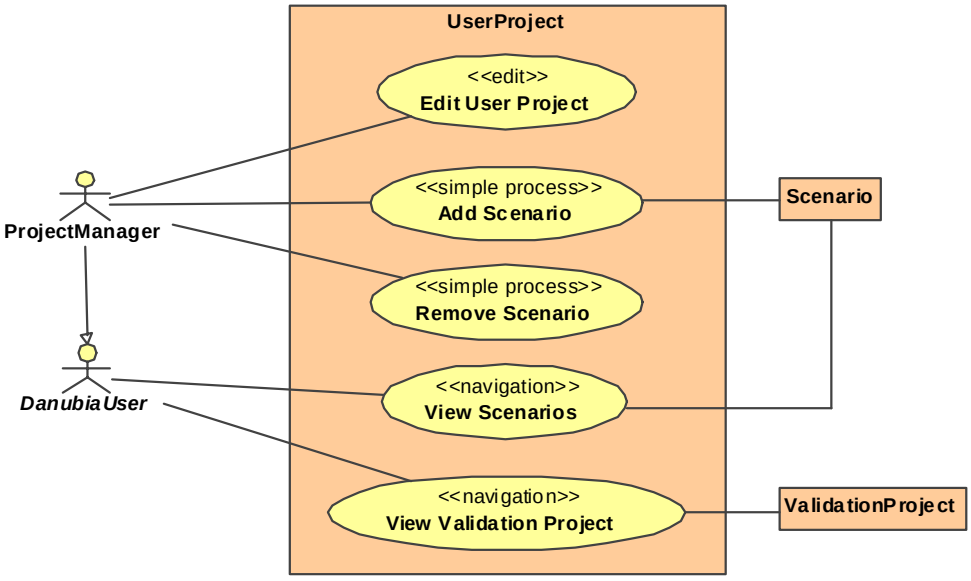


Figure 94. Web use cases for content class *UserProject*

6.1.2 Content

The content model of a Web application captures the structure and the functionality of a Web application, neglecting the navigation, process and presentation aspects as discussed in 4.3. It is, in a first step, derived automatically from the requirements model, i.e. the

analysis content model and the Web use case model presented in the last section. In a second step this model is refined by the user.

6.1.2.1 Results of Transformation Requirements2Content

The requirements model presented in the previous section is automatically transformed to the content model by the transformation *Requirements2Content* presented in 4.3.2. For each simple process in the Web use case model an operation is generated by the rule *SimpleProcess2Operation*. If a target content class for the simple process is specified then a corresponding return type is assigned to this operation. The resulting operation is integrated with the analysis content class by the rule *ContentClass2ContentClassWithOperations.* An overview of the transformation is depicted in Figure 95. The other Web use case types are taken into account as presented in the following sections.

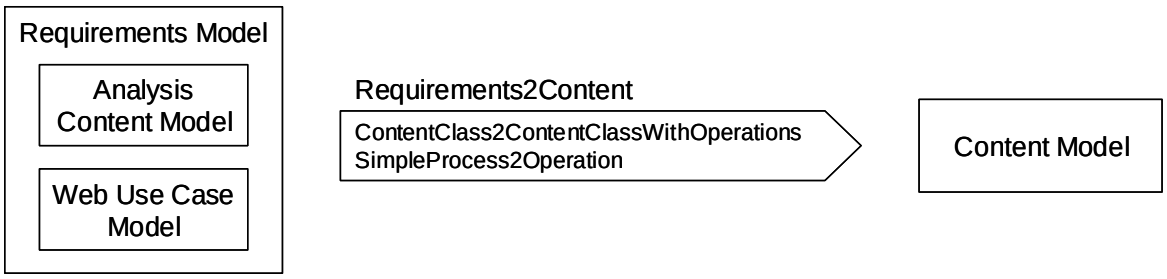


Figure 95. Transformation *Requirements2Content*

The resulting content model is depicted in Figure 96. For instance, for the simple process *Remove Project* depicted in Figure 92 the operation *removeProject* was added to the content class *ProjectManager*.

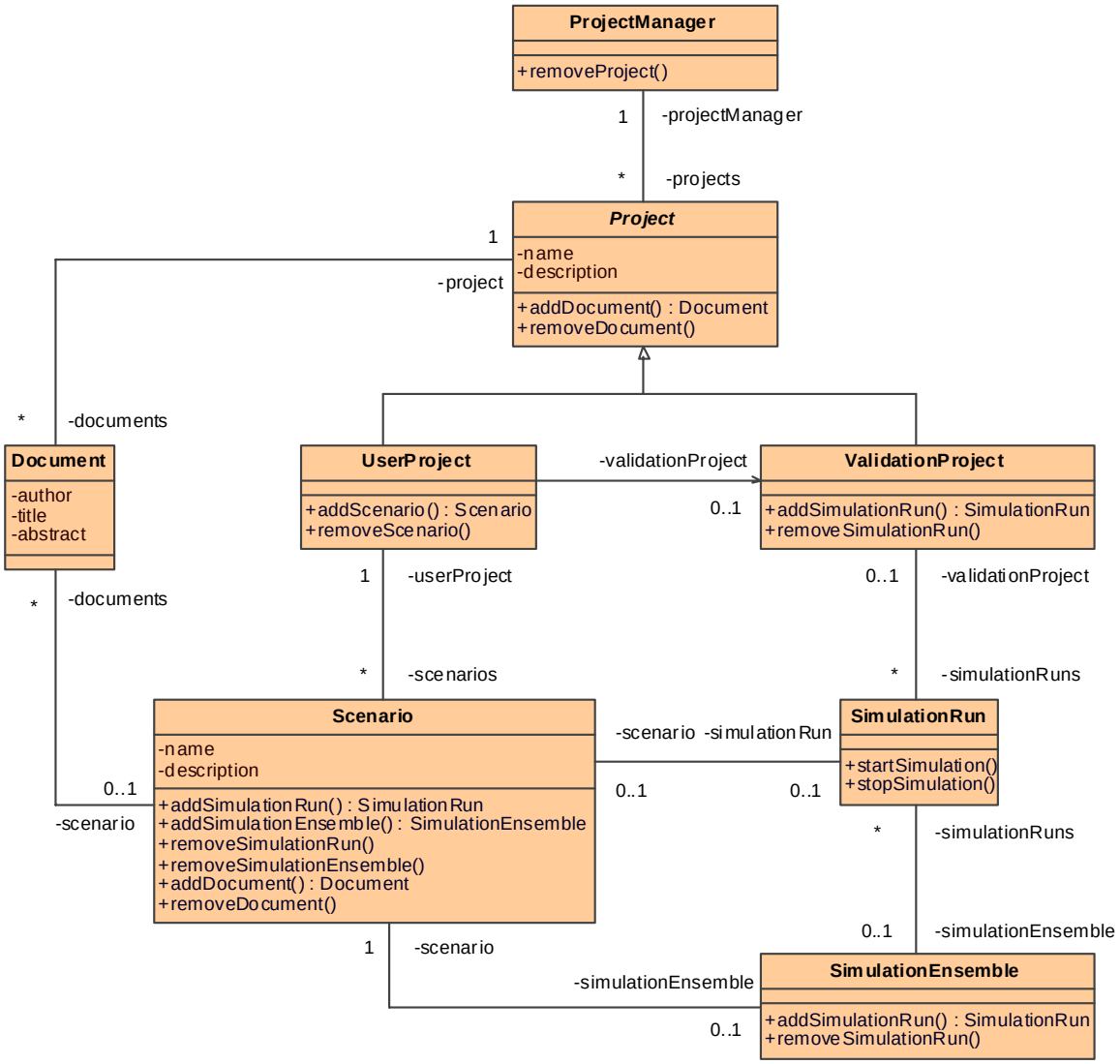


Figure 96. Content model derived by the transformation *Requirements2Content*

6.1.2.2 Manual Refinement

The automatically derived content model has to be manually refined by the developer as described in 4.3.3. The manually refined content model for the case study is depicted in Figure 97. The following modifications of the automatically derived content model have been made:

- Addition of *id* attributes and the attribute *state* for *SimulationRun* to represent the *state* of a simulation run

- Specification of the type for all attributes, including the specification of the enumeration *SimulationState*
- Specification of all multi-valued association ends as ordered properties
- Specification of parameters for all automatically derived operations to represent the data a user has to provide for the invocation of the operation, for example the parameter *project* for the method *removeProject* of the content class *ProjectManager* to indicate that the user has to provide the project that should be removed

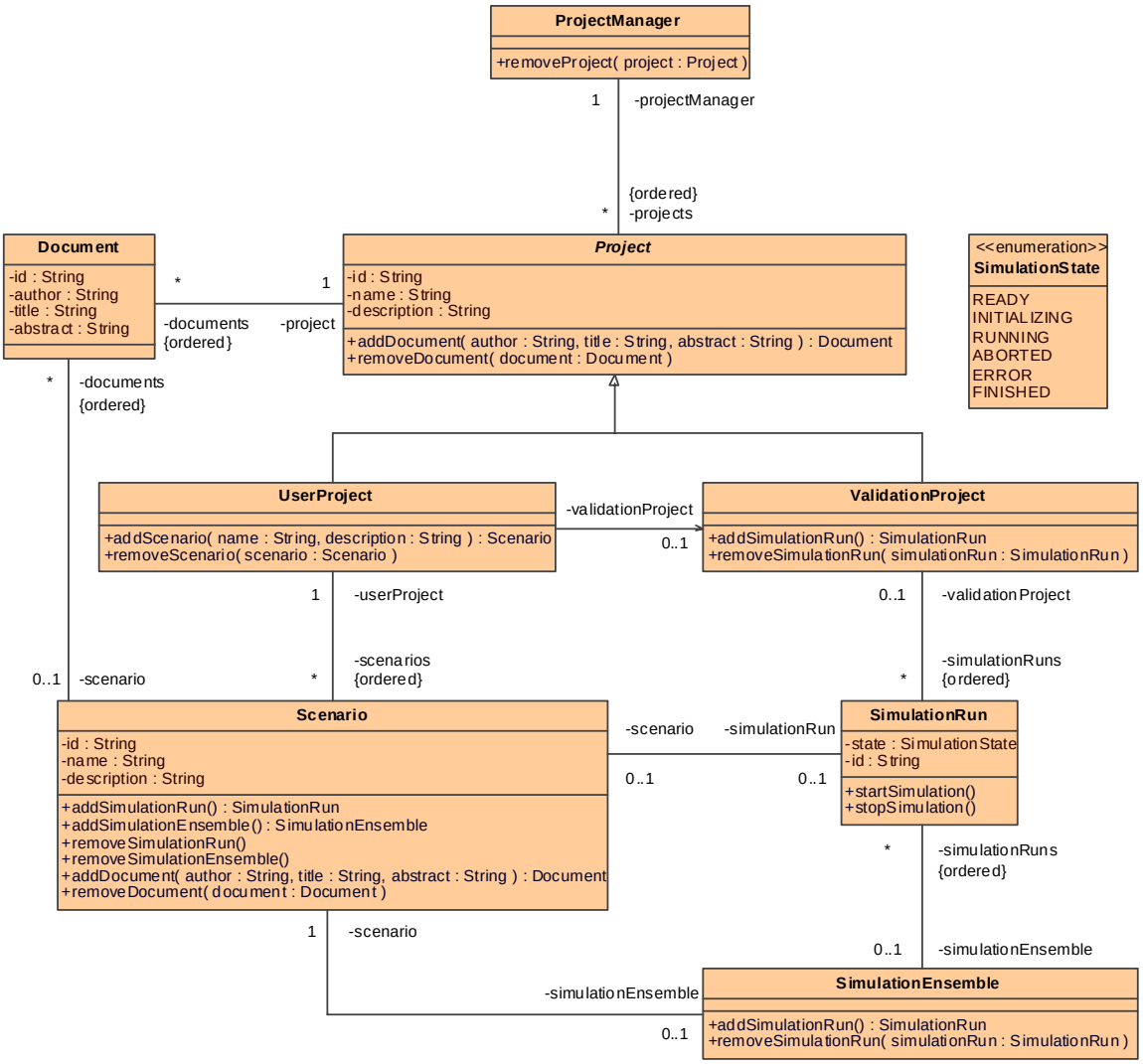


Figure 97. Manually refined content model

6.1.3 Navigation

As presented in detail in 4.3.3, the objective of navigation modeling is to specify the static functionality of a Web application, i.e. a static navigation view of the content. Nodes represent information from the content model and links specify the navigation paths between nodes. In the following three sections the stepwise construction of the navigation model is demonstrated for the case study. The first step comprises the initial derivation of the navigation model from the requirements model and the content model. In the following two steps, first indices and then menus are added to the navigation model. Each step comprises the automatic derivation by a transformation as well as the manual refinement by the developer. Note that with the exception of assigning the home node of the application, manual refinement of the navigation model is not necessary for automatically deriving an executable navigation model.

6.1.3.1 Navigation Space

The navigation space model is the starting point for the construction of the navigation model. It provides a first navigational view of the content model by defining navigation classes and navigation links. It is automatically derived from the requirements model and the content model. This derived navigation model has then to be refined by the developer.

6.1.3.1.1 Results of Transformation RequirementsAndContent2Navigation

The transformation *RequirementsAndContent2Navigation* presented in 4.4.2.1 automatically generates the navigation space model from the requirements model and the content model. For each content class from the content model, which is either the content class or the target of a navigation use case from the requirements model, a navigation class is constructed by the rule *ContentClass2NavigationClass*. Further, for each attribute in the content model a corresponding navigation property is generated by the rule *Property2NavigationProperty*. Properties in the content model which are navigable (in terms of UML properties) ends of associations are mapped to navigation links by the rule *AssociationProperty2NavigationLink*.

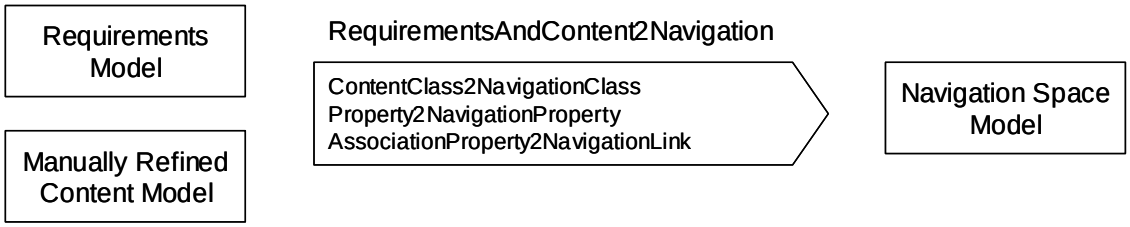


Figure 98. Transformation *RequirementsAndContent2Navigation*

The automatically derived initial navigation space model for the case study is depicted in Figure 99.

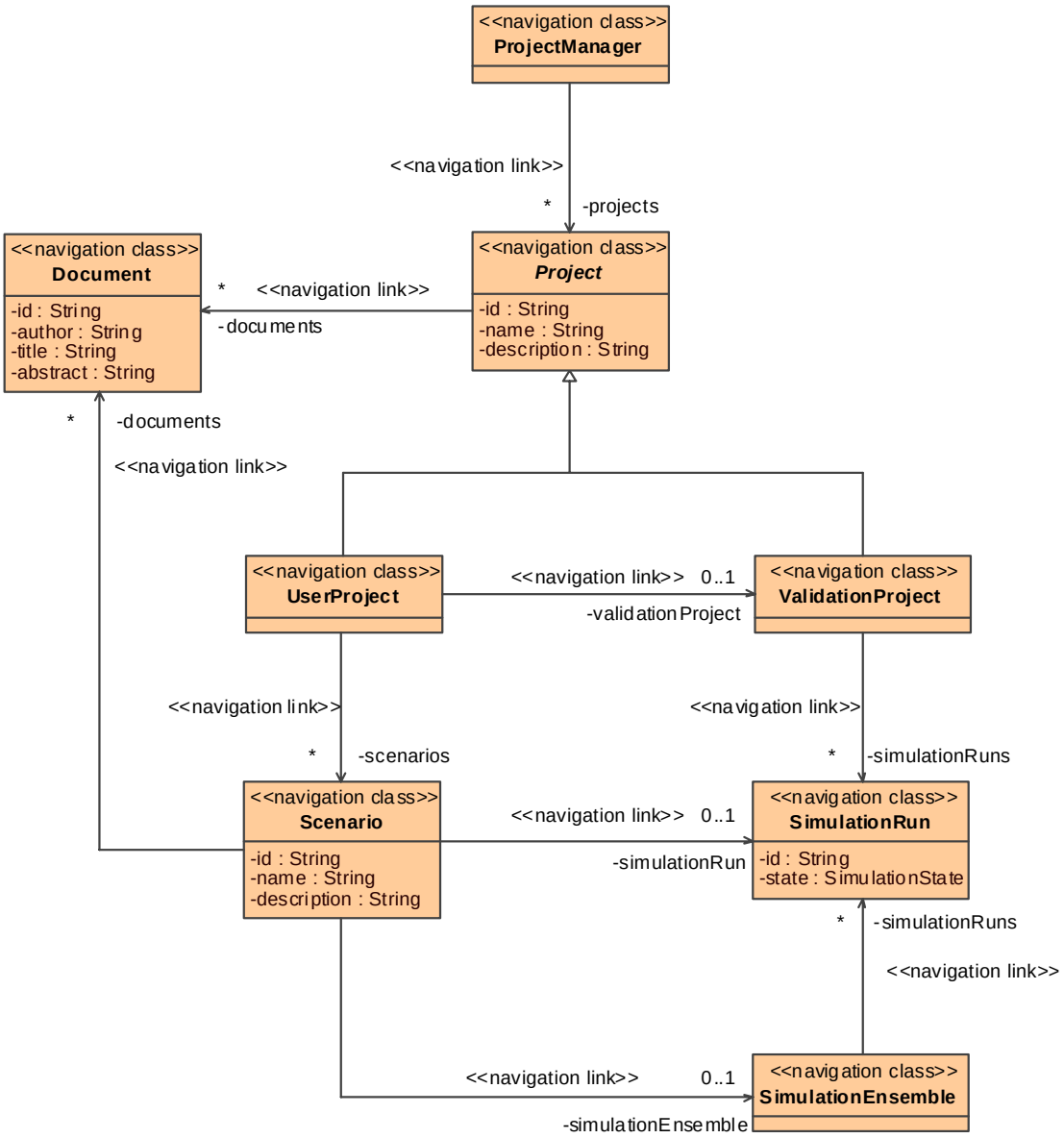


Figure 99. Navigation space model derived by the transformation *RequirementsAndContent2Navigation*

6.1.3.1.2 Manual Refinement

The initial navigation space model for the case study presented in the last section was manually refined, resulting in the navigation model depicted in Figure 100. First, the navigation class *ProjectManager* was designated as entry point of the Web application by setting the *isHome* meta property. Second, back navigation links were added to allow the user to navigate back to each navigation class.

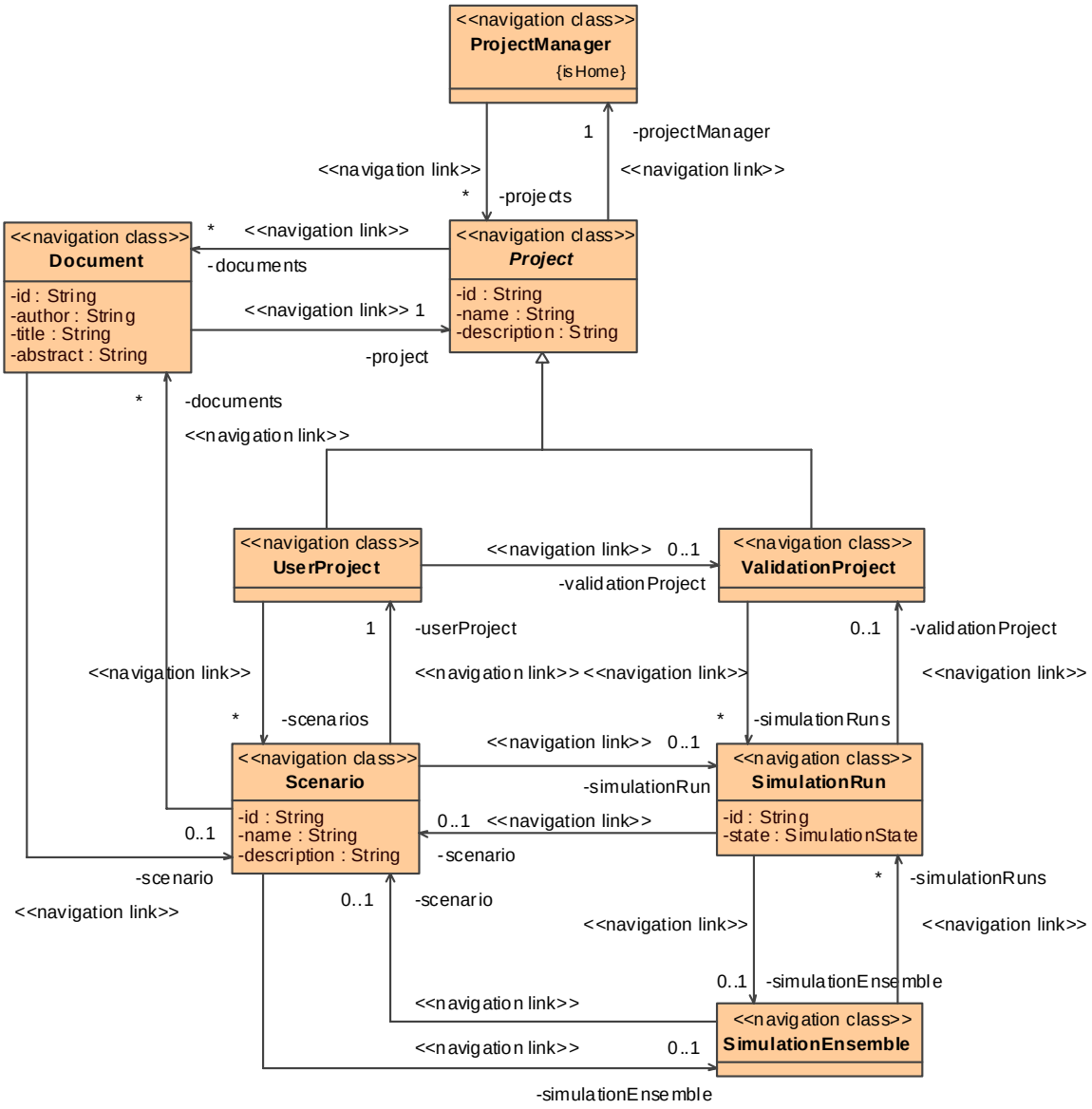


Figure 100. Manually refined navigation space model

6.1.3.2 Addition of Indices

The manually refined navigation space model presented in the last section still comprises multi-valued ends of navigation links between navigation classes. The transformation *AddIndices* presented in 4.4.3.1 inserts index access primitives between the corresponding navigation classes.

6.1.3.2.1 Results of Transformation AddIndices

The transformation *AddIndices* depicted in Figure 101 comprises only one transformation rule *NavigationProperty2Index* that transforms each multi-valued navigation property which is member of a navigation link to an index access primitive.

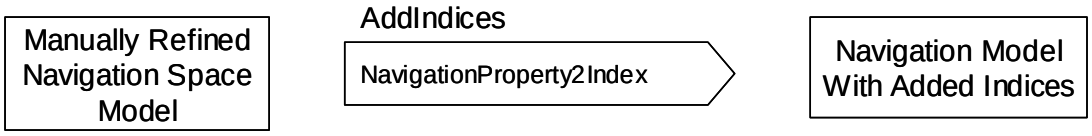


Figure 101. Transformation *AddIndices*

The resulting navigation model with automatically added indices derived from Figure 100 is depicted in Figure 102. Note that in order to avoid name collisions the transformation *AddIndices* automatically prepends the name of the source navigation class to a generated index if otherwise a name collision in the same namespace would occur, for example *ProjectDocumentIndex* and *ScenarioDocumentIndex* instead of two colliding *DocumentIndex* indices. This may result in rather long automatically generated names, but the developer still may change the name to a shorter name in the following manual refinement step. In the case study the automatically generated names are left unchanged in order to stress the systematic evolution of model elements.

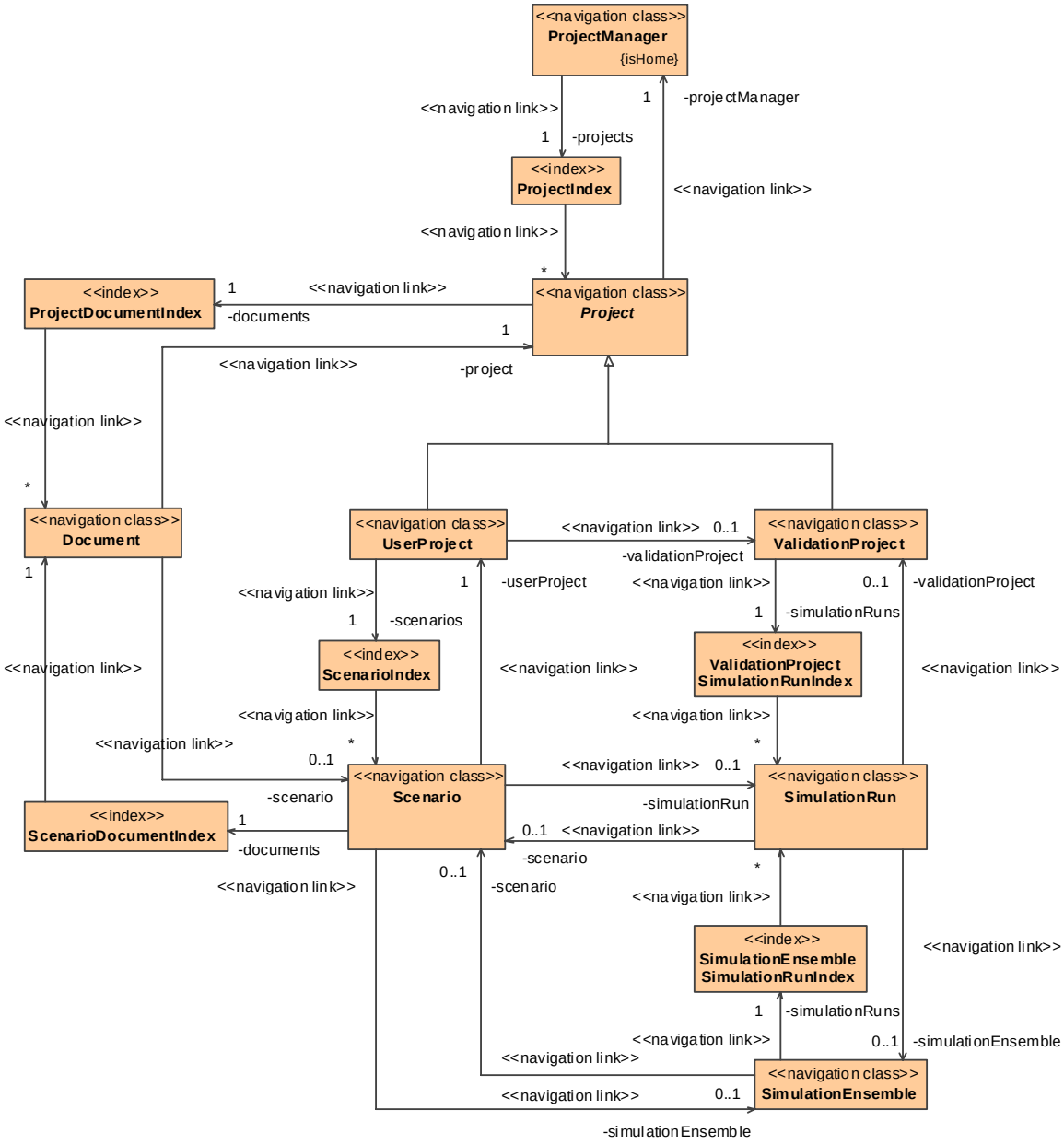


Figure 102. Navigation model with added indices derived by the transformation *AddIndices*

6.1.3.2.2 Manual Refinement

In order to remove the number of navigation links some of the automatically derived navigation links to indices were replaced by associations with composite aggregation kind as depicted in Figure 104. This results in compound nodes for:

- *ProjectIndex*: part of *ProjectManager*

- *ScenarioIndex*: part of *UserProject*
- *ValidationProjectSimulationRunIndex*: part of *ValidationProject*
- *SimulationEnsembleSimulationRunIndex*: part of *SimulationEnsemble*

In Figure 103 the differences between the final results of the automatically derived and the manually refined addition of the index *ProjectIndex* are demonstrated. In the former case an anchor links to the page for the index and in the latter case the page for the index is included in the page for the project manager.

Figure 103. Differences between the automatically derived (above) and the manually refined (below) addition of index *ProjectIndex*

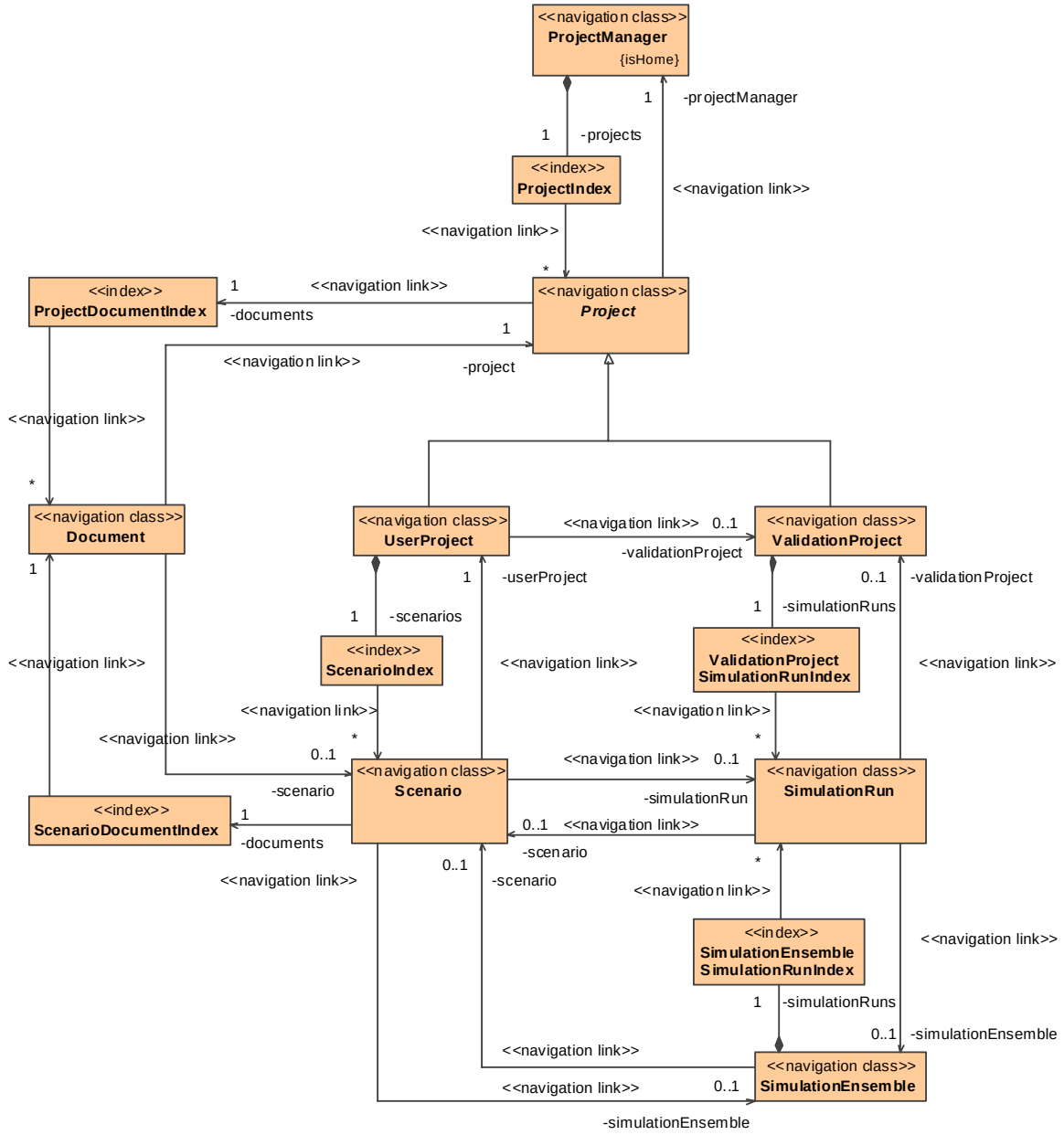


Figure 104. Navigation model after manual refining the automatically added indices

6.1.3.3 Addition of Menus

After the addition of indices as described in the last section, menus are added to the navigation model to organize the outgoing links of navigation classes. The transformation *AddMenus* presented in 4.4.4.1 automatically adds a menu to each navigation class with outgoing links. The resulting navigation model can then be manually refined optionally.

6.1.3.3.1 Results of Transformation AddMenus

The transformation *AddMenus* comprises two transformation rules as depicted in Figure 105. Each navigation class with at least one outgoing link (this condition includes super navigation classes as well), or if for the corresponding content class at least one Web process use case is defined, is transformed to a navigation class with a menu by the rule *NavigationClass2NavigationClassWithMenu*. The second rule *NavigationProperty2MenuProperty* transforms each navigable (in terms of UML properties) navigation property of an outgoing link to a corresponding property of the menu created by the former rule.

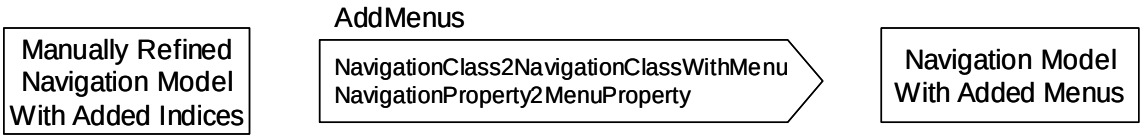


Figure 105. Transformation *AddMenus*

The resulting generated menus for the navigation classes *Project* and *UserProject* are depicted in Figure 106. The inheritance relationship between a project and a user project was mapped to a corresponding inheritance relationship between the generated menus.

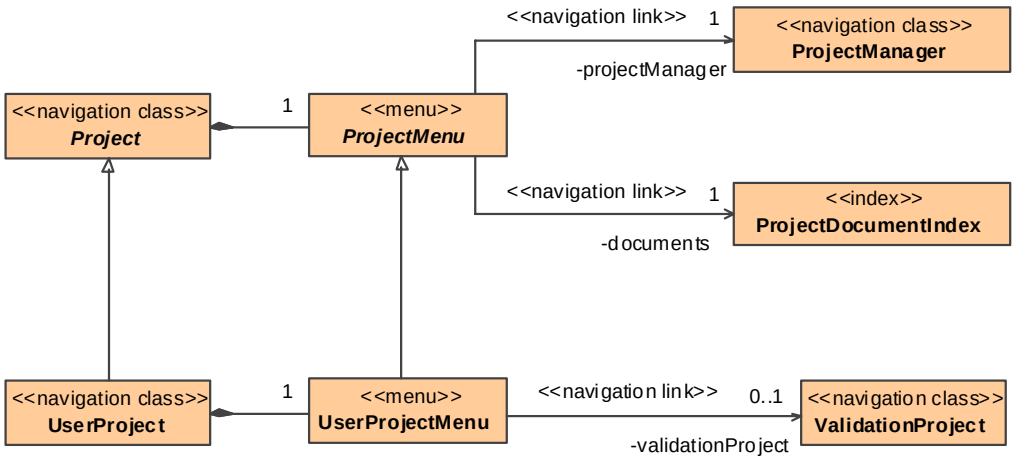


Figure 106. Navigation model with added menus derived by the transformation *AddMenus*

6.1.3.3.2 Manual Refinement

The resulting navigation model was not further refined manually.

6.1.4 Process

Web processes represent the dynamic aspects of a Web application. Processes are integrated in the navigation model by the means of process classes and process links. On the other hand the behavior of Web processes is defined with the process flow model, i.e. process activities. The data required by process activities is captured by the process data model which is developed concurrently with the development of the process flow model. For more details about Web process modeling see 4.5.

6.1.4.1 Process Integration

The process classes and process links needed for the integration of processes in the navigation model are derived from the corresponding Web process use cases in the requirements model. A process link leading to a process class represents the invocation of a process, and a process link leaving a process class represents the presentation of the result of the process.

6.1.4.1.1 Results of Transformation ProcessIntegration

As described in 4.5.1.2, the rule *Menu2IntegratedMenu* of the transformation *ProcessIntegration* creates a designated process class for each Web process use case in the requirements model, see Figure 107. Additionally, an entry process link from the menu for the content class of the Web process use case leading to this process class is generated. If a target is defined for the Web process use case then also an exit process link from the process class to the navigation class created for the target content class is generated.

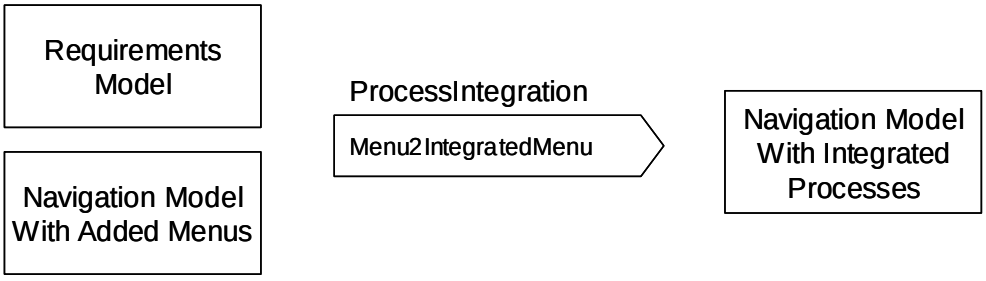


Figure 107. Transformation *ProcessIntegration*

The following figures depict the process classes and links automatically generated for the Web process use cases for the content classes *ProjectManager*, *Project* and *UserProject* in the requirements model.

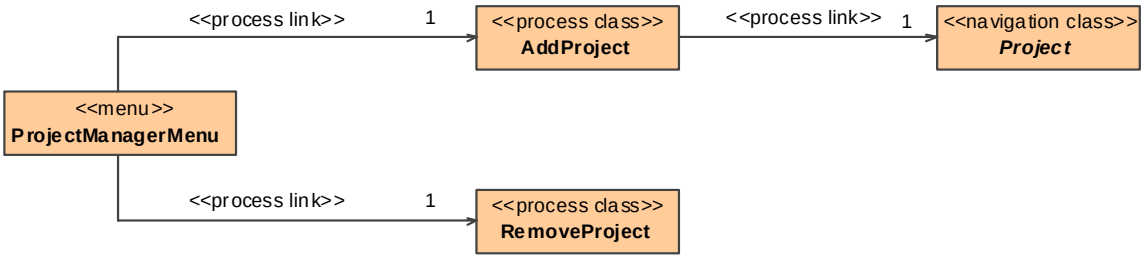


Figure 108. Automatically derived process classes and links for content class *ProjectManager*

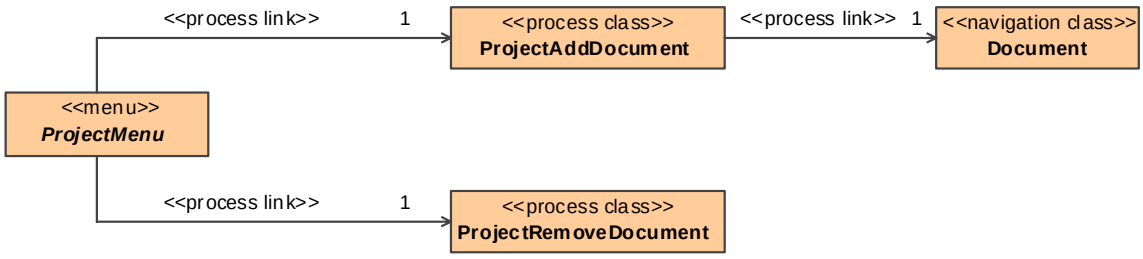


Figure 109. Automatically derived process classes and links for content class *Project*

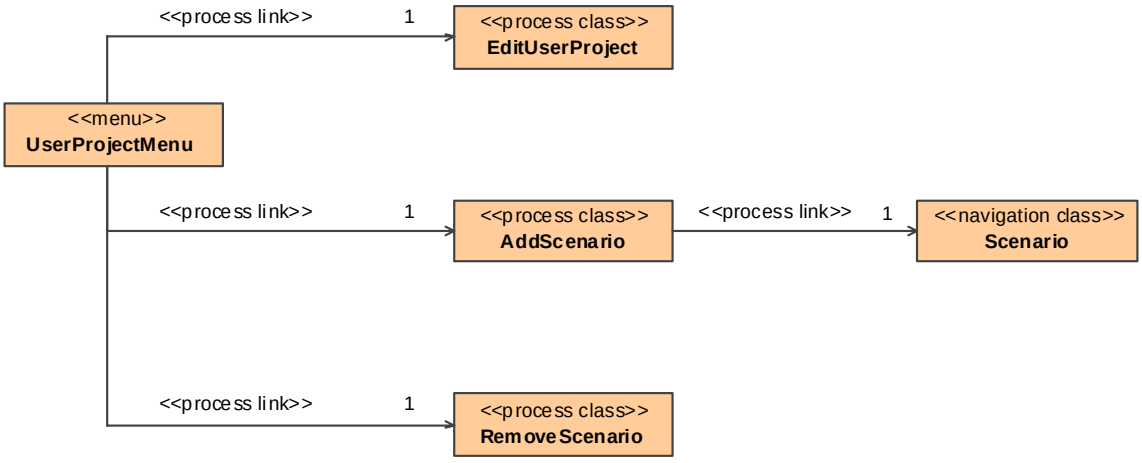


Figure 110. Automatically derived process classes and links for content class *UserProject*

6.1.4.1.2 Manual Refinement

A manual refinement of the automatically generated entry process links was necessary to ensure that processes can only be invoked when certain conditions are fulfilled. As dis-

cussed in 4.5.1.3, guard conditions of links have to be added manually using the expression language presented in 4.1.3. The following guards were defined for the case study, see the referenced figures for the corresponding expressions:

- The process *RemoveProject* can only be invoked when the list of projects of a project manager is not empty, see Figure 111
- The process *ProjectRemoveDocument* can only be invoked when the list of documents of a project is not empty, see Figure 112
- The process *RemoveScenario* can only be invoked when the list of scenario of a user project is not empty, see Figure 113

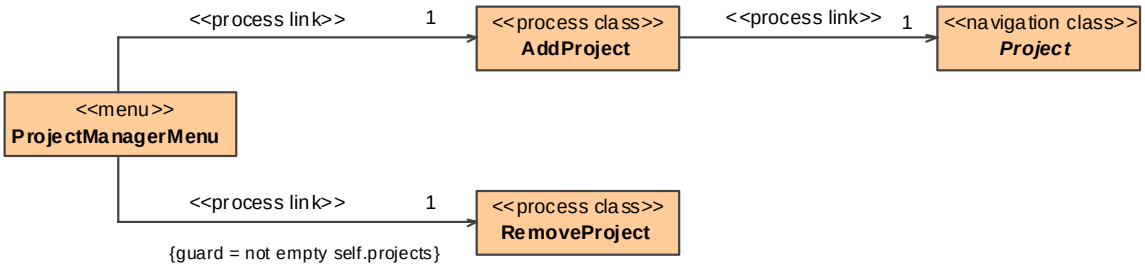


Figure 111. Manually refined process classes and links for content class *ProjectManager*

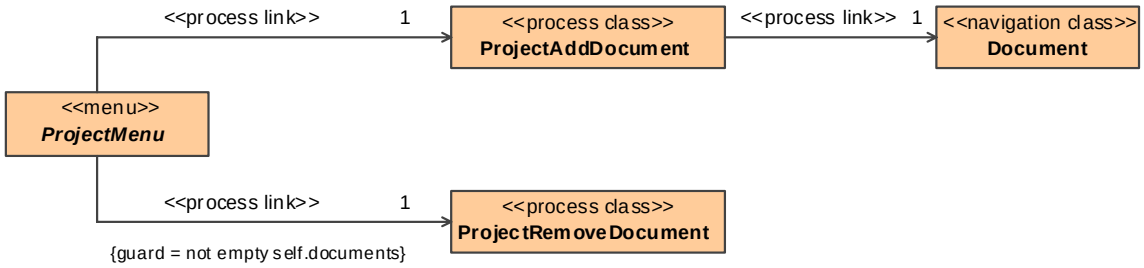


Figure 112. Manually refined process classes and links for content class *Project*

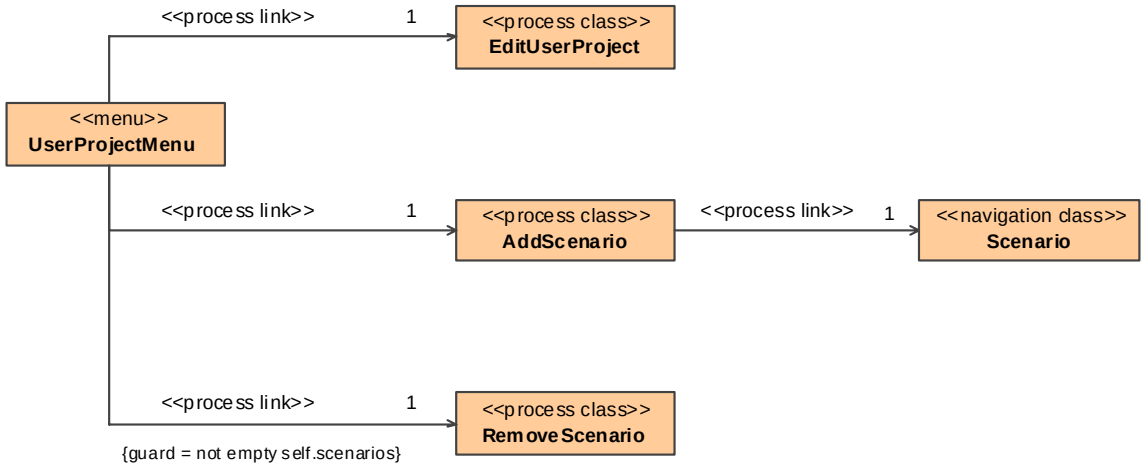


Figure 113. Manually refined process classes and links for content class *UserProject*

6.1.4.2 Process Data and Flow

For each Web process a process activity has to be specified which defines the control and data flow of the process. Further, for each user action of a process activity a process data class has to be defined to capture the data the user has to enter for continuing the execution of the user action. The process flow and the process data models are usually developed concurrently by adding a corresponding process class to the process data model when adding a user action to the process flow model.

6.1.4.2.1 Results of Transformation CreateProcessDataAndFlow

As presented in 4.5.2.2, the transformation *CreateProcessDataAndFlow* depicted in Figure 114 automatically generates the process data and the process flow for all Web process use cases from the requirements model. The data of a process is captured by process data classes and the flow of a process is represented by a process activity which is owned by the designated process class in the navigation model with integrated processes presented in the last section. The transformation comprises three transformation rules which are illustrated in the following.

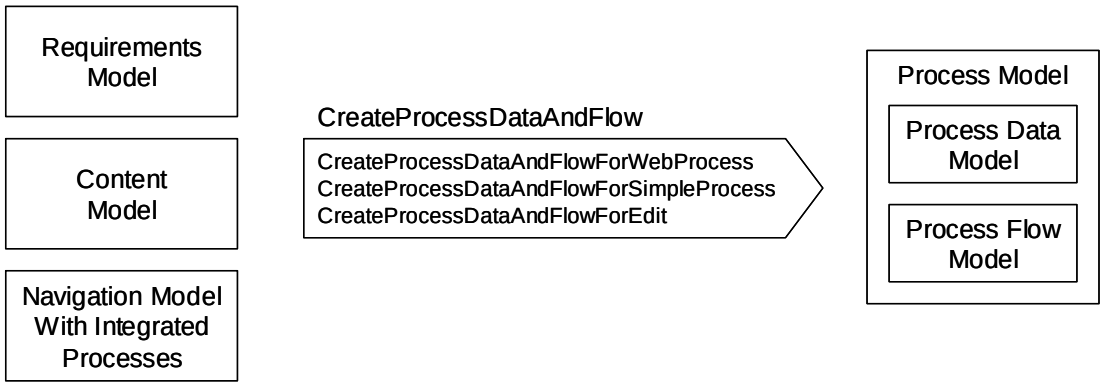


Figure 114. Transformation *CreateProcessDataAndFlow*

The rule *CreateProcessAndDataFlowForWebProcess* generates the process data and flow for complex processes, i.e. neither edit processes nor simple processes. Only the parameters and the activity parameter nodes of the corresponding process activity can be generated, as in the case of the Web process *AddProject*, see Figure 115. The input activity parameter node *ProjectManager* receives an object token of type *ProjectManager*, which corresponds to the content class associated to the source of the incoming process link leading to the process. On the other hand, the output activity parameter node *Project* has to receive an object of type *Project*, corresponding to the content class associated to the target of the outgoing process link leaving the process. Thus, the resulting process flow is incomplete and it has to be refined by the developer as presented in the next section.

For simple processes the complete process flow and data is generated by the rule *CreateProcessAndDataFlowForSimpleProcess*. The generated model elements for the simple processes *RemoveProject*, *AddScenario* and *StartSimulation* are depicted in Figure 116 to Figure 118. The process *AddScenario* serves as an example for the generated model elements in the following. It starts with the input activity parameter node that receives a user project object token for the user project from which the process was invoked, see Figure 117. This token is duplicated by a fork node. One of these duplicated tokens provides the target input pin for the invocation of a call operation action (see below). The other token triggers the user action *AddScenarioInput* for querying input from the user as represented by the associated process data class *AddScenarioInput*. When the user has finished entering the two data fields corresponding to the two attributes *name* and *description* of the process data class the values of these fields are placed at the two corresponding output pins of the user action. These two output pins are connected with two corresponding input pins of the call operation action *addScenario*. The input pins correspond to the parameters of the operation *addScenario* in the content model, cf. Figure 97. After the invocation of the operation on the object provided by the target input pin, the result of the operation call is avail-

able at the scenario output pin, and it is transferred to the output activity parameter node. With the availability of an object token at the output parameter node the process terminates and the resulting scenario object is shown to the user.

For edit processes the complete process flow and data is generated by the rule *CreateProcessAndDataFlowForEdit* as well. An example for the edit process *EditUserProject* is depicted in Figure 119. The process flow starts with the input activity parameter node that receives a user project object token for the user project which should be edited. This object token provides the data for the input pin of the user action *EditUserProjectInput* to determine which object should be edited. As represented by the corresponding process data class *EditUserProjectInput*, the user can modify the attributes of the user project corresponding to the attributes of the process data class *name*, *description* and *id*. After the completion of the input a control flow reaches the activity final node and the process terminates.

Figure 115. Automatically derived incomplete process flow for web process *AddProject*

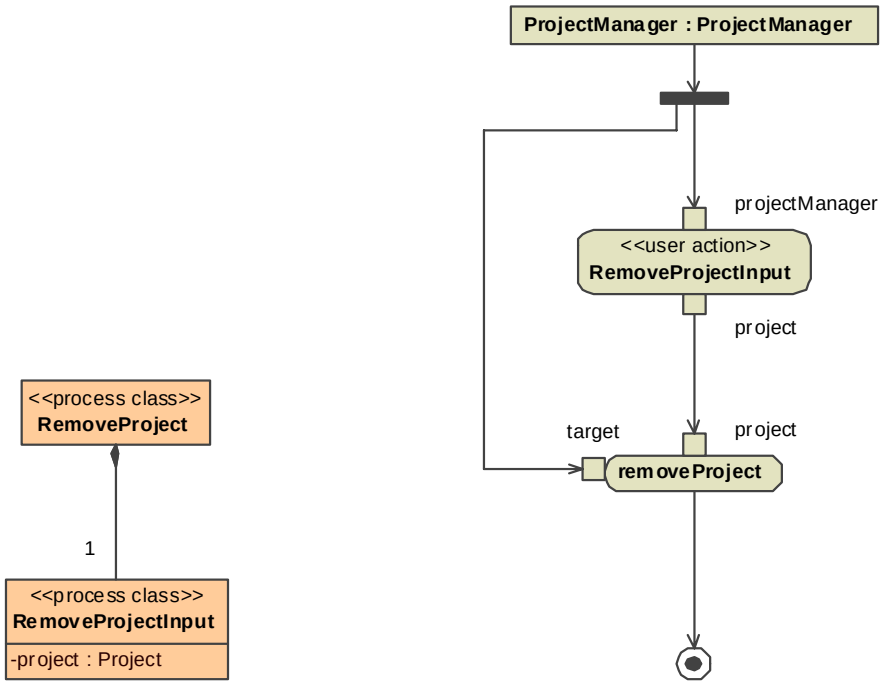


Figure 116. Automatically derived process data and flow for simple process *RemoveProject*

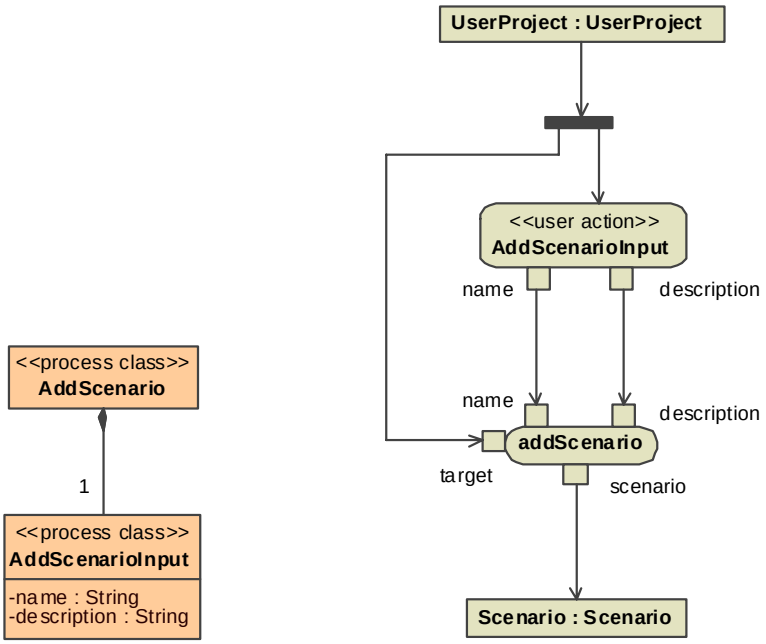


Figure 117. Automatically derived process data and flow for simple process *AddScenario*

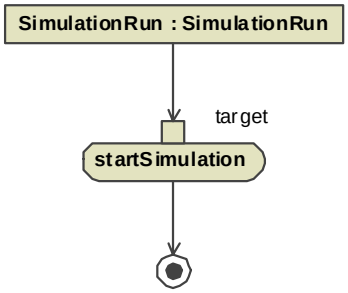


Figure 118. Automatically derived process flow for simple process *StartSimulation*

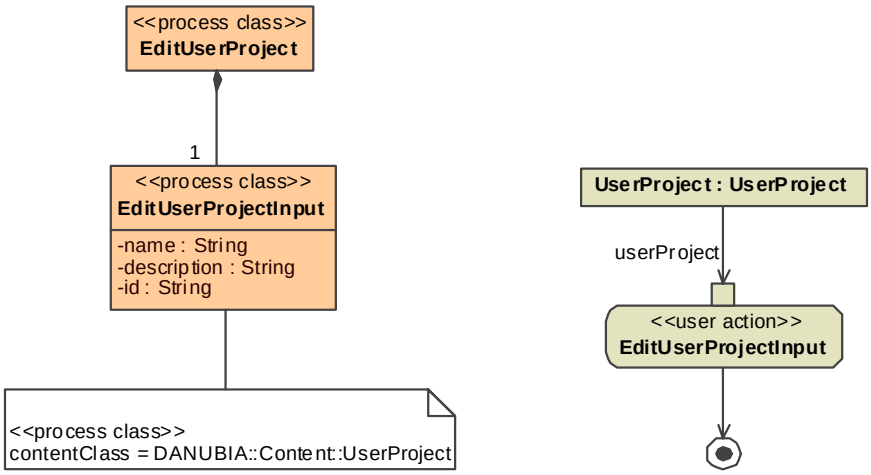


Figure 119. Automatically derived process data and flow for edit process *EditUserProject*

6.1.4.2.2 Manual Refinement

For complex processes, i.e. neither simple processes nor edit processes, the process data and flow has to be completely defined by the developer, with exception of the automatically generated parameters and activity parameter nodes. In Figure 120 the manually defined process flow for the process *AddProject* is depicted. It comprises three user actions and two call operation actions. The first user action *ProjectKindInput* is used to query which kind of project the user wants to add to the project list. Depending on the output of the user action, which is represented by an enumeration type (see below), either the user action *AddValidationProjectInput* or *AddUserProjectInput* is executed to query the parameters for the subsequent call operation action *addValidationProject* or *addUserProject*, respectively. Note that these two call operation actions require different parameters, which have to be provided by the corresponding user actions. Further, the user action *AddUserProjectInput* requires an input pin for the selection of a validation project from a collection of validation projects (see below). After the termination of either call operation action the corresponding project object is passed through a merge node to the output activity parameter node. Taking advantage of the dynamic navigation feature of this approach, either the page for a validation project or for a user project is then shown to the user.

The process data required for the process flow of the process *AddProject* is depicted in Figure 121. For each user action a process class was defined. The process class *ProjectKindInput* captures the selection of a project kind. Therefore a special enumeration type *ProjectKind* was defined. The process class *AddValidationProjectInput* corresponds to the parameters of the operation *addValidationProject* and therefore two attributes of type *String* are required. For the operation *addUserProject* an additional attribute *validation-*

Project is required for the process class *AddUserProjectInput*. The selection of a validation project is optional, hence the multiplicity of the attribute is 0..1. Additionally, a *rangeExpression* has to be defined for attributes which are neither of primitive type nor enumerations, to express in terms of an expression in the expression language the collection from which the value of the attribute should be chosen. In this case the collection is given be the property *validationProjects* of the content class given be the specified *contentClass* for the process class (see below).

Additionally, the definition of the general Web process *AddProject* requires an extension of the content model as depicted in Figure 122. On the one hand the operations *addValidationProject* and *addUserProject,* that are invoked by the introduced call operation actions, have to be added to the content class *ProjectManager*. On the other hand a derived attribute *validationProjects* has to be introduced which is used for the selection of a validation project. Note that the used expression language is not expressive enough to express the value of this attribute directly. When generating code for the content model a getter operation *getValidationProjects* is generated that has to be completed by the developer to return the set of available validation projects.

For simple processes which require the input of a value other than a primitive type or an enumeration, the automatically derived process data has to be refined by the developer in order to define the corresponding *rangeExpression* property as already explained above. In the case study this is the case for all simple processes for removing an object, such as for example the process *RemoveProject* as depicted in Figure 123. Additionally, for all those processes it was chosen to add a further user action to confirm the remove action. The input is represented by a particular process class which uses the special enumeration type *YesNoEnum*. Depending on the output of this confirm user action either the corresponding call operation action is triggered or the process terminates because a token reaches the activity final node directly.

Edit processes do not require manual refinement, but for the case study it was chosen not to let the user edit all attributes of the corresponding content class. Therefore, the automatically generated attribute *id* was removed from the automatically generated process classes, for instance for the edit process *EditUserProject* as depicted in Figure 125.

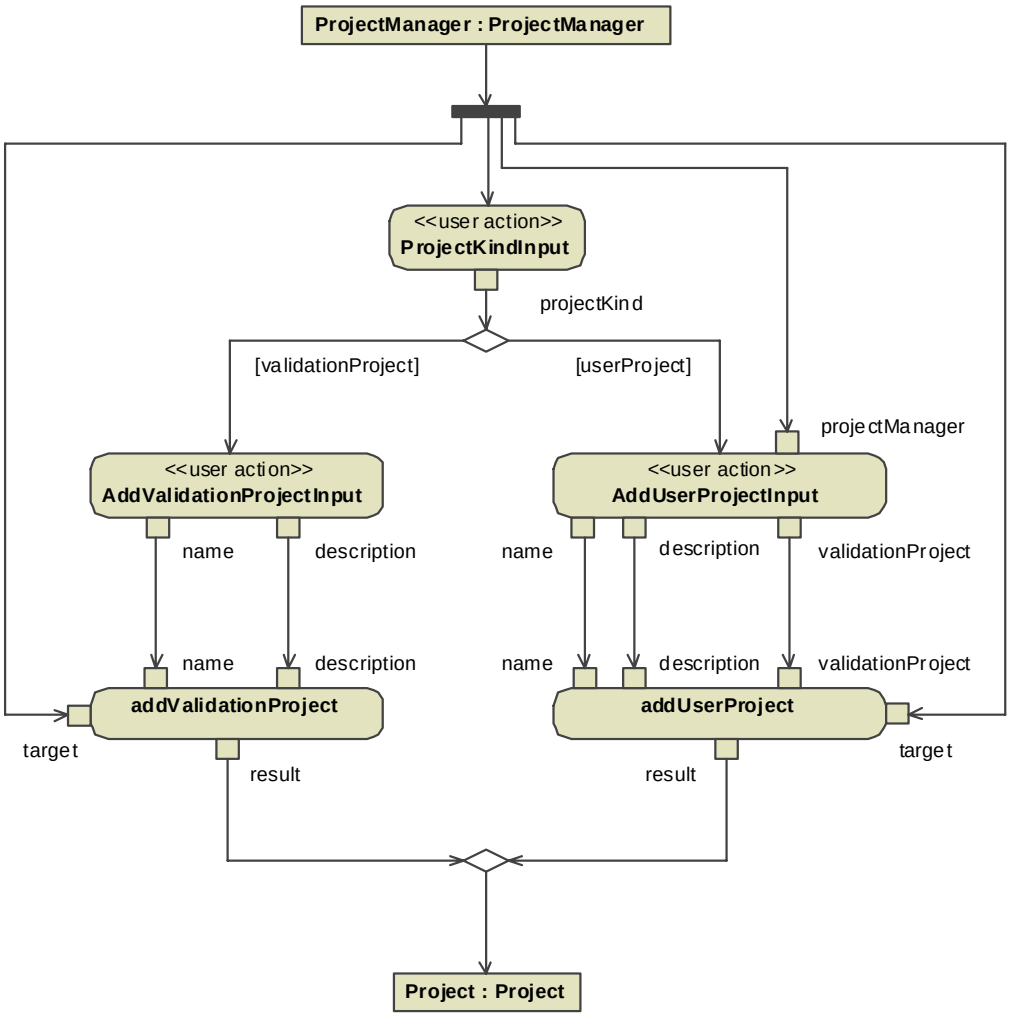


Figure 120. Manually refined process flow for process *AddProject*

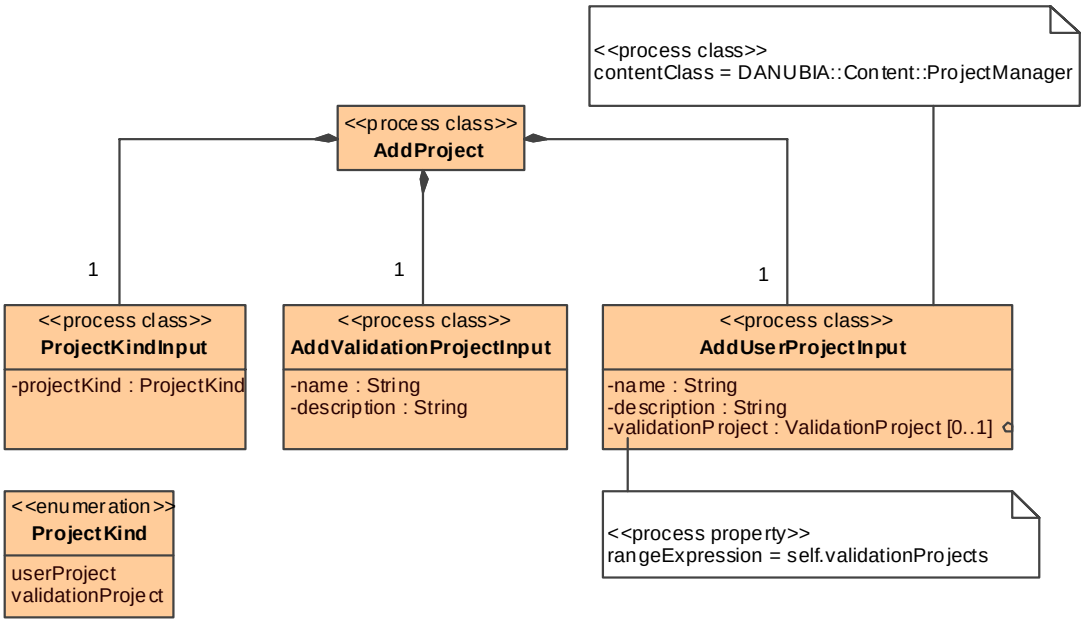


Figure 121. Manually specified process data for process *AddProject*

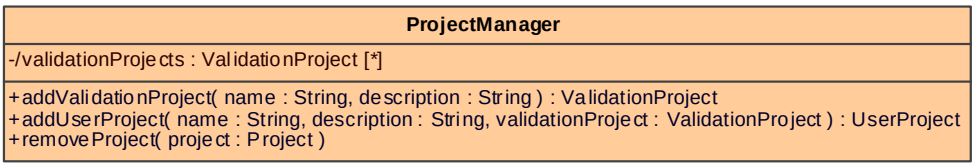


Figure 122. Refined content model for process *AddProject*

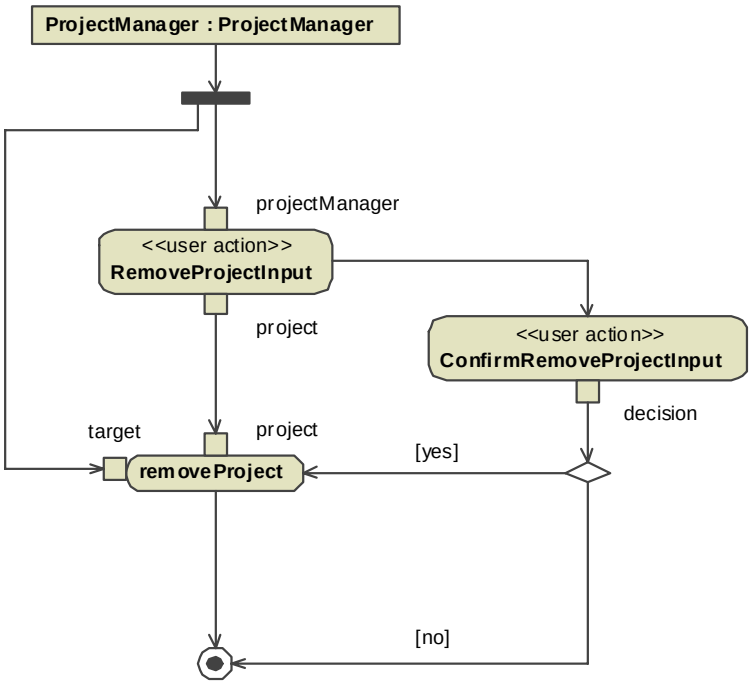


Figure 123. Manually refined process flow for process *RemoveProject*

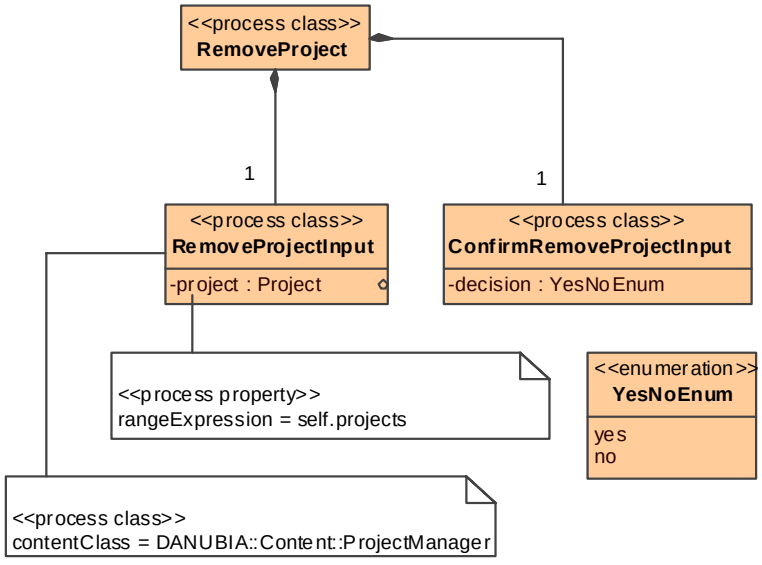


Figure 124. Manually refined process data for process *RemoveProject*

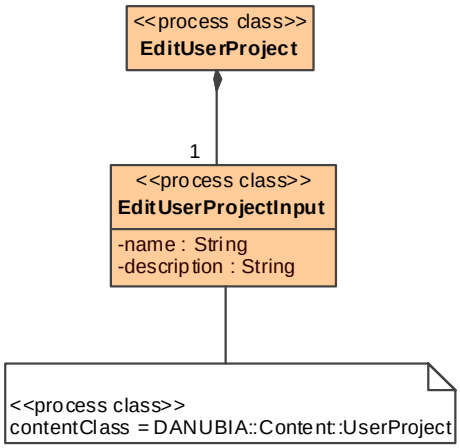


Figure 125. Manually refined process data for process *EditUserProject*

6.1.5 Presentation

The presentation model defines the layout for the underlying navigation and process models, as presented in 4.6. Presentation classes represent Web pages and are composed of user interface elements and other presentation classes.

6.1.5.1 Results of Transformation NavigationAndProcess2Presentation

The transformation *NavigationAndProcess2Presentation* depicted in Figure 126 automatically derives a presentation model from the navigation model and the process model, see 4.6.2. For navigation classes, menus and indices in the navigation model a presentation class is constructed by the rules *NavigationClass2PresentationClass*, *Menu2PresentationClass* and *Index2PresentationClass*. The rule *ProcessClass2PresentationClass* creates a presentation class for each process class in the process model. For each attribute of a node a corresponding presentation property with the type of a user interface element is created by the former rules.

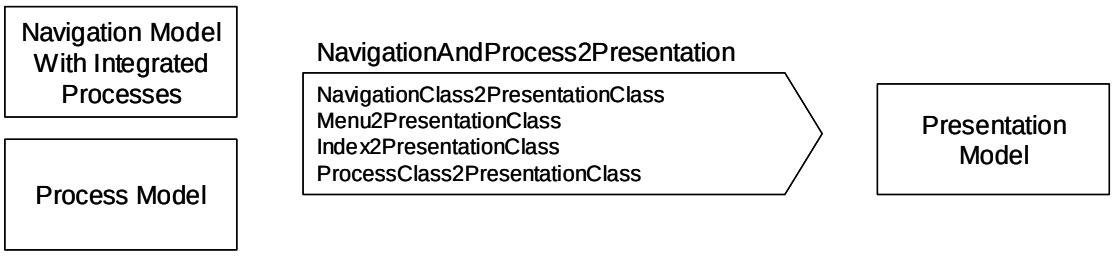


Figure 126. Transformation *NavigationAndProcess2Presentation*

A composite aggregation in the navigation model is mapped to a corresponding composite aggregation in the presentation model, such as for example the composite aggregation between navigation class *ProjectManager* and index *ProjectIndex* depicted in Figure 104 is mapped to a composite aggregation between the corresponding presentation classes, see Figure 127.

Links in the navigation model are mapped to anchors in the presentation model. The multiplicities of the corresponding presentation properties correspond to the multiplicities in the navigation model, see again Figure 127.

Attributes of navigation classes are mapped to text elements, such as for example the attributes *id*, *name* and *description* of a user project as depicted in Figure 130. The text element is generic in the sense that it is assumed that all kind of attribute types can be converted to a textual representation.

Attributes of process data classes representing process data are mapped to input elements. Presentation classes must not be defined for the other process classes which represent processes as a whole from the navigation model with integrated processes. Primitive types are mapped to text input elements, enumeration types to enumeration input elements and all other types are mapped to selection elements. See for example the process data classes for the process *AddProject* depicted in Figure 128.

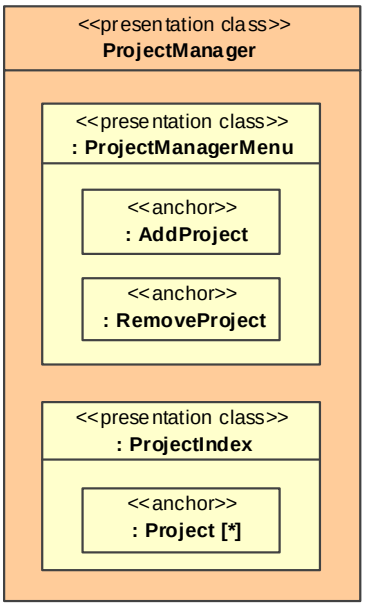


Figure 127. Automatically derived presentation classes for the navigation class *ProjectManager*, the menu *ProjectManagerMenu* and the index *ProjectIndex*

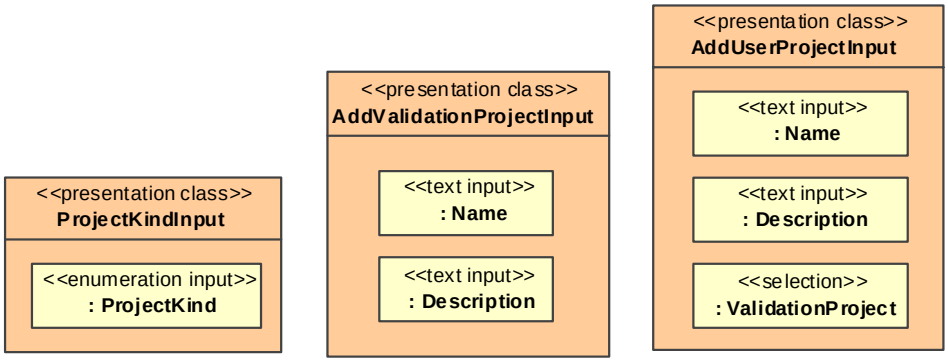


Figure 128. Automatically derived presentation classes for the process classes *ProjectKindInput, AddValidationProjectInput* and *AddUserProjectInput* of process *AddProject*

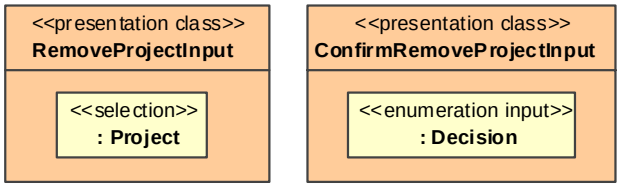


Figure 129. Automatically derived presentation classes for the process classes *RemoveProjectInput* and *ConfirmRemoveProjectInput* of process *RemoveProject*

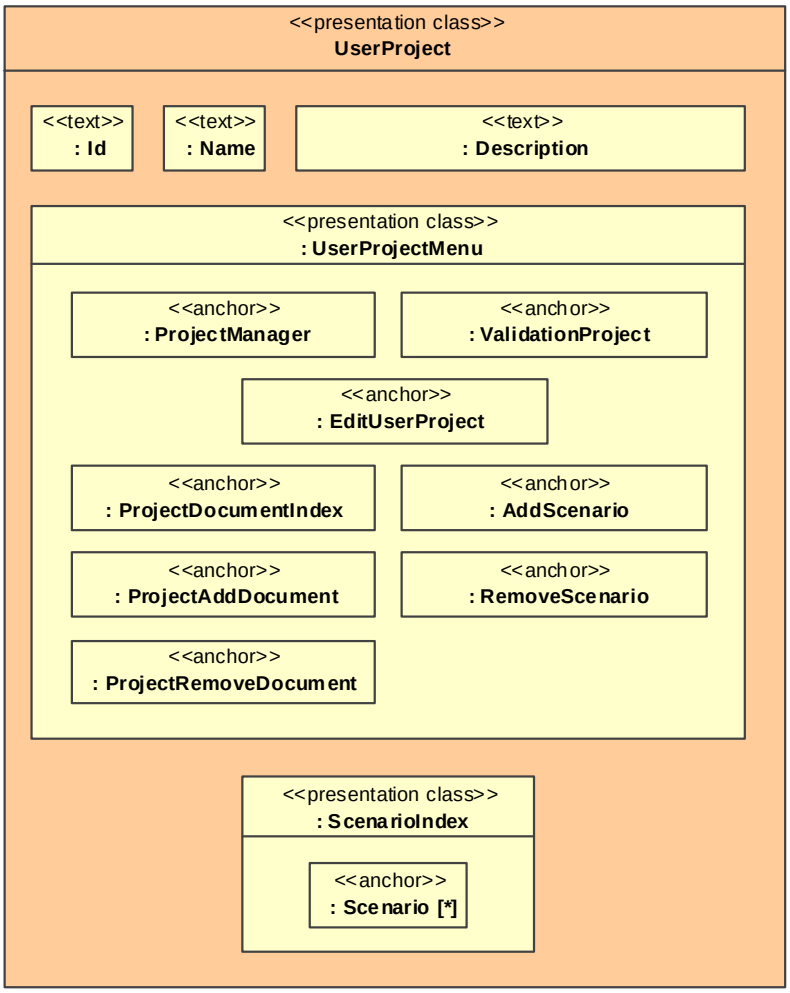


Figure 130. Automatically derived presentation classes for the navigation class *UserProject*, the menu *UserProjectMenu* and the index *ScenarioIndex*

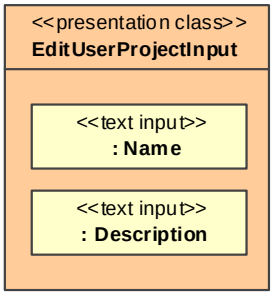


Figure 131. Automatically derived presentation class for the process class *EditUserProjectInput* of process *EditUserProject*

6.1.5.2 Manual Refinement

After the automatical derivation of the presentation model for the case study as presented in the last section some manual refinements have been made as described in 4.6.3, and the resulting final presentation model is presented in this section.

The screenshots in Figure 132 demonstrate the differences between the automatically derived and the manually refined presentation classes for the navigation class *ProjectManager* and the index *ProjectIndex.* First, a static element *Caption* with a welcome message has been added, see Figure 133 for the corresponding manually refined Web pages. Additionally, the format expression of the anchor contained in the project index has been set to the value *"Project #${id} - ${name}"* to provide a meaningful labeling of the index items.

In a similar way the format has been set for all other anchor elements contained in index presentation classes and for selection elements. Additionally, a CSS style definition has been applied to the static text elements added to the presentation classes that are displayed when the user has to confirm that something should be deleted, see Figure 134. The style definition *"color:red"* results in rendering the text of the caption element in red.

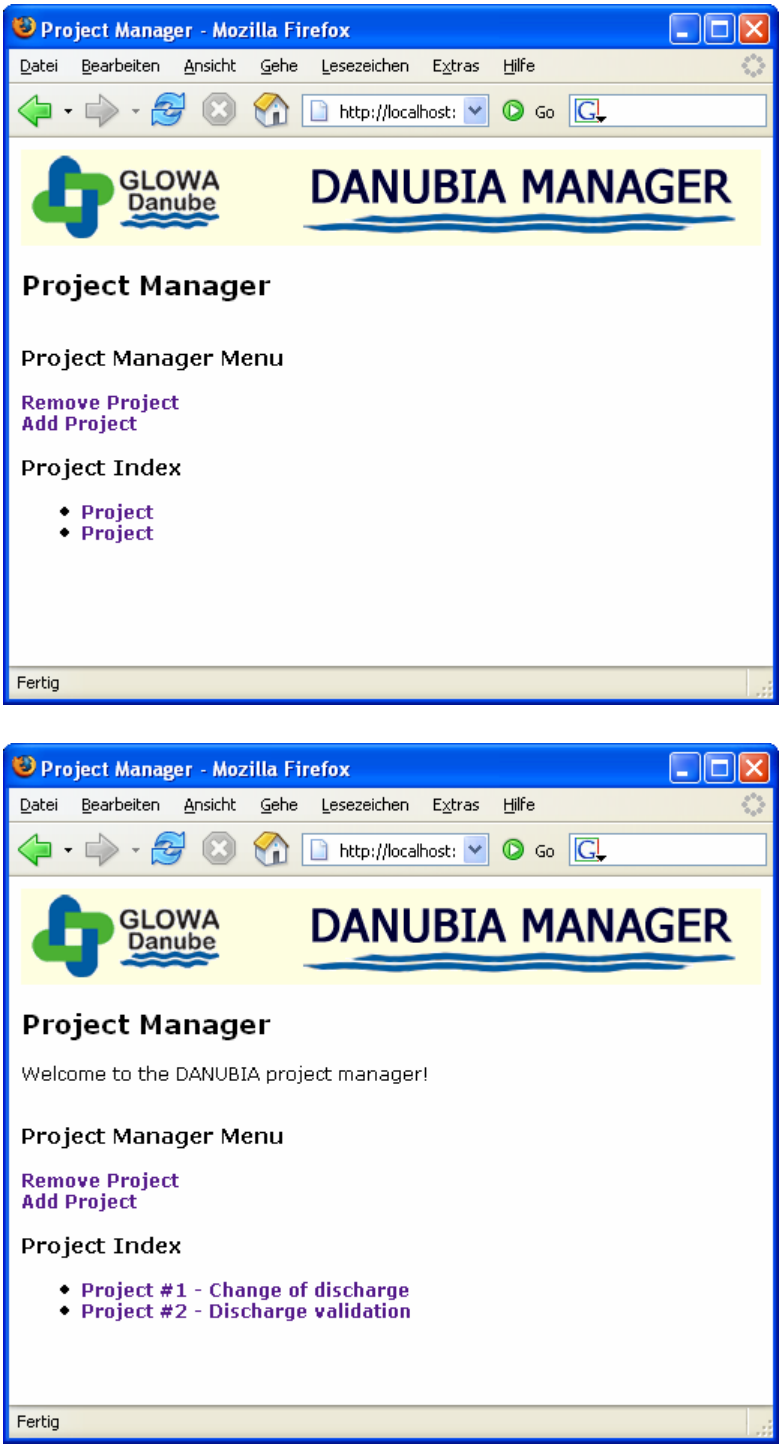


Figure 132. Differences between the automatically derived (above) and the manually refined (below) presentation classes for the navigation class *ProjectManager* and the index *ProjectIndex*

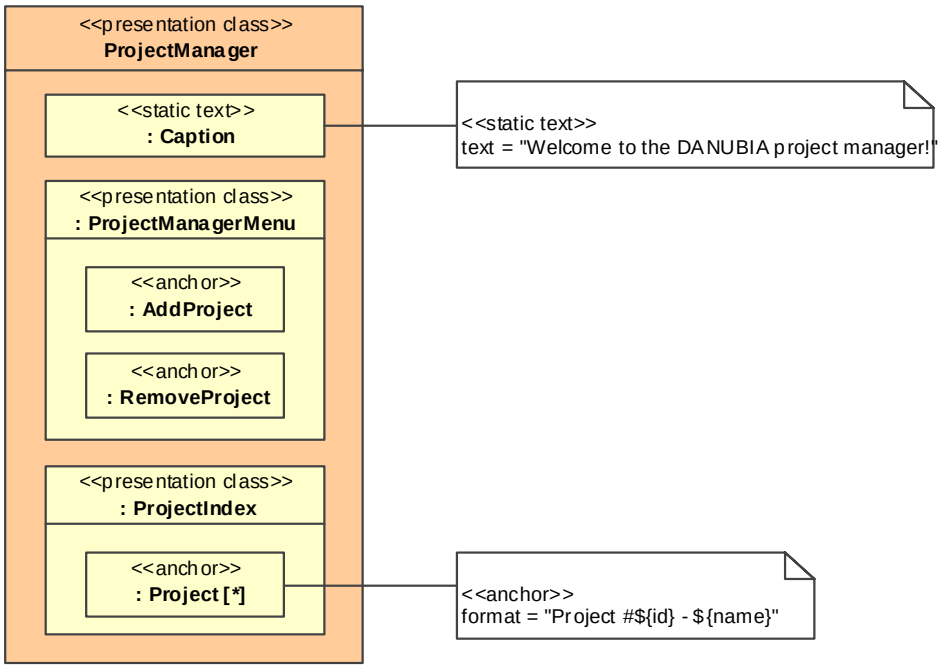


Figure 133. Manually refined presentation classes *ProjectManager* and *ProjectIndex*

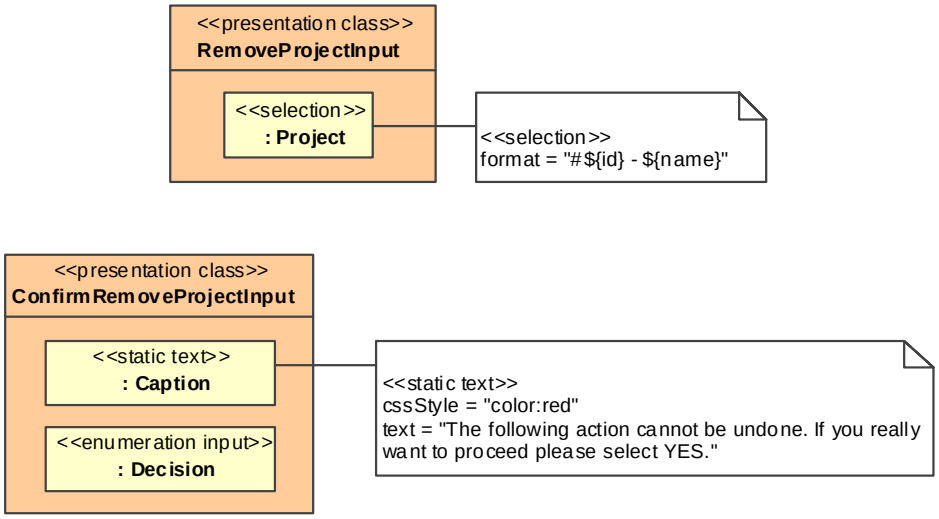


Figure 134. Manually refined presentation classes *RemoveProjectInput* and *ConfirmRemoveProjectInput*

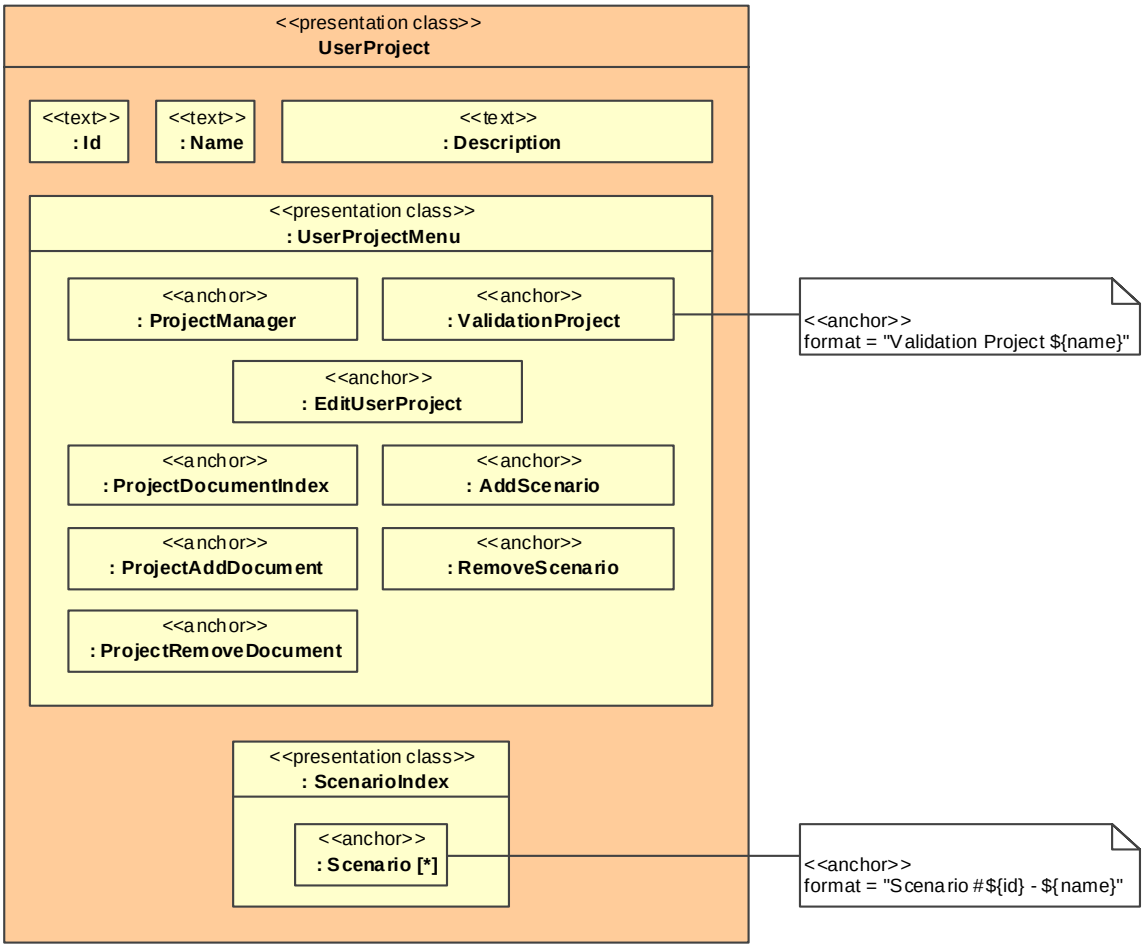


Figure 135. Manually refined presentation classes *UserProject* and *ScenarioIndex*

6.2 Platform Specific Implementation

In this section the model driven platform specific implementation of the case study is presented following the approach described in chapter 5. The following sections demonstrate how the platform independent models for each of the concerns of a Web application presented in the previous section are transformed to code.

6.2.1 Content

The DANUBIA system is not a conventional Web application due to its nature as an environmental simulation system and some technological constraints, which are discussed in the following.

Simulations can be run on a variety of platforms, ranging from laptops or desktop computers to cluster or grid computing infrastructures. Therefore, one important requirement is that simulations can be run offline, i.e. without a user interface, which requires reading configuration data from configuration files. Due to the same reason, a database cannot be used for the simulation configuration.

The idea for the implementation of the content model is to use executable instances of JavaBeans, i.e. lightweight components, to represent the data and functionality for the administration and configuration of the DANUBIA system. The JavaBeans code is generated from the content model as presented in the next section. JavaBeans instances are stored in an XML file by using the *BeanFactory* facility provided by the Spring framework[12]. This XML file is used on the one hand instead of a database for persistence of the beans which are manipulated by the runtime environment. On the other hand it can be read offline by the DANUBIA core system for the configuration of simulation runs. Thus, the DANUBIA user interface can be used for online and offline simulation runs. The system is online when a RMI network connection to the core system exists. The core system can then be triggered directly from the user interface to start a simulation run as sketched in Figure 136. When the user clicks on the corresponding link for the *Start Simulation* process, then this request is delegated to the generic runtime environment presented in 5.1.2. This leads to the execution of the corresponding call operation action *startSimulation* within the Web process engine, and in consequence to the invocation of the corresponding method of the JavaBean for the simulation run. The JavaBean delegates the call to the DANUBIA core system by invoking the method *startSimulation* on the remote interface *DanubiaServerAccess*. The corresponding implementation calls the method *loadConfiguration* of the class *ConfigurationAdmin* which results in loading the XML bean definition file. If the core system is offline, then the simulation has to be started manually as sketched in Figure 137. In contrast to the online scenario the user manually starts the console application *DanubiaCommander* and supplies the command “startSimulation” and the simulation id on the command line. The commander then communicates with the core system in exactly the same way as from the generic runtime environment.

[12] The technique used for representing JavaBeans instances is identical to the technique used for JavaBeans that represent configuration data of the runtime environment as presented in 5.1.3. The fundamental difference is that here JavaBeans represent model elements (model level) while the JavaBeans used for configuration data represent metamodel elements (metamodel level).

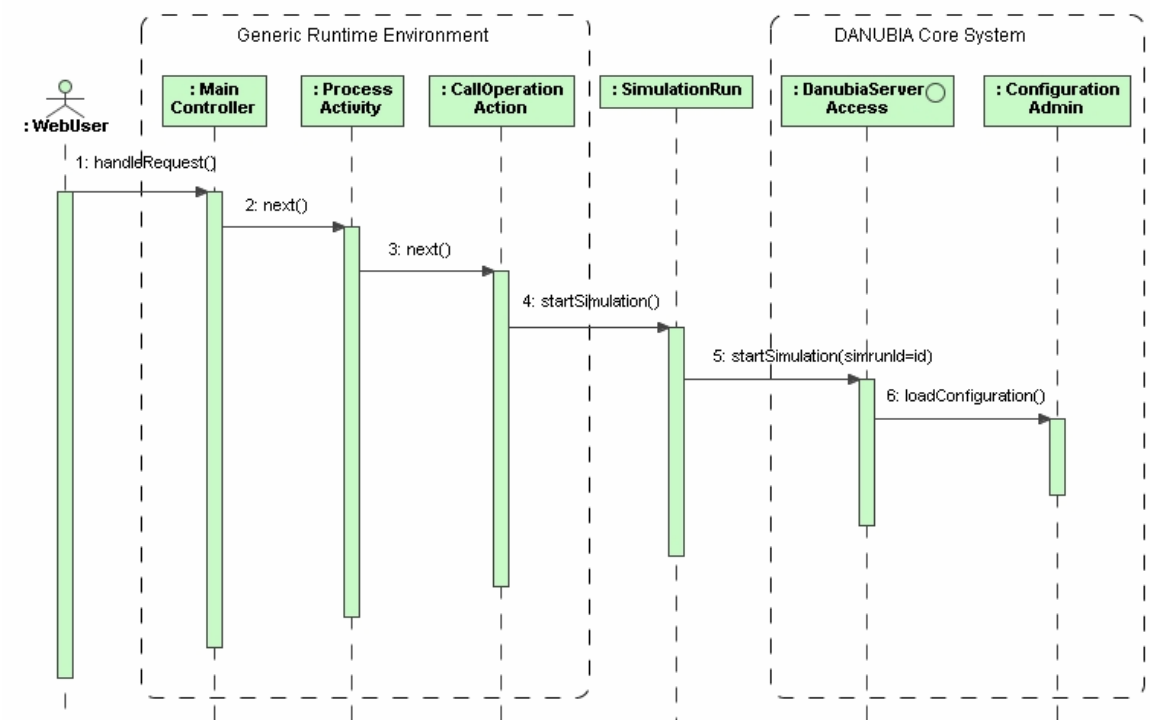


Figure 136. Online starting of a simulation run

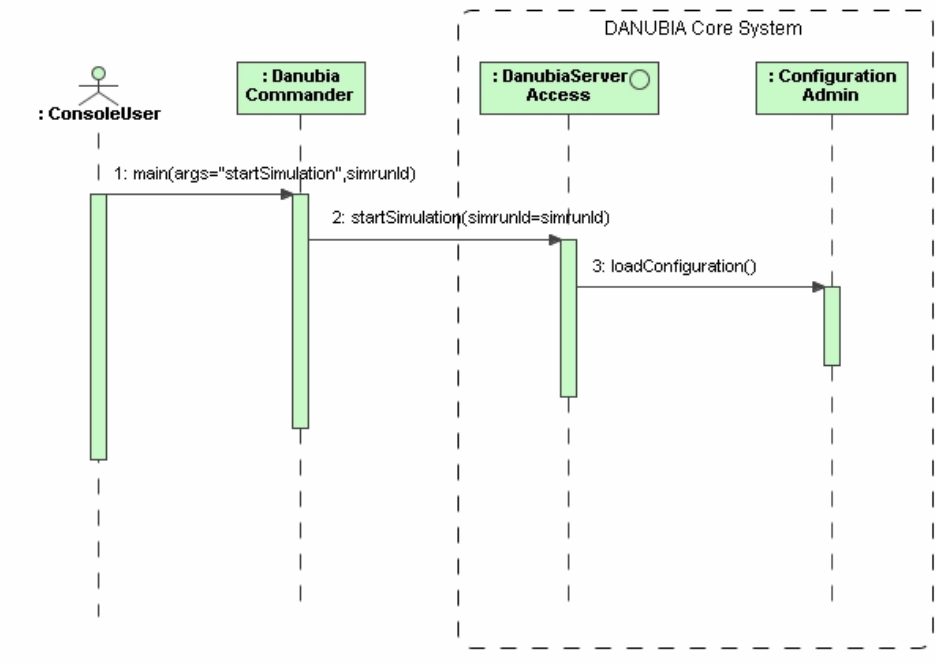


Figure 137. Offline starting of a simulation run

6.2.1.1 Results of Transformation Content2JavaBeans

As presented in 5.2, the transformation *Content2JavaBeans* depicted in Figure 138 transforms the content model to a Java model representing JavaBeans. Each content class is mapped to a Java class by the rule *Class2Class*. The rule *Enumeration2Enumeration* maps each content enumeration to a Java enumeration. Further, for each owned attribute of a content class a corresponding Java field together with a getter and a setter method is generated by the rule *Property2ClassMembers*. All fields are properly initialized. For collection types the corresponding parameterized Java collection types are used. For each operation in the content model a corresponding Java method is generated by the rule *Operation2Method* with an empty method body which has to be completed manually.

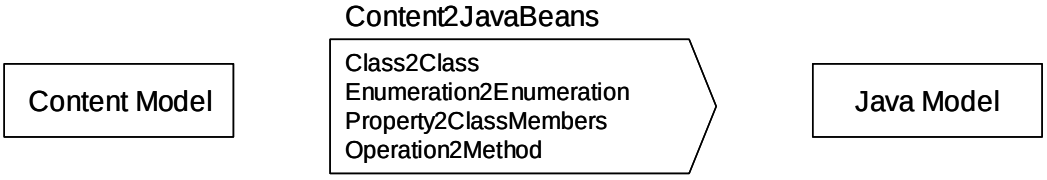


Figure 138. Transformation *Content2JavaBeans*

The resulting Java model is then serialized to code as explained in 5.2.4. The following code sample shows the generated source code for the content class *Project*. Java fields and the corresponding getter and setter methods were generated for the non multi-valued attributes *id*, *name*, *description* and *projectManager*. For the multi-valued attribute *documents* a parameterized Java *ArrayList* is used. The two operations *addDocument* and *removeDocument* were generated with an empty method body.

```
package danubia.content.beans;

public abstract class Project {
    private String _id = "";
    private String _name = "";
    private String _description = "";

    public String getId() {
        return _id;
    }

    public void setId(String _id) {
        this._id = _id;
    }
```

```
    public String getName() {
        return _name;
    }

    public void setName(String _name) {
        this._name = _name;
    }

    public String getDescription() {
        return _description;
    }

    public void setDescription(String _description) {
        this._description = _description;
    }

    private danubia.content.beans.ProjectManager _projectManager;

    public danubia.content.beans.ProjectManager getProjectManager() {
        return _projectManager;
    }

    public void setProjectManager(danubia.content.beans.ProjectManager _projectManager) {
        this._projectManager = _projectManager;
    }

    private java.util.List<danubia.content.beans.Document> _documents =
        new java.util.ArrayList<danubia.content.beans.Document>();

    public java.util.List<danubia.content.beans.Document> getDocuments() {
        return _documents;
    }

    public void setDocuments(java.util.List<danubia.content.beans.Document> _documents) {
        this._documents = _documents;
    }

    public void removeDocument(danubia.content.beans.Document _document) {
    }

    public danubia.content.beans.Document addDocument(String _author, String _title,
            String _abstract) {
        return null;
    }

}
```

6.2.1.2 Manual Refinement

The automatically generated JavaBeans source code for the content model has to be completed by the developer by implementing the body of all operations in the content model. Continuing with the example from the previous section the methods *addDocument* and *removeDocument* have to be implemented. The following code sample shows a possible implementation of these methods.

```
package danubia.content.beans;

public abstract class Project {

    // …
    // manually refined code:
    public void removeDocument(danubia.content.beans.Document _document) {
        documents.remove( _document);
    }

    public danubia.content.beans.Document addDocument(String _author, String _title,
            String _abstract) {
        Document d = new Document();
        d.setId( "1" );
        d.setAuthor( _author );
        d.setTitle( _title );
        d.setAbstract( _abstract );
        d.setProject( this );
        _documents.add( d );
        return d;
    }

}
```

6.2.2 Navigation

As discussed in 5.4 the runtime environment needs information about the navigation model to handle dynamic navigation, i.e. to resolve navigation class inheritance. Figure 139 depicts such a situation where dynamic navigation plays are role in the case study. The page for the project manager contains an index of projects. The anchor for each index item points to the navigation class *Project*. At runtime, depending on the type of the content object, this reference to the navigation class *Project* is resolved to the most specific sub navigation class of *Project* whose corresponding content class is compatible with the actual

dynamic content object. Thus, the navigation class *Project* is either resolved to the navigation class *UserProject* or *ValidationProject* as illustrated in the figure. Another example for dynamic navigation is the exit link of the process *AddProject* which also leads to the target navigation class *Project*, see Figure 108.

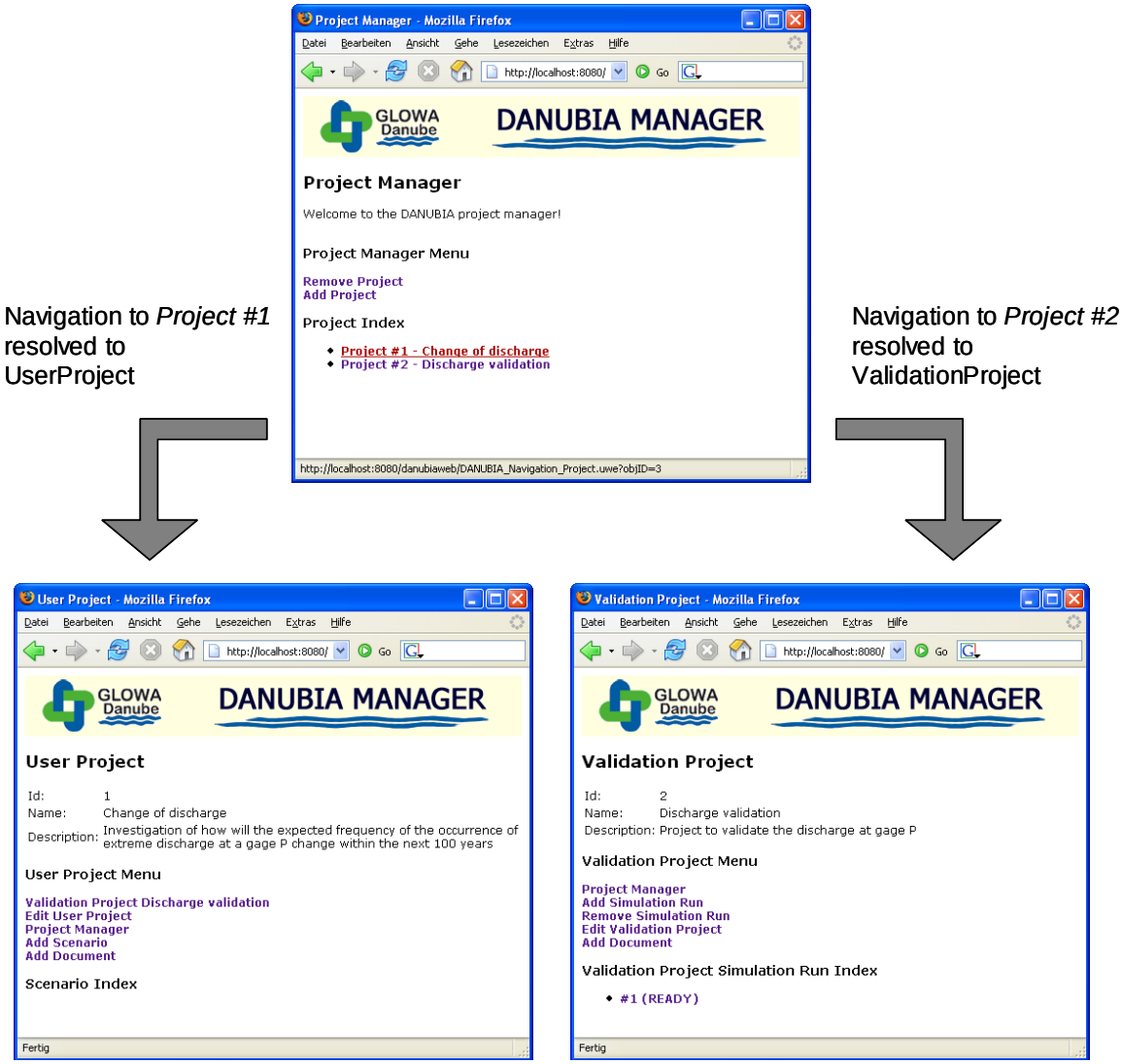


Figure 139. Screenshots for dynamic navigation to sub navigation classes of *Project*

6.2.2.1 Results of Transformation Navigation2Conf

As presented in 5.4, the information about the navigation model concerning inheritance between navigation classes is represented by configuration data of the runtime environment. The navigation model is therefore mapped by the transformation *Navigation2Conf*

depicted in Figure 140 to an XML model which is then serialized to an XML document. Each navigation class is mapped by the rule *NavigationClass2Conf* to an XML bean node. This node is used for the instantiation of the Java class *NavigationClassInfo* (cf. Figure 75) when the Web application is configured by the Spring bean factory. Each such info class is initialized with the name of the navigation class, the Java type of the content class and the specific sub navigation classes.

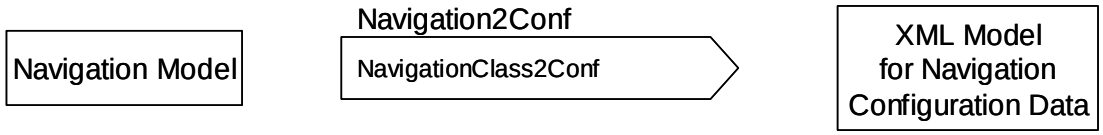


Figure 140. Transformation *Navigation2Conf*

The generated configuration code for the navigation classes of the case study that play a role for dynamic navigation is listed in the following.

```
<bean class="uwe.runtime.NavigationClassInfo" id="DANUBIA_Navigation_Project">
    <property name="name"><value>DANUBIA_Navigation_Project</value></property>
    <property name="specific">
        <list>
            <ref bean="DANUBIA_Navigation_ValidationProject"></ref>
            <ref bean="DANUBIA_Navigation_UserProject"></ref>
        </list>
    </property>
    <property name="contentClass">
        <value>danubia.content.beans.Project</value>
    </property>
</bean>

<bean class="uwe.runtime.NavigationClassInfo" id="DANUBIA_Navigation_UserProject">
    <property name="name"><value>DANUBIA_Navigation_UserProject</value></property>
    <property name="specific">
        <list>
        </list>
    </property>
    <property name="contentClass">
        <value>danubia.content.beans.UserProject</value>
    </property>
</bean>

<bean class="uwe.runtime.NavigationClassInfo" id="DANUBIA_Navigation_ValidationProject">
    <property name="name"><value>DANUBIA_Navigation_ValidationProject</value></property>
    <property name="specific">
```

```
        <list>
        </list>
    </property>
    <property name="contentClass">
        <value>danubia.content.beans.ValidationProject</value>
    </property>
</bean>
```

6.2.2.2 Manual Refinement

Manual refinement of the automatically generated navigation configuration data is generally not necessary.

6.2.3 Process

As presented in 5.5 processes are executed in the runtime environment by a specialized implementation of UML activities which is automatically configured to execute application specific processes. An example for the corresponding configuration data is given in 6.2.3.1.

In order to demonstrate how processes are executed in the runtime environment, the execution of the process *RemoveProject* depicted in Figure 123 is presented in the following. For a description of the algorithm for executing processes in the runtime environment see 5.5.1. On the one hand the relevant log output from the runtime environment is listed, and on the other hand a series of figures showing the token state at each step of the process execution is presented. Additionally, the pages shown to the user during the execution of the process are presented. The execution of the process can be split into three parts. Each part comprises the automatic execution of the process within the runtime environment until the next user input is required or the process has terminated.

In the first part the process is started and an object token holding the project manager object is placed in the input activity parameter node (Figure 143). Then this token is moved to the fork node. The duplicated object tokens are placed in the target input pin of the call operation action and in the project manager input pin of the user action *RemoveProjectInput* (Figure 144). The user action indicates that it is waiting for input from the user and the corresponding page is shown (Figure 141). For the first part the following log output was produced by the runtime environment:

```
[uwe.runtime.MainController] - Request URI: danubiaweb/DANUBIA_Process_RemoveProject.uwe
[uwe.runtime.MainController] - Starting process RemoveProject
[uwe.runtime.process.ActivityParameterNode] - ProjectManager: Received ObjectToken
```

```
    danubia.content.beans.ProjectManager
[uwe.runtime.MainController] - Executing next step of process RemoveProject
[uwe.runtime.process.ActivityParameterNode] - ProjectManager: Removing token
[uwe.runtime.process.ForkJoinNode] - Received ObjectToken
    danubia.content.beans.ProjectManager
[uwe.runtime.process.InputPin] - target: Received ObjectToken
    danubia.content.beans.ProjectManager
[uwe.runtime.process.InputPin] - projectManager: Received ObjectToken
    danubia.content.beans.ProjectManager
[uwe.runtime.process.UserAction] - RemoveProjectInput: Running - waiting for input
```

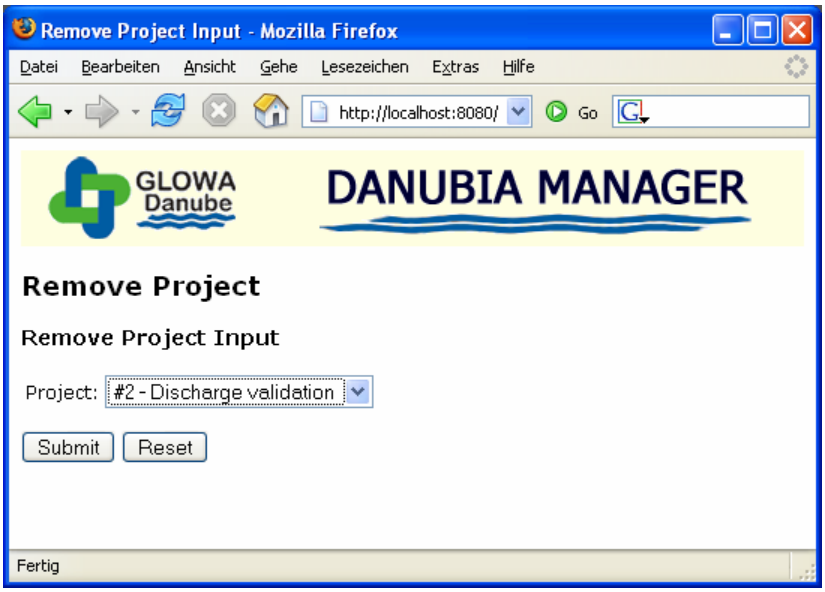


Figure 141. Resulting page after the first part of executing the process *RemoveProject*

After the selection of the validation project #2 and pressing the submit button, the second part of the execution begins. First, an object token for the selected project is placed in the output pin of the user action *RemoveProjectInput* and a control token is offered to the user action *ConfirmRemoveProjectInput* (Figure 145). The object token then moves to the project input pin of the call operation action and the control token triggers the execution of the user action *ConfirmRemoveProjectInput* (Figure 146). This user action indicates that it is waiting for input from the user and the corresponding page is shown (Figure 142). For the second part the following log output was produced by the runtime environment:

```
[uwe.runtime.MainController] - Request URI: /danubiaweb/__processinput__.uwe
[uwe.runtime.MainController] - Executing next step of process RemoveProject
[uwe.runtime.process.OutputPin] - project: Received ObjectToken
    danubia.content.beans.ValidationProject
[uwe.runtime.process.UserAction] - RemoveProjectInput: State finished
```

```
[uwe.runtime.process.InputPin] - projectManager: Removing token
[uwe.runtime.process.UserAction] - ConfirmRemoveProjectInput: Received ControlToken
[uwe.runtime.process.OutputPin] - project: Removing token
[uwe.runtime.process.InputPin] - project: Received ObjectToken
    danubia.content.beans.ValidationProject
[uwe.runtime.process.UserAction] - ConfirmRemoveProjectInput: Running - waiting for input
```

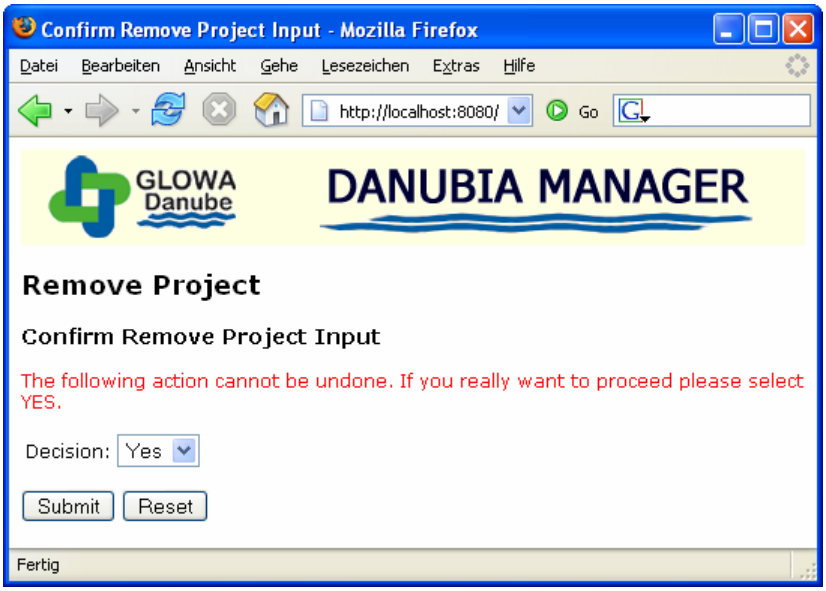


Figure 142. Resulting page after the second part of executing the process *RemoveProject*

After the user selects “yes” and presses the submit button the third and last part of the process execution starts. An object token with the value of the user decision is placed at the output pin of the user action *ConfirmRemoveProjectInput* (Figure 147). The decision node then offers this token to both outgoing edges but only the edge with the guard “yes” accepts the token. Thus, this token triggers the execution of the call operation action (Figure 148). After invoking the corresponding method *removeProject* of the project manager class a control token is offered to the outgoing edge (Figure 149). Finally, this control token is placed at the activity final node and the execution of the process terminates (Figure 150). Because the process *RemoveProject* has no exit link, the project manager page from which the process was invoked is shown again to the user.

```
[uwe.runtime.MainController] - Request URI: /danubiaweb/__processinput__.uwe
[uwe.runtime.MainController] - Executing next step of process RemoveProject
[uwe.runtime.process.OutputPin] - decision: Received ObjectToken java.lang.String
[uwe.runtime.process.UserAction] - ConfirmRemoveProjectInput: Removing token
[uwe.runtime.process.OutputPin] - decision: Removing token
[uwe.runtime.process.DecisionMergeNode] - Received ObjectToken java.lang.String
```

[uwe.runtime.process.CallOperationAction] - Received ControlToken
[uwe.runtime.process.CallOperationAction] - Invoking method removeProject
[uwe.runtime.process.InputPin] - project: Removing token
[uwe.runtime.process.InputPin] - target: Removing token
[uwe.runtime.process.CallOperationAction] - Removing token
[uwe.runtime.process.ActivityFinalNode] - Received ControlToken
[uwe.runtime.MainController] - Process RemoveProject has terminated

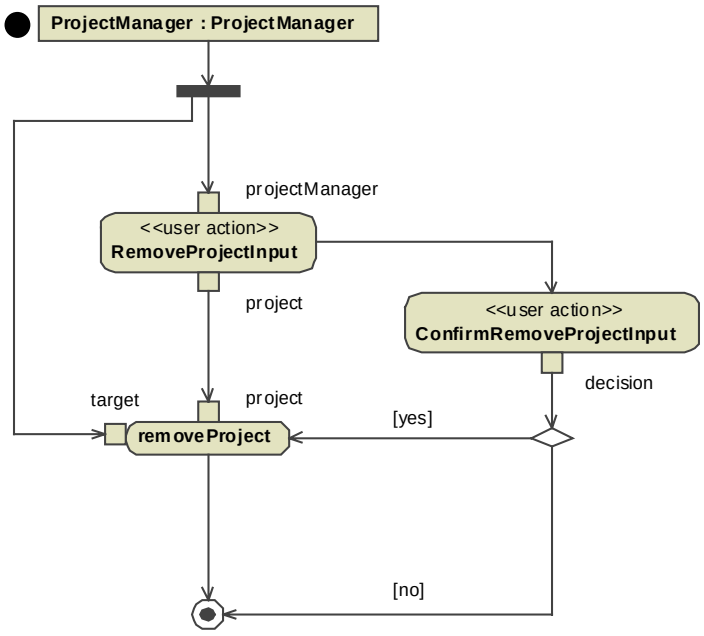


Figure 143. Token flow when executing process *RemoveProject* – step 1

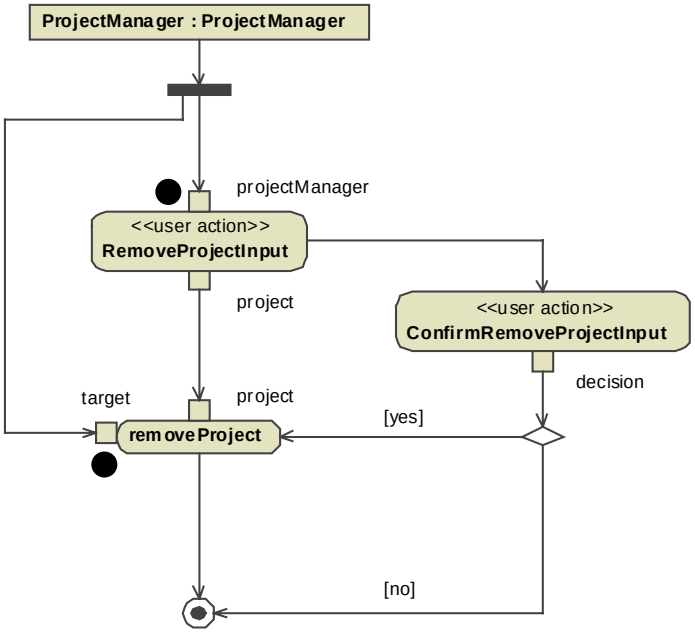


Figure 144. Token flow when executing process *RemoveProject* – step 2

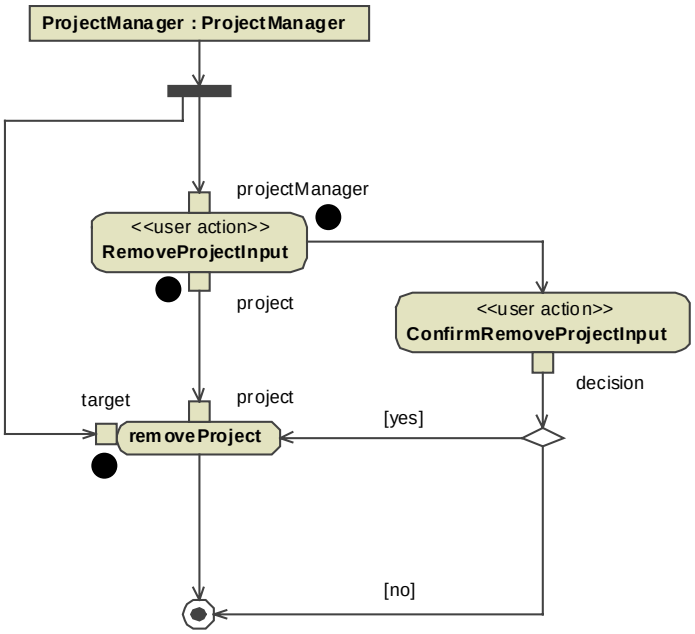


Figure 145. Token flow when executing process *RemoveProject* – step 3

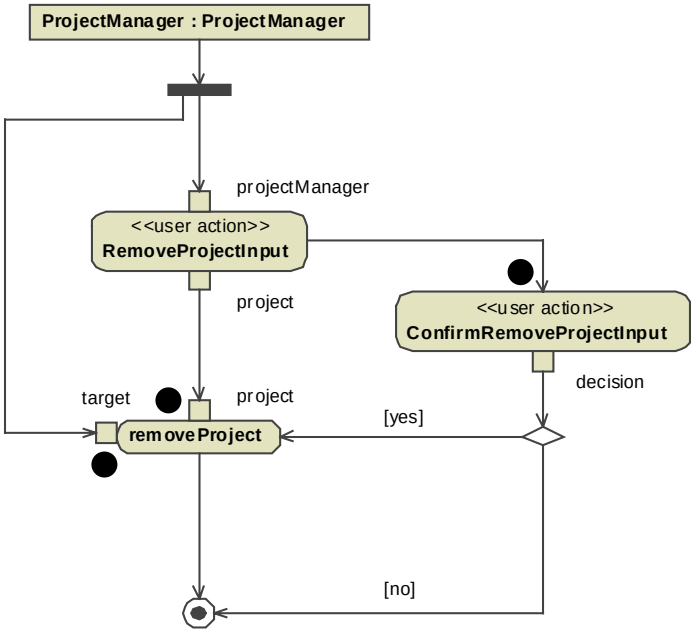


Figure 146. Token flow when executing process *RemoveProject* – step 4

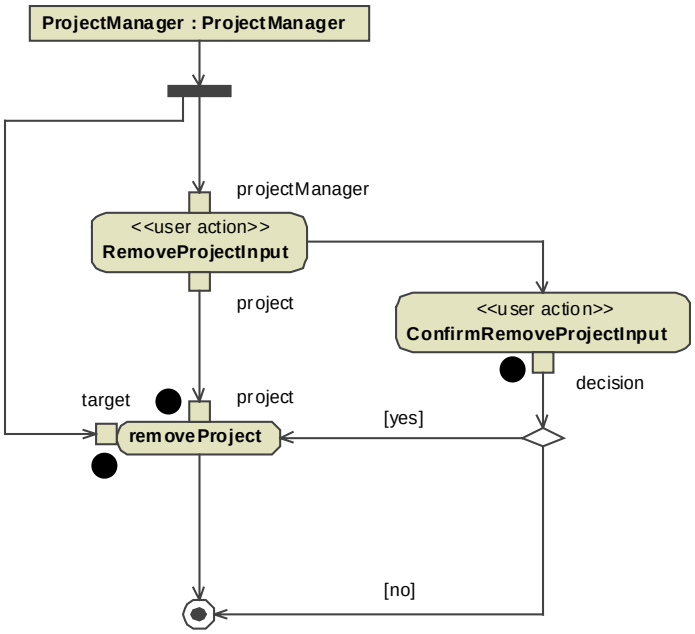


Figure 147. Token flow when executing process *RemoveProject* – step 5

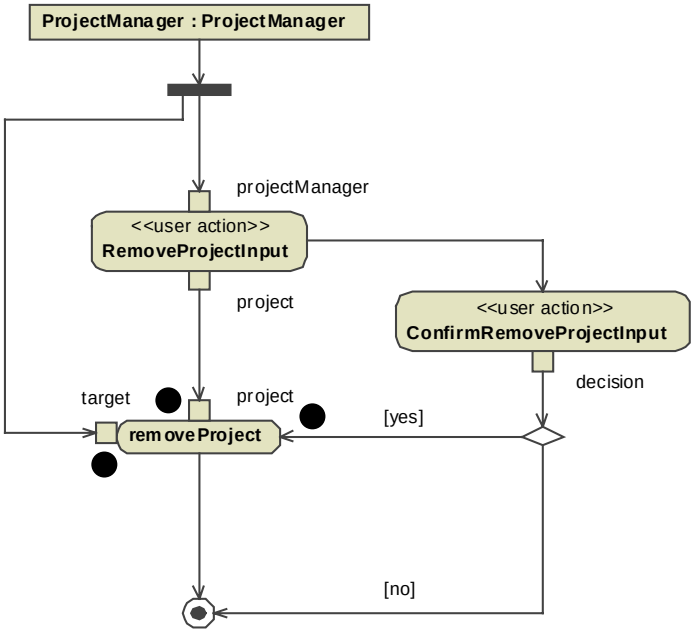


Figure 148. Token flow when executing process *RemoveProject* – step 6

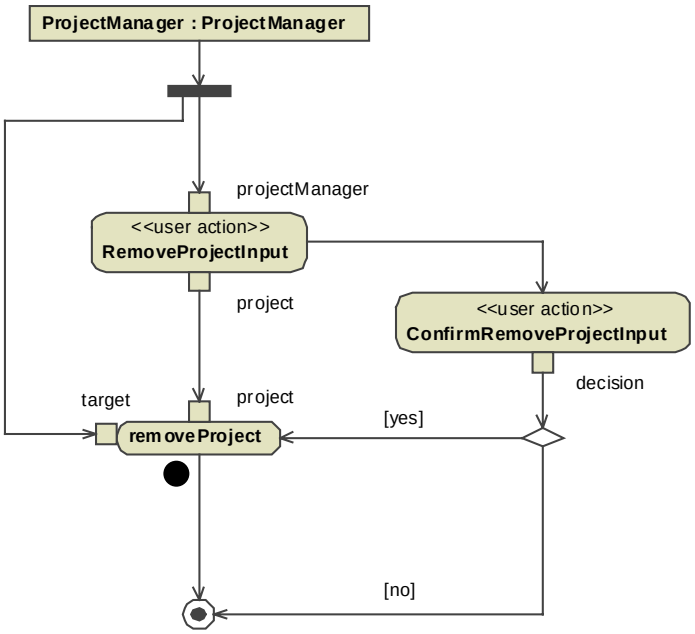


Figure 149. Token flow when executing process *RemoveProject* – step 7

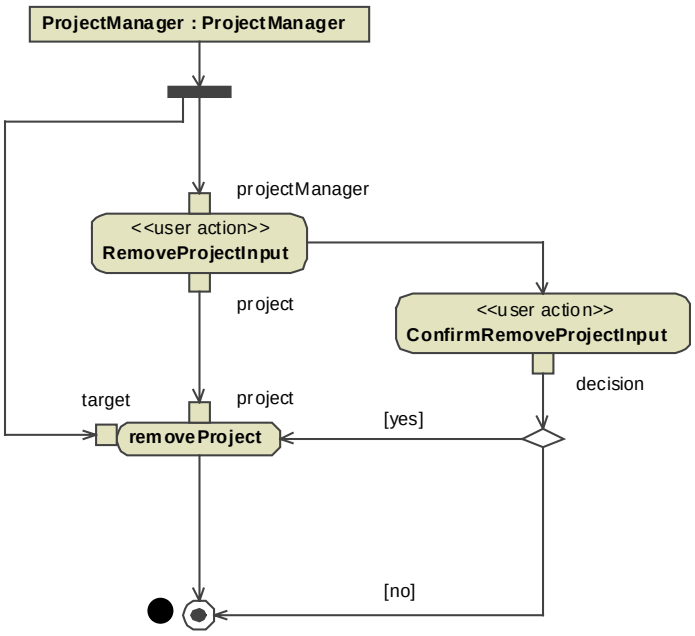


Figure 150. Token flow when executing process *RemoveProject* – step 8

6.2.3.1 Results of Transformation Process2Conf

As presented in 5.5, the process model is mapped to configuration data for the process runtime environment. The process model is therefore mapped by the transformation *Process2Conf* depicted in Figure 151 to an XML model which is then serialized to an XML document. Each process activity is mapped by the rule *ProcessActivity2Conf* to an XML bean node. Other rules are responsible for mapping activity nodes and edges to bean nodes. The corresponding Java classes presented in 5.5 that together represent an executable process are instantiated by the Spring bean factory upon configuration of the Web application.

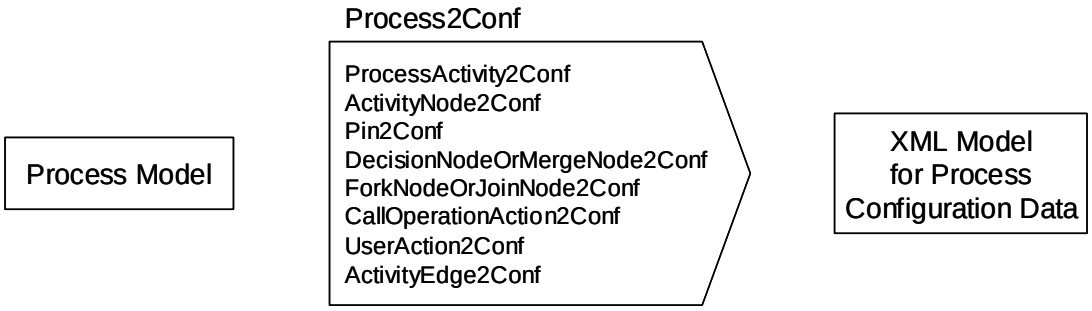


Figure 151. Transformation *Process2Conf*

The following configuration data listing shows the generated code for the process activity and the input activity parameter node of the process *RemoveProject*.

```
<bean class="uwe.runtime.process.ProcessActivity"
    id="ProcessActivity_DANUBIA_Process_RemoveProject_RemoveProject">
    <property name="name"><value>RemoveProject</value></property>
    <property name="processClass">
        <value>DANUBIA_Process_RemoveProject</value>
    </property>
    <property name="entryNode">
        <value>DANUBIA_Navigation_ProjectManager</value>
    </property>
    <property name="activityNodes">
        <list>
            <ref bean="ActivityParameterNode _DANUBIA_Process
                _RemoveProject_RemoveProject_ProjectManager"></ref>
            <ref bean="id_196"></ref>
            <ref bean="UserAction_DANUBIA_Process
                _RemoveProject_RemoveProject_RemoveProjectInput"></ref>
            <ref bean="id_197"></ref>
            <ref bean="id_198"></ref>
            <ref bean="UserAction_DANUBIA_Process
                _RemoveProject_RemoveProject_ConfirmRemoveProjectInput"></ref>
            <ref bean="id_199"></ref>
            <ref bean="OutputPin_DANUBIA_Process
                _RemoveProject_RemoveProject_RemoveProjectInput_project"></ref>
            <ref bean="OutputPin_DANUBIA_Process
                _RemoveProject_RemoveProject_ConfirmRemoveProjectInput_decision"></ref>
            <ref bean="id_200"></ref>
            <ref bean="id_201"></ref>
            <ref bean="InputPin_DANUBIA_Process
                _RemoveProject_RemoveProject_RemoveProjectInput_projectManager"></ref>
        </list>
```

```
    </property>
    <property name="activityEdges">
        <list>
            <ref bean="id_202"></ref>
            <ref bean="id_203"></ref>
            <ref bean="id_204"></ref>
            <ref bean="id_205"></ref>
            <ref bean="id_206"></ref>
            <ref bean="id_207"></ref>
            <ref bean="id_208"></ref>
            <ref bean="id_209"></ref>
            <ref bean="id_210"></ref>
        </list>
    </property>
    <property name="inputParameterNode">
        <ref bean="ActivityParameterNode_DANUBIA_Process
            _RemoveProject_RemoveProject_ProjectManager"></ref>
    </property>
</bean>

<bean class="uwe.runtime.process.ActivityParameterNode" id="ActivityParameterNode_DANUBIA
    _Process_RemoveProject_RemoveProject_ProjectManager">
    <property name="name"><value>ProjectManager</value></property>
    <property name="activity">
        <ref bean="ProcessActivity_DANUBIA_Navigation_RemoveProject_RemoveProject"></ref>
    </property>
    <property name="incoming"><list></list></property>
    <property name="outgoing"><list>…</list></property>
</bean>
```

6.2.3.2 Manual Refinement

Manual refinement of the automatically generated process configuration data is generally not necessary.

6.2.4 Presentation

Java Server Pages are used for the case study as technology for the model driven implementation of the presentation concern. The following sections comprise the automatic generation of JSPs from the presentation model and the customization of the resulting pages.

6.2.4.1 Results of Transformation Presentation2JSP

As presented in 5.6, the transformation *Presentation2JSP* depicted in Figure 152 transforms the presentation model to a JSP model representing Java Server Pages. The transformation comprises three main rules. The rule *PresentationClass2JSP* maps presentation classes to the JSP model. Sub rules of this rule are responsible for mapping presentation classes for specific associated node types, such as for example presentation classes that are associated to navigation classes. The presentation properties owned by a presentation class are mapped by the rule *PresentationProperty2JSP*. User interface elements are mapped by the rule *UIElement2JSP*. Again, sub rules are responsible for mapping specific user interface element types, such as for example text elements. The resulting JSP model is then serialized to JSP pages which can directly be executed in the proposed runtime environment without any modification by the developer.

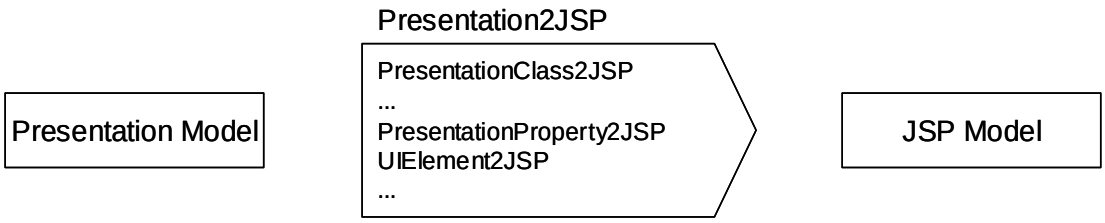


Figure 152. Transformation *Presentation2JSP*

The following code sample shows the generated JSP code for the presentation class *ProjectManager* and serves as an example for the structure of the generated pages.

```
<%@ page language="java" %>
<%@ include file="/WEB-INF/jsp/include.jspf" %>
<html>
    <head>
        <title>Project Manager</title>
    </head>
    <%@ include file="/WEB-INF/jsp/style.jspf" %>
    <body>
        <%@ include file="/WEB-INF/jsp/header.jspf" %>
        <div>
            <h2>Project Manager</h2>
            <p class="" style="">Welcome to the DANUBIA project manager!</p>
            <div>
                <h3>Project Manager Menu</h3>
                <div>
                    <span class="" style="">
```

```
                    <c:if test="${not empty self and ( not empty self.projects )}">
                        <c:set var="obj" scope="request" value="${self}"></c:set>
                        <a href="DANUBIA_Process_RemoveProject.uwe?
                            objID=<%= objID( request ) %>">Remove Project</a>
                    </c:if>
                </span>
            </div>
            <div>
                <span class="" style="">
                    <c:if test="${not empty self and ( true )}">
                        <c:set var="obj" scope="request" value="${self}"></c:set>
                        <a href="DANUBIA_Process_AddProject.uwe?
                            objID=<%= objID( request ) %>">Add Project</a>
                    </c:if>
                </span>
            </div>
        </div>
        <div>
            <h3>Project Index</h3>
            <ul>
                <c:forEach items="${self.projects}" var="self_projects_it">
                    <li>
                        <span class="" style="">
                            <c:if test="${not empty self_projects_it and ( true )}">
                                <c:set var="obj" scope="request"  value="${self_projects_it}"></c:set>
                                <a href="DANUBIA_Navigation_Project.uwe?
                                    objID=<%= objID( request ) %>">
                                    <c:out value='Project #${self_projects_it.id} –
                                        ${self_projects_it.name}' />
                                </a>
                            </c:if>
                        </span>
                    </li>
                </c:forEach>
            </ul>
        </div>
    </div>
  </body>
</html>
```

The first include statement is needed for including some common JSP code, such as for example for the declaration of the JSTL tag libraries. The other include statements are used for customization of the JSPs as discussed in the next section.

The JSP code for each generated presentation class starts with a *h<1+nesting-depth>* tag containing the formatted name of the presentation class, i.e. depending on the nesting depth of the presentation class, a different tag is used, e.g. *h2* for the root presentation class in the containment hierarchy *ProjectManager*, and *h3* for the nested presentation classes *ProjectManagerMenu* and *ProjectIndex*.

Then, for each presentation property of a presentation class, JSP code is embedded in the resulting page, depending on the type of the presentation property. For static elements static code is generated. Output elements are transformed to dynamic JSP code using the *c:out* tag of the Java Standard Tag Library (JSTL). Input elements are transformed to *input* tags. Anchors are mapped to JSP code, for example for the link to the process *RemoveProject*:

```
<c:if test="${not empty self and ( not empty self.projects )}">
    <c:set var="obj" scope="request" value="${self}"></c:set>
    <a href="DANUBIA_Process_RemoveProject.uwe?
        objID=<%= objID( request ) %>">Remove Project</a>
</c:if>
```

The outer *c:if* tag is used to test on the one hand if the target object of the link is valid *("not empty self"*) and on the other hand if the guard condition of the link is fulfilled *("not empty self.projects"*). The variable *self* holds a reference to the actual content object that this page presents. If the conditions are fulfilled, then the inner code is executed. First a variable *obj* is set to the target content class. The JSP scriptlet code *"<%= objID(request) %>"* calls a method *objID* defined within the included *include.jspf* file, which reads the variable *obj* and returns a unique id for the target content object. This id together with the corresponding object is also stored in the session context and allows the runtime environment to resolve the target object when the link to the page *DANUBIA_Process_RemoveProject.uwe* is executed.

6.2.4.2 Manual Refinement

The generated Java Server Pages did not have to be manually refined. However, the appearance of the resulting JSPs can be customized by modifying two files which are included by all pages: the file *header.jspf* is included at the beginning of the *body* tag of each page. Thus, it can be used to apply a common page header to all pages. For the DANUBIA Web application the DANUBIA logo was included. For defining common style definitions for all pages the file *style.jspf* can be modified. In Figure 153 the project manager page without appearance customization is depicted. After inclusion of the DANUBIA

logo in the page header and providing a default style definition the resulting page looks as depicted in Figure 154. It has to be stressed that only files that are included by the generated JSPs were modified. Thus, these modifications are not lost when running the transformation *Presentation2JSP* again.

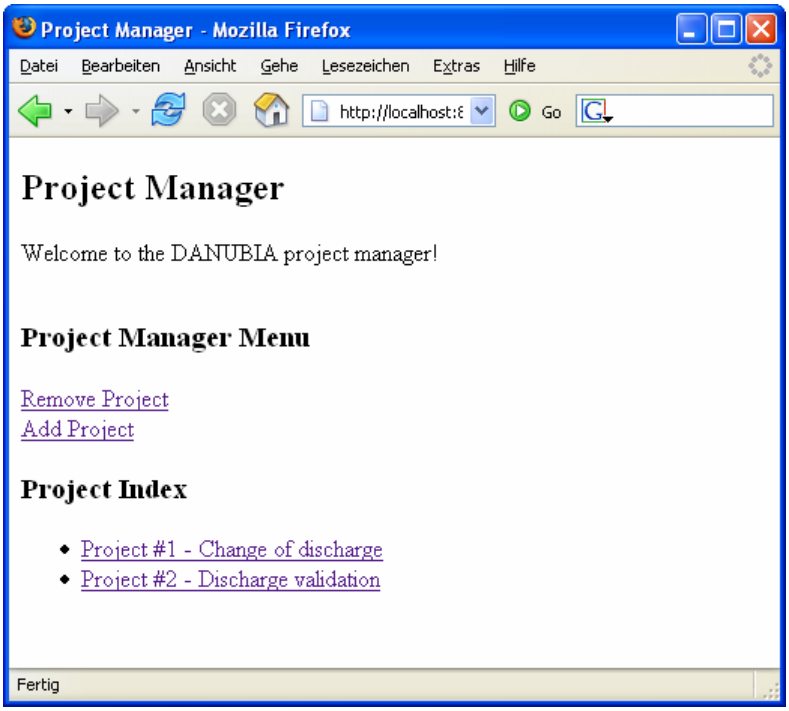


Figure 153. Generated JSP (appearance not yet customized)

Figure 154. Generated JSP (appearance customized)

6.3 Evaluation

This section gives a brief evaluation of the experiences gained from the development of the case study. The vision of the MDA is that applications are modeled at a platform independent level and are transformed by means of model transformations to platform specific implementations. Thus, the essential question is, how much manual refinement was necessary after the generation of the platform specific implementation, because these steps would probably have to be repeated for another platform.

The main development efforts have been on the construction of the platform independent models, which have been elaborated by alternating automatic transformation and manual refinement steps as presented in 6.1. The following manual refinement activities were required:

- Addition of missing details to the content model, such as attribute types or multiplicities
- Designation of a home node within the navigation model

- Assignment of guard conditions for process links
- Definition of process data and flow for complex processes
- Definition of range expressions for process properties with a complex type, i.e. process properties with neither a primitive type nor an enumeration type

On the other hand, due to the choice of Java Beans as technology for the implementation of the content model, the generated Java Beans classes had to be manually refined for all content classes with at least one operation or one derived attribute. The transformation of the remaining models to the corresponding platform specific models for the generic platform did not require manual refinement.

7 CONCLUSION

To conclude this work, in the following the main results are summarized first. Then the limitations of this approach are discussed, and finally an outlook to possible future research is given.

7.1 Results

The overall result of this work is the elaboration of a complete MDA-based approach for Web application development from analysis to the generated implementation. After the evaluation of current transformation approaches the choice of ATL has proven as adequate for this work, most important due to the available tool support for running ATL transformations. Nevertheless, when a fully fledged implementation of QVT becomes available, it might be preferable to use the future standard QVT instead of ATL. The transformations of this work are easily transferable to QVT.

For the platform independent analysis and design a metamodel has been defined as a conservative extension of the UML 2 metamodel together with OCL constraints, expressing well-formedness rules, and a UML profile as notation. Transformation rules have been defined for the systematic stepwise evolution of models. The drawback of using the general modeling language UML, in contrast to a small domain specific language (DSL), is that transformation rules sometimes become quite lengthy due to the complex structure of the UML metamodel from which only a small part is actually needed. Additionally, many OCL constraints have to be defined to ensure the correct use of the modeling elements which are specializations of elements from the UML metamodel.

The decomposition of the transformation to the platform specific implementation along the content, navigation, process and presentation concerns has proven to be useful to reduce the complexity of this transformation and to allow for a higher degree of decoupling between the corresponding technological counterparts. A generic platform based on Spring Web framework extended by a custom runtime environment has been proposed to support

a broad range of different target technologies for the different concerns of a Web application. Two alternative technologies for the implementation of the content model have been presented to show the flexibility of the approach.

The use of UML activities for process flow modeling has proven as too complex to be directly transformable to code. Therefore a process runtime environment was implemented as part of the overall runtime environment to support the execution of UML activities based on the semantics of token flows. The process flow model was therefore transformed to configuration data for the process runtime environment.

Finally, the results of this work have been successfully applied to the DANUBIA case study. Due to the choice of Java Beans as technology for the implementation of the content model, the generated Java Beans classes had to be manually refined by implementing the corresponding operations derived from the content model. On the other hand, the transformation of the remaining models to the corresponding platform specific models for the generic platform did not require manual refinement.

The technical details of this work including the metamodels, the transformation environment and the runtime environment are available on the UWE homepage [UWE].

7.2 Limitations

As already stated in Section 3.1, the fine-grained specification of the behavior of operations in the content model is not considered in this work. In this sense the transformations to platform specific models presented here could be considered as being not fully automatic. This work focuses on the modeling of coarse-grained behavior by the means of a process model which is used to compose the fine-grained behavior, i.e. the invocation of operations in the content model. Nevertheless, several alternative ways exist for the fully executable specification of operations, but the challenge will be to ensure that these specifications are independent of platform specific constructs and that they can be transformed to the platform specific level. UML allows the use of user-defined (textual) languages, so called action languages, for the specification of behavior as discussed in [Mellor02]. An action language can also be used within executable activity diagrams that specify the behavior of an operation.

The development of data-intensive Web applications with this approach can result cumbersome because the create, retrieve, update and delete (CRUD) operations found in data-intensive Web approaches such as for example WebML [Ceri02] are realized in this work

by the more general Web processes. Therefore, for each create or delete operation a corresponding Web process has to be designed and the corresponding service has to be implemented manually. The retrieve and update operations on the other hand can already be handled fully automatically by using an appropriate technology for the content concern that provides a database mapping, such as for example Enterprise Java Beans (EJB). The approach can be improved for data-intensive Web applications by extending the platform independent metamodel with corresponding modeling elements, such as for example special Web process use case types and process activity actions that represent database operations.

One concern of Web applications not included in this work is adaptivity. Adaptive Web applications adapt themselves to dynamic user and context properties, i.e. they allow for personalization and contextualization. User properties comprise the user's preferences, interests or knowledge whereas context properties are related to the environment, e.g. the user location. All concerns presented in this work may be adapted, i.e. content, navigation, presentation and even processes. For an overview of current adaptive approaches see [Kappel03b]. A proposal for the (non model driven) treatment of adaptivity within the integral UWE approach is detailed in [Koch01a]. Because adaptivity is a crosscutting concern it should be addressed with aspect-oriented techniques, thus aspect-orientation in the realm of model-driven development is an important future research topic.

Aspect-orientation provides a way of modularization of concerns that would otherwise be scattered across modules. For an overview over aspect-oriented modeling techniques see [Filman04]. The main tasks that have to be done for the integration of adaptivity using aspect-oriented techniques are on the other hand the appropriate aspect-oriented platform independent modeling of adaptivity, and on the other hand the transformation of these aspects to the platform specific models. This transformation corresponds to the weaving activity when employing aspect-orientation in programming languages. In [Baumeister05] a possible way of modeling adaptivity with aspects within the UWE approach is sketched and the use of aspects for modeling access control for Web applications is presented in [Zhang05]. Another approach called AspectUWA also investigates the combination of aspect-oriented modeling and model-driven development for adaptive Web applications [Schauerhuber06a]. Additionally, the Spring framework provides support for the application of aspect-oriented techniques for Web applications [Spring].

7.3 Future Research

Future Web application approaches should provide enhanced support for the Web 2.0 [O'Reilly05], which is the vaguely defined designation for the recent and still ongoing evolution of the Web. It is not related to a specific technology or a single development, but rather to the perceived synergy effect of a bundle of recent technologies and developments. Technologies for Web applications that can be characterized as Web 2.0 applications are for example Web Services [W3C02] or Ajax [Garrett05].

The broader scope of Web Services is the Service Oriented Architecture (SOA) approach [Dostal05]. The basic idea of the SOA approach is to see the realization of a business process as a composition of services. Hence, the application logic of a system is distributed over several independent and loosely coupled services. Services are provided by service providers and used by service consumers. To find a service some kind of directory facility is necessary. The Service Oriented Architecture approach uses software components [Szyperski02] for providing services. Although different component technologies such as Enterprise JavaBeans (EJB), CORBA or DCOM could be used, Web Services are especially suited for the SOA approach. Web Services make use of XML for service metadata, communication and directory services, which allows the platform independent implementation and the use of the internet as the communication layer. By providing adequate platform specific metamodels and transformations for the content aspect, Web services can be integrated into the approach presented in this work.

The acronym Ajax stands for Asynchronous JavaScript and XML. Ajax incorporates several technologies to close the gap between rich and responsive desktop applications and Web applications by introducing an intermediate layer between the user and the server, the so called Ajax engine. Although this layer still uses the stateless HTTP protocol for the communication with the server it allows asynchronous user interaction with the application. The downside of this approach is a higher Web server load in comparison to traditional Web applications due to the use of a polling mechanism for receiving events from the server. The key technologies of Ajax are a standards-based presentation using XHTML and CSS, dynamic display and interaction using the Document Object Model (DOM), data interchange and manipulation using XML and XSLT, asynchronous data retrieval using XMLHttpRequest and JavaScript for binding everything together [Garrett05]. An already widespread application using Ajax is for example Google Maps[13] which allows the user to

[13] http://maps.google.com

interactively navigate within geographical maps. Although the presentation metamodel in this work is designed for traditional Web applications, Web applications for the Ajax framework could as well be generated from the platform independent models by providing appropriate platform specific metamodels and transformations for the presentation aspect. Nevertheless, for taking full advantage of the features of Ajax, i.e. supporting the development of rich and responsive Web applications, the presentation metamodel as presented here would have to be extended. Additional behavioral models for handling user interface events would allow to model responsive user interfaces for the Ajax framework. Such an extended metamodel would not be generic anymore as only Web applications for the Ajax framework could be generated. On the other hand it would be possible to generate Ajax Web applications as well as traditional desktop applications from the same platform independent models. As Ajax and similar frameworks are gaining relevance and acceptance for the development of Web applications, future investigations should continue examining the model driven development of responsive Web applications.

A further future research topic is the combination of model driven Web engineering with technologies for the Semantic Web. According to [Berners-Lee01], the Semantic Web is an extension of the current web, which better defines the meaning of information, enabling computers and people to work better in cooperation. The strength of the Semantic Web approach is the ability to explicitly represent knowledge by using ontologies and to carry out automated reasoning. SHDM [Lima03] for example is a MDWE approach that maps object-oriented Web application models to ontologies. This allows for example to infer navigation links by using Semantic Web technologies. Another possible application of Semantic Web technologies is to represent the target platform by an ontology as proposed in [Wagelaar05]. This would allow for automatically selecting and configuring a number of reusable model transformations for a concrete platform, using description logics.

8 TABLE OF FIGURES

Figure 1. Development process overview 15
Figure 2. Platform specific implementation using a generic platform 17
Figure 3. MDA Pattern, from [Miller03] 24
Figure 4. Example for a platform specific model and the corresponding code 26
Figure 5. Pattern for model type transformations 27
Figure 6. Metamodeling hierarchy example (adapted from [OMG05a]) 29
Figure 7. Transformation between metamodel and UML profile 32
Figure 8. Relationships between QVT parts [OMG05b] 39
Figure 9. Graphical notation of QVT Relations 42
Figure 10. Transformation metamodel 51
Figure 11. Interoperability between transformation approaches 53
Figure 12. PIM2PSM transformations 58
Figure 13. Decomposed PIM2PSM transformation 59
Figure 14. ATL model handlers 61
Figure 15. The ATL Eclipse plug-in 62
Figure 16. UWEXML process 65
Figure 17. Global UWE process overview (from [Koch06b]) 67
Figure 18. Metamodel Package Structure 74
Figure 19. Relationships between models and metamodels 75
Figure 20. Metamodel for transformation traces 78
Figure 21. Example for transformation trace 79
Figure 22. Example for using transformation traces for incremental update 81
Figure 23. Use of an expression language 83
Figure 24. Metamodel for requirements modeling 87
Figure 25. Analysis content model 89
Figure 26. Use cases for content class ProjectManager 90
Figure 27. Use cases for content class UserProject 90
Figure 28. Transformation Requirements2Content 92
Figure 29. Content model derived by transformation Requirements2Content 93
Figure 30. Illustration of the rules for adding operations 93
Figure 31. Refined design content model 96

Figure 32. Metamodel for navigation modeling (backbone) 99
Figure 33. Metamodel for navigation modeling (access structures) 100
Figure 34. Transformation RequirementsAndContent2Navigation 104
Figure 35. Navigation space model derived by transformation RequirementsAndContent2Navigation 105
Figure 36. Manually refined navigation space model 108
Figure 37. Transformation AddIndices 109
Figure 38. Navigation model with added indices derived by transformation AddIndices 109
Figure 39. Navigation model with refined indices 111
Figure 40. Transformation AddMenus 112
Figure 41. Navigation model with added menus derived by transformation AddMenus 112
Figure 42. Metamodel for integration of processes in the navigation model 117
Figure 43. Transformation ProcessIntegration 118
Figure 44. Integrated navigation model for content class ProjectManager derived by transformation ProcessIntegration 119
Figure 45. Manually refined process classes and links for content class ProjectManager 120
Figure 46. Metamodel for the process data modeling 122
Figure 47. Metamodel for the process flow modeling 123
Figure 48. UML control nodes 124
Figure 49. UML object nodes 124
Figure 50. Transformation CreateProcessDataAndFlow 127
Figure 51. Incomplete process flow for web process AddProject derived by rule CreateProcessDataAndFlowForWebProcess 127
Figure 52. Automatically derived process data and flow for simple process RemoveProject derived by rule CreateProcessDataAndFlowForSimpleProcess 130
Figure 53. Automatically derived process data and flow for edit process EditUserProject derived by rule CreateProcessDataAndFlowForEdit 133
Figure 54. Manually refined process flow for process AddProject 136
Figure 55. Manually specified process data for process AddProject 137
Figure 56. Refined content model for process AddProject 137
Figure 57. Manually refined process flow for process RemoveProject 138
Figure 58. Manually refined process data for process RemoveProject 138
Figure 59. Manually refined process data for process EditUserProject 139
Figure 60. Metamodel for presentation modeling (backbone) 141
Figure 61. Metamodel for presentation modeling (user interface elements) 142
Figure 62. Metamodel for presentation modeling (output and static elements) 142
Figure 63. Metamodel for presentation modeling (input elements) 142
Figure 64. Transformation NavigationAndProcess2Presentation 145

Figure 65. Automatically derived presentation classes for the navigation class ProjectManager, menu ProjectManagerMenu and index ProjectIndex 146
Figure 66. Automatically derived presentation classes for the navigation class UserProject and for the menu UserProjectMenu 147
Figure 67. Automatically derived presentation classes for the process classes ProjectKindInput, AddValidationProjectInput and AddUserProjectInput of process AddProject 147
Figure 68. Manually refined presentation classes for the navigation class ProjectManager, menu ProjectManagerMenu and index ProjectIndex 151
Figure 69. Manually refined presentation classes for the navigation class UserProject and for the menu UserProjectMenu 152
Figure 70. Decomposed PIM2PSM transformation 154
Figure 71. Common Web platforms and the proposed generic platform 156
Figure 72. Decomposition of the PIM2PSM transformation for the generic platform 157
Figure 73. Dispatching of a Web request to the DispatcherServlet 159
Figure 74. Handling of a Web request within the Spring Web framework 163
Figure 75. Runtime environment 164
Figure 76. Control flow within the runtime environment 166
Figure 77. Example configuration of the runtime environment 170
Figure 78. XML metamodel 171
Figure 79. Java metamodel 179
Figure 80. Transformation Content2JavaBeans 181
Figure 81. Transformation Content2JavaInterfaces 188
Figure 82. Dynamic navigation structure 191
Figure 83. Transformation Navigation2Conf 193
Figure 84. Runtime process activity 195
Figure 85. Runtime control nodes 196
Figure 86. Runtime object nodes 196
Figure 87. Runtime actions 197
Figure 88. Transformation Process2Conf 201
Figure 89. JSP metamodel 204
Figure 90. Transformation Presentation2JSP 205
Figure 91. Analysis content model 213
Figure 92. Web use cases for content class ProjectManager 214
Figure 93. Web use cases for content class Project 215
Figure 94. Web use cases for content class UserProject 215
Figure 95. Transformation Requirements2Content 216
Figure 96. Content model derived by the transformation Requirements2Content 217
Figure 97. Manually refined content model 218

Figure 98. Transformation RequirementsAndContent2Navigation 220
Figure 99. Navigation space model derived by the transformation RequirementsAndContent2Navigation 221
Figure 100. Manually refined navigation space model 222
Figure 101. Transformation AddIndices 223
Figure 102. Navigation model with added indices derived by the transformation AddIndices 224
Figure 103. Differences between the automatically derived (above) and the manually refined (below) addition of index ProjectIndex 225
Figure 104. Navigation model after manual refining the automatically added indices 226
Figure 105. Transformation AddMenus 227
Figure 106. Navigation model with added menus derived by the transformation AddMenus 227
Figure 107. Transformation ProcessIntegration 228
Figure 108. Automatically derived process classes and links for content class ProjectManager 229
Figure 109. Automatically derived process classes and links for content class Project 229
Figure 110. Automatically derived process classes and links for content class UserProject 229
Figure 111. Manually refined process classes and links for content class ProjectManager 230
Figure 112. Manually refined process classes and links for content class Project 230
Figure 113. Manually refined process classes and links for content class UserProject 231
Figure 114. Transformation CreateProcessDataAndFlow 232
Figure 115. Automatically derived incomplete process flow for web process AddProject 233
Figure 116. Automatically derived process data and flow for simple process RemoveProject 233
Figure 117. Automatically derived process data and flow for simple process AddScenario 234
Figure 118. Automatically derived process flow for simple process StartSimulation 234
Figure 119. Automatically derived process data and flow for edit process EditUserProject 235
Figure 120. Manually refined process flow for process AddProject 237
Figure 121. Manually specified process data for process AddProject 238
Figure 122. Refined content model for process AddProject 238
Figure 123. Manually refined process flow for process RemoveProject 239
Figure 124. Manually refined process data for process RemoveProject 239
Figure 125. Manually refined process data for process EditUserProject 240

Figure 126. Transformation NavigationAndProcess2Presentation 241
Figure 127. Automatically derived presentation classes for the navigation class ProjectManager, the menu ProjectManagerMenu and the index ProjectIndex 242
Figure 128. Automatically derived presentation classes for the process classes ProjectKindInput, AddValidationProjectInput and AddUserProjectInput of process AddProject 242
Figure 129. Automatically derived presentation classes for the process classes RemoveProjectInput and ConfirmRemoveProjectInput of process RemoveProject 242
Figure 130. Automatically derived presentation classes for the navigation class UserProject, the menu UserProjectMenu and the index ScenarioIndex 243
Figure 131. Automatically derived presentation class for the process class EditUserProjectInput of process EditUserProject 243
Figure 132. Differences between the automatically derived (above) and the manually refined (below) presentation classes for the navigation class ProjectManager and the index ProjectIndex 245
Figure 133. Manually refined presentation classes ProjectManager and ProjectIndex 246
Figure 134. Manually refined presentation classes RemoveProjectInput and ConfirmRemoveProjectInput 246
Figure 135. Manually refined presentation classes UserProject and ScenarioIndex 247
Figure 136. Online starting of a simulation run 249
Figure 137. Offline starting of a simulation run 249
Figure 138. Transformation Content2JavaBeans 250
Figure 139. Screenshots for dynamic navigation to sub navigation classes of Project 253
Figure 140. Transformation Navigation2Conf 254
Figure 141. Resulting page after the first part of executing the process RemoveProject 256
Figure 142. Resulting page after the second part of executing the process RemoveProject 257
Figure 143. Token flow when executing process RemoveProject – step 1 258
Figure 144. Token flow when executing process RemoveProject – step 2 259
Figure 145. Token flow when executing process RemoveProject – step 3 259
Figure 146. Token flow when executing process RemoveProject – step 4 260
Figure 147. Token flow when executing process RemoveProject – step 5 260
Figure 148. Token flow when executing process RemoveProject – step 6 261
Figure 149. Token flow when executing process RemoveProject – step 7 261
Figure 150. Token flow when executing process RemoveProject – step 8 262
Figure 151. Transformation Process2Conf 263
Figure 152. Transformation Presentation2JSP 265
Figure 153. Generated JSP (appearance not yet customized) 268
Figure 154. Generated JSP (appearance customized) 269

Figure 155. UML Profile for trace modeling 303
Figure 156. UML Profile for requirements modeling 304
Figure 157. UML Profile for navigation modeling 305
Figure 158. UML Profile for process modeling 306
Figure 159. UML Profile for presentation modeling (backbone) 307
Figure 160. UML Profile for presentation modeling (output and static elements) 307
Figure 161. UML Profile for presentation modeling (input elements) 307

9 REFERENCES

[Abouzahra05] A. Abouzahra, J. Bézivin, M. D. Del Fabro, F. Jouault. A Practical Approach to Bridging Domain Specific Languages with UML profiles. In Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05, San Diego, California, USA. 2005.

[AGG] The Attributed Graph Grammar System, http://tfs.cs.tu-berlin.de/agg/, last visited 20.04.2007.

[Agrawal03] A. Agrawal, G. Karsai and F. Shi. Graph Transformations on Domain-Specific Models. Journal on Software and Systems Modeling, 2003.

[Almeida04] J. P. A. Almeida, R. M. Dijkman, M. van Sinderen, L. F. Pires. On the Notion of Abstract Platform in MDA Development, EDOC 2004:, 2004.

[Amelunxen06] C. Amelunxen, A. Königs, T. Rötschke, A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink, J. Warmer (eds.), Model Driven Architecture - Foundations and Applications: Second European Conference, Heidelberg: Springer Verlag, 2006; Lecture Notes in Computer Science (LNCS), Vol. 4066, Springer Verlag, 361--375.

[Andries96] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph Transformation for Specification and Programming. Technical Report 7/96, Universität Bremen, 1996.

[AndroMDA] http://www.andromda.org/, last visited 20.04.2007.

[ArcStyler] Interactive Objects ArcStyler, http://www.arcstyler.com/, last visited 20.04.2007.

[ArgoUML] ArgoUML open source UML modeling tool, http://argouml.tigris.org/, version 0.22, last visited 03.04.2007.

[Atkinson01] C. Atkinson, T. Kühne. The Essence of Multilevel Metamodeling. In 4th International Conference of the Unified Modeling Language, 2001.

[ATL05a] ATLAS INRIA & LINA research group. ATL Starter's Guide, version 0.1, http://www.eclipse.org/m2m/atl/doc/, December 2005, last visited 20.04.2007.

[ATL05b] ATLAS INRIA & LINA research group. Specification of the ATL Virtual Machine, version 0.1, http://www.eclipse.org/m2m/atl/doc/, 2005, last visited 20.04.2007.

[ATL05c] ATLAS INRIA & LINA research group. ATL Transformation Description Template, version 0.1, http://www.eclipse.org/m2m/atl/doc/, December 2005, last visited 20.04.2007.

[ATL05d] ATLAS INRIA & LINA research group. KM3: Kernel MetaMetaModel, version 0.3, http://www.eclipse.org/m2m/atl/doc/, August 2005, last visited 20.04.2007.

[ATL06a] ATLAS INRIA & LINA research group. ATL User Manual, version 0.7, http://www.eclipse.org/m2m/atl/doc/, February 2006, last visited 20.04.2007.

[Baumeister05] H. Baumeister, A. Knapp, N. Koch, G. Zhang. Modelling Adaptivity with Aspects. In 5th International Conference on Web Engineering (ICWE 2005), Sydney, Australia, David Lowe and Martin Gaedke (Eds.). LNCS 3579, ©Springer Verlag, 406-416, July 2005.

[Baresi02] L. Baresi, F. Garzotto, P. Paolini. Meta-modeling Techniques meets Web Application Design Tools. Proc. of FASE 2002, LNCS 2306, Springer Verlag, pp. 294-307, 2002.

[Baresi05] L. Baresi, L. Mainetti. Beyond Modeling Notations: Consistency and Adaptability of W2000 Models. In Proc. Of SAC'05, ACM Symposium on Applied Computing, Santa Fe, USA, 2005.

[Baresi06] L. Baresi, S. Colazzo, L. Mainetti, and S. Morasca. W2000: A Modeling Notation for Complex Web Applications. In E. Mendes and N. Mosley (eds.) Web Engineering, pages 335-408, Springer, 2006.

[Barth04] M. Barth, R. Hennicker. A. Kraus, M. Ludwig: DANUBIA: An Integrative Simulation System for Global Research in the Upper Danube Basin, Cybernetics and Systems, Vol.7-8, Pages: 639-666, Taylor&Francis, Oct.-Dec. 2004.

[Bast04] W. Bast. Software Factories vs. MDA, http://www.theserverside.net/news/thread.tss?thread_id=30082, posted on 19.11.2004, last visited 20.04.2007.

[Berners-Lee01] T. Berners-Lee, J. Hendler, O. Lassila. The Semantic Web. Scientific American, May 2001.

[Bézivin03] J. Bézivin, G. Dupé, F. Jouault, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In the online proceedings of the OOPSLA'03 Workshop on Generative Techniques in the Context of the MDA, http://www.softmetaware.com/oopsla2003/mda-workshop.html, 2003, last visited 20.04.2007.

[Bézivin05] J. Bézivin. On the Unification Power of Models. Software and System Modeling (SoSym) 4(2):171—188. 2005.

[Börger03] E. Börger, R. Stärk. Abstract State Machines. A method for High-Level System Design and Analysis. Springer-Verlag, 2003.

[Budinsky03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. J. Grose. Eclipse Modeling Framework, Addison-Wesley, 2003.

[Cáceres04] P. Cáceres, E. Marcos, B. Vela. A MDA-based Approach for Web Information Systems, Workshop on Software Model Engineering (WisME 2004), 2004.

[Cachero02] C. Cachero, J. Gómez. Advanced Conceptual Modeling of Web Applications: Embedding Operation Interfaces in Navigation Design, 21st International Conference on Conceptual Modeling, El Escorial, Madrid, November 2002.

[Cachero03] C. Cachero. OO-H: Una extensión a los métodos OO para el modelado y generación automática de interfaces hipermediales, PhD Thesis, http://www.dlsi.ua.es/~ccachero/pTesis.htm, 2003, last visited 20.04.2007.

[Ceri02] S. Ceri, P. Fraternali, M. Brambilla, A. Bongio, S. Comai, M. Matera. Designing Data-Intensive Web Applications. Morgan Kaufmann, 2002.

[Chen76] P. P.-S. Chen. The Entity-Relationship Model - Toward a Unified View of Data. In ACM Transactions on Database Systems 1/1/1976 ACM-Press ISSN 0362-5915, 1976.

[Cleaveland01] C. Cleaveland. Program Generators with XML and Java. Prentice-Hall, 2001.

[Cockburn01] A. Cockburn. Agile Software Development. Addison-Wesley Professional, 2001.

[Cocoon] Apache Cocoon Project, http://cocoon.apache.org/, 2006, last visited 20.04.2007.

[CSS] W3C Cascading Style Sheets, http://www.w3.org/Style/CSS/, 2006, last visited 20.04.2007.

[Czarnecki98] K. Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Ph.D. Thesis, Computer Science Department, Technical University of Ilmenau, Ilmanau, Germany, 1998

[Czarnecki03] K. Czarnecki, S. Helsen. Classification of Model Transformation Approaches. In Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture, Anaheim, California, USA, 2003.

[Díaz04] O. Díaz and J. Rodríguez. Portlets as Web Components: an Introduction. Journal of Universal Computer Science, 10(4):454–472, Apr. 2004.

[Dijkstra76] E. W. Dijkstra: A Discipline of Programming. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Djuric04] D. Djurić, D. Gašević, V. Devedžić, V. Damjanović. A UML profile for OWL ontologies. In Proceedings of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden, 2004.

[Dostal05] W. Dostal, M. Jeckle, I. Melzer, B. Zengler. Service-orientierte Architekturen mit Web Services. Spektrum Verlag, 2005.

[Ehrig05] K. Ehrig, C. Ermel, S. Hänsgen, G. Taentzer. Generation of Visual Editors as Eclipse PlugIns. Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05), 2005.

[Ehrig06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science, Springer, 2006.

[Escalona04a] M. J. Escalona. Modelos y técnicas para la especificación y el análisis de la navegación en sistemas software. PhD Thesis, University of Seville, 2004.

[Escalona04b] M. J. Escalona, N. Koch. Requeriments Engineering for Web Applications: A Comparative Study. Journal of Web Engineering, Rinton Press, Vol. 2, No. 3, February 2004, 192-212.

[Escalona06] M. J. Escalona, N. Koch. Metamodelling the Requirements of Web Systems. In Proc. of 2nd International Conference on Web Information Systems and Technologies, INSTICC, 310-317, Setubal, Portugal, April 2006.

[Filman04] R. E. Filman, T. Elrad, S. Clarke, M. Aksit. Aspect-Oriented Software Development. Addison-Wesley, 2004.

[Finkelstein02] A. Finkelstein, A. Savigni, G. Kappel, W. Retschitzegger, B. Pöll, E. Kimmerstorfer, W. Schwinger, T. Hofer, C. Feichtner. Ubiquitous Web Application Development - A Framework for Understanding, Proc. of SCI2002, 2002.

[Fons03] J. Fons, V. Pelechano, M. Albert, O. Pastor. Development of Web Applications from Web Enhanced Conceptual Schemas. In Workshop on Conceptual Modeling and the Web, ER'03, volume 2813 of Lecture Notes in Computer Science, Springer, 2003.

[Fowler04a] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern, http://martinfowler.com/articles/injection.html, last visited 20.04.2007.

[Fowler04b] M. Fowler. Domains Specific Languages (DSL), http://www.martinfowler.com/bliki/DomainSpecificLanguage.html, last visited 20.04.2007.

[Fowler05a] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?, http://martinfowler.com/articles/languageWorkbench.html, last visited 20.04.2007.

[Gall95] H. Gall, M. Hauswirth, R. Klösch. Objektorientierte Konzepte in Smalltalk, C++, Objective-C, Eiffel und Modula-3, Informatik-Spektrum 18: 195-202. Springer-Verlag, 1995.

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1995.

[Gardner03] T. Gardner, C. Griffin, J. Koehler, and R. Hauser. A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard, 2003.

[Garrett05] J. J. Garrett. Ajax: A New Approach to Web Applications. http://www.adaptivepath.com/publications/essays/archives/000385.php, 2005, last visited 20.04.2007.

[Garzotto93] F. Garzotto, P. Paolini, and D. Schwabe. HDM- A Model-Based Approach to Hypertext Application Design. ACM Transactions on Information Systems, 11(1):1–26, 1993.

[Gerber02] A. Gerber, M. Lawley, K. Raymond, J. Steel, A. Wood. Transformation: The Missing Link of MDA, In A. Corradini, H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.): Graph Transformation: First International Conference (ICGT 2002), Barcelona, Spain,

October 7-12, 2002. Proceedings. LNCS vol. 2505, Springer-Verlag, pp. 90 – 105, 2002.

[GLOWA-Danube] GLOWA-Danube research project, http://www.glowa-danube.de/, last visited 20.04.2007.

[Greenfield04] J. Greenfield, K. Short. Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools, Wiley Publishing, Inc, 2004.

[Harmelen01] M. van Harmelen. Interactive System Design Using Oo&hci Methods, In Object Modeling and User Interface Design, van Harmelen M. (Ed), Addison Wesley, 2001.

[Heckel05] R. Heckel. Graph Transformation in a Nutshell. In Language Engineering for Model-Driven Software Development, Dagstuhl Seminar Proceedings 04101, 2005.

[Hennicker00] R. Hennicker, N. Koch. A UML-based Methodology for Hypermedia Design. In A. Evans, S. Stuart, and B. Selic, editors, UML'2000 - The Unified Modeling Language - Advancing the Standard, LNCS 1939, York, England, ©Springer Verlag, October 2000.

[Hennicker02] R. Hennicker, M. Barth, A. Kraus, M. Ludwig. DANUBIA: A Web-based Modeling and Decision Support System for Integrative Global Change Research in the Upper Danube Basin, in BMBF, German Programme on Globale Change in the Hydrological Cycle (Phase I, 2000 - 2003), Status Report, S.35-38, 2002.

[Hennicker03] R. Hennicker, M. Barth, A. Kraus, M. Ludwig. An Integrated Simulation System for Global Change Research in the Upper Danube Basin First World Congress on Information Technology in Environmental Engineering, ITEE 2003, 2003.

[Hennicker05] R. Hennicker, S. Janisch, A. Kraus, M. Ludwig, W. Mauser, U. Strasser, R. Ludwig: DANUBIA: Design and Implementation of an Integrative Simulation and Decision Support System for the Upper Danube Basin. In Geophysical Research Abstracts (EGU'05), volume 7, 08908 of Abstracts of the European Geosciences Union General Assembly. Vienna, Austria, 2005.

[Hitz05] M. Hitz, G. Kappel, E. Kapsammer, W. Retschitzegger. UML@Work, Dpunkt Verlag, 2005.

[J2EE] The Java EE 5 Tutorial, http://java.sun.com/javaee/5/docs/tutorial/doc/, 2006, last visited 20.04.2007.

[Jacobson99] I. Jacobson, G. Booch, J. Rumbaugh. The Unified Software Development Process, Addison Wesley, 1999.

[Jamda] Jamda: The Java Model Driven Architecture 0.2, http://sourceforge.net/projects/jamda/, May 2003.

[JMI] Java Metadata Interface 1.0, http://java.sun.com/products/jmi, July 2002, last visited 20.04.2007.

[JSP] Sun microsystem. Java Server Pages Technology, http://java.sun.com/products/jsp/index.jsp, last visited 20.04.2007.

[Jouault06a] F. Jouault, I. Kurtev. On the Architectural Alignment of ATL and QVT. In Proceedings of ACM Symposium on Applied Computing (SAC 06), model transformation track, Dijon, Bourgogne, France, 2006.

[Jouault06b] F. Jouault. New announcements for ATL'2006. 2nd AMMA/ATL Workshop on Model Engineering (AWME2), http://www.sciences.univnantes.fr/lina/atl/www/presentations/awme2/02%20-%20New%20announcements%20for%20ATL'2006%20(Jouault).ppt, 2006, last visited 08.12.2006.

[Kappel03a] G. Kappel, B. Pröll, S. Reich, W. Retschizegger. Web Engineering, dpunkt Verlag, 2003.

[Kappel03b] G. Kappel, B. Pröll, W. Retschitzegger, W. Schwinger. Customisation for Ubiquitous Web Applications – A Comparison of Approaches, International Journal of Web Engineering and Technology (IJWET), Inderscience Publishers, 2003.

[Kleppe03] A. Kleppe, J. Warmer, W. Bast. MDA Explained. The Model Driven Architecture: Practise and Promise, Addison-Wesley, 2003.

[Knapp03] A. Knapp, N. Koch, F. Moser, G. Zhang. ArgoUWE: A CASE Tool for Web Applications. In Jolita Ralyté and Colette Roland, editors,

Proc. 1st Int. Wsh. Engineering Methods to Support Information Systems Evolution (EMSISE'03), pages 37-50, Genève, 2003.

[Knapp05] A. Knapp, N. Koch, G. Zhang. Modelling the Behaviour of Web Applications with ArgoUWE. In David Lowe and Martin Gaedke, editors, Proc. 5th Int. Conf. Web Engineering (ICWE'05), volume 3579 of Lect. Notes Comp. Sci., pages 624-626. ©Springer, Berlin, 2005.

[Knapp06] A. Knapp, G. Zhang. Model Transformations for Integrating and Validating Web Application Models. In Heinrich C. Mayr and Ruth Breu, editors, Proc. Modellierung 2006 (MOD'06), volume P-82 of Lect. Notes Informatics, pages 115-128. Gesellschaft für Informatik, 2006.

[Koch01a] N. Koch. Software Engineering for Adaptive Hypermedia Systems: Reference Model, Modeling Techniques and Development Process. PhD Thesis, Ludwig-Maximilians-Universität München, UNI-DRUCK Verlag, 2001.

[Koch01b] N. Koch, A. Kraus, R. Hennicker. The Authoring Process of the UML-based Web Engineering Approach. In Daniel Schwabe, editor, First International Workshop on Web-oriented Software Technology (IWWOST01), June 2001.

[Koch02a] N. Koch, A. Kraus. The Expressive Power of UML-based Web Engineering, Proceedings of the 2nd. International Workshop on Web Oriented Software Technology (IWWOST'2002), Workshop at the ECOOP'2002, Malaga, 2002.

[Koch03a] N. Koch, A. Kraus, C. Cachero and S. Meliá. Modeling Web Business Processes with OO-H and UWE, In Third International Workshop on Web-oriented Software Technology (IWWOST03). D. Schwabe, O. Pastor, G. Rossi, and L. Olsina (Eds.), 27-50, July 2003.

[Koch04a] N. Koch, A. Kraus, C. Cachero and S. Meliá: Integration of Business Processes in Web Applications Models, Journal of Web Engineering, Rinton Press, Vol. 3, No. 1 (2004), 022-049, 2004.

[Koch06a] N. Koch, G. Zhang, M. J. Escalona. Model Transformations from Requirements to Web System Design. In Proc. of 6th International

Conference on Web Engineering (ICWE 2006), Palo Alto, USA, ACM Press, July 2006.

[Koch06b] N. Koch. Transformations Techniques in the Model-Driven Development Process of UWE. In Proc. of 2nd Model-Driven Web Engineering Workshop (MDWE 2006), Palo Alto, USA, ACM Press July 2006.

[Kraus02] A. Kraus, N. Koch: Generation of Web Applications from UML Design Models using an XML Publishing Framework, Proceedings of the Integrated Design and Process Technology Conference, IDPT'2002, Pasadena, 2002.

[Kraus03a] A. Kraus, N. Koch: A Metamodel for UWE, Technical Report 0301, University of Munich, 2003.

[Kraus03b] A. Kraus, N. Koch. Towards a Common Metamodel for the Development of Web Applications, In Third International Conference on Web Engineering (ICWE 2003). J.M. Cueva Lovelle, B.M. Gonzalez Rodriguez, L. Joyanes Aguilar, J.E. Labra Gayo and M.P. Paule Ruiz, editors, LNCS 2722, Springer Verlag, 497-506, July 2003.

[Kurtev06a] I. Kurtev, K. van den Berg, F. Jouault. Rule-based Modularization in Model Transformation Languages illustrated with ATL, In Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, Model transformation (MT 2006), pages 1202—1209, 2006.

[Lara02] J. de Lara, H. Vangheluwe. AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling. In Proc. FASE'02, Springer LNCS 2306, pp. 174 – 188, 2002.

[Lassila99] O. Lassila, R. Swick. Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/, 22 February 1999, last visited 20.04.2007.

[Lieberman01] B. Lieberman. UML Activity Diagrams: Versatile Roadmaps for Understanding System Behavior, Rational Edge Electronic Magazine for the Rational Community, 2001.

[Lima03] F. Lima, D. Schwabe. Application Modeling for the Semantic Web. Proceedings of LA-Web 2003, Santiago, Chile, IEEE Press, pp. 93-103, 2003.

[Ludwig02] R. Ludwig, W. Mauser, S. Niemeyer, A. Colgan, R. Stolz, H. Escher-Vetter, M. Kuhn, M. Reichstein, J. Tenhunen, A. Kraus, M. Ludwig, M. Barth, R. Hennicker. Web-based Modeling of Water, Energy and Matter Fluxes to Support Decision Making in Mesoscale Catchments - the Integrative Perspective of GLOWA-Danube, Physics and Chemistry of the Earth, 2002.

[Ludwig07] M. Ludwig. Modellierung und Architektur eines integrativen Umweltsimulationssystems. PhD Thesis, 2007.

[Lundell06] B. Lundell, B. Lings, A. Persson, A. Mattsson. UML Model Interchange in Heterogeneous Tool Environments: An Analysis of Adoptions of XMI 2, in MoDELS 2006: Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science 4199, Springer-Verlag, 2006.

[M2M] Eclipse Model-to-Model Transformation (M2M) Project, http://www.eclipse.org/m2m/, 2007, last visited 20.04.2007.

[MagicDraw] MagicDraw UML modeling tool, http://www.magicdraw.com/, last visited 20.04.2007.

[Mahmoud04] G. H. Mahmoud. Using and Programming Generics in J2SE 5.0. http://java.sun.com/developer/technicalArticles/J2SE/generics/index.html, 2004, last visited 15.11.2006.

[Markopoulos00] P. Markopoulos. Supporting Interaction Design with UML, Task Modelling, TUPIS' 2000 Workshop at the UML'2000, 2000.

[Markopoulos02] P. Markopoulos. Modelling User Tasks with the Unified Modelling Language, 2002.

[Marschall03] F. Marschall and P. Braun. Model Transformations for the MDA with BOTL. In Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications, University of Twente, Enschede, The Netherlands, CTIT Technical Report TR–CTIT–03–27,

University of Twente, http://trese.cs.utwente.nl/mdafa2003, 2003, last visited 20.04.2007.

[McNeile03] A. McNeile. MDA: The Vision with the Hole? http://www.metamaxim.com/download/documents/MDAv1.pdf, 2003, last visited 20.04.2007.

[Meliá05a] S. Meliá, A. Kraus, N. Koch. MDA Transformations Applied to Web Application Development. In 5th International Conference on Web Engineering (ICWE 2005), Sydney, Australia, David Lowe and Martin Gaedke (Eds.). LNCS 3579, Springer Verlag, 465-471, July 2005.

[Meliá05b] S. Meliá, J. Gomez. Applying Transformations to Model Driven Development of Web applications. 1st International Workshop on Best Practices of UML (ER, 2005) Klagenfurt, Austria, October 2005.

[Meliá06a] S. Meliá, J. Gómez. The WebSA Approach: Applying Model-Driven Engineering To Web Applications, Journal of Web Engineering (JWE), 5(2): 121-149, 2006.

[Meliá06b] S. Meliá, J. Gómez. UPT: A Graphical Transformation Language based on a UML Profile. Proceedings of European Workshop on Milestones, Models and Mappings for Model-Driven Architecture (3M4MDA 2006), 2nd European Conference on Model Driven Architecture (EC-MDA 2006), 2006.

[Mellor02] S. J. Mellor, M. J. Balcer. Executable UML: A Foundation for Model Driven Architecture. Addison-Wesley Professional, 2002.

[Miller03] J. Miller, J. Mukerji. MDA Guide, Object Management Group (OMG), Inc, Version 1.0.1, 2003.

[Moreno05a] N. Moreno, A. Vallecillo. Modeling Interactions between Web Applications and Third Party Systems . In Proc. of the V International Workshop on Web Oriented Software Technologies (IWWOST'05), Porto, Portugal, June 13, 2005.

[Moreno05b] N. Moreno, R. Romero, A. Vallecillo. Incorporating Cooperative Portlets in Web Application Development. In Proc. of the Workshop

on Model-driven Web Engineering (MDWE 2005), pp. 70-79, Sydney, Australia, July 26, 2005.

[Moreno05c] N. Moreno, A. Vallecillo. A Model-based Approach for Integrating Third Party Systems with Web Applications. In Proc. of the International Conference on Web Engineering (ICWE 2005), Sydney, Australia, July 2005. LNCS 3579, 441-452, Springer-Verlag.

[Moreno06] N. Moreno, P. Fraternalli, A. Vallecillo. A UML 2.0 Profile for WebML Modeling, In Proc. of 2nd Model-Driven Web Engineering Workshop (MDWE 2006), ACM, Palo Alto, USA, July 2006, to appear.

[Muller05] P.-A. Muller, P. Studer, F. Fondement, J. Bézivin. Platform independent Web application modeling and development with Netsilon. Software & System Modeling, 4(4), Nov. 2005.

[Muñoz05] J. Muñoz, V. Pelechano. MDA vs Factorías de Software. In Proceedings of DSDM05, Granada, Spain, September 2005.

[NetBeans] NetBeans, http://www.netbeans.org/, last visited 20.04.2007.

[Nunes00] J.N. Nunes, J.F. Cunha. Towards a UML Profile for Interaction Design: The Wisdom approach, Proceedings of the Unified Modeling Language Conference, UML´2000, Evans A. and Kent S. (Eds.). LNCS 1939, Springer Publishing Company, 2000.

[Nunes06] D. A. Nunes, D. Schwabe. Rapid prototyping of web applications combining domain specific languages and model driven design. In Proc. of 6th International Conference on Web Engineering (ICWE 2006), Palo Alto, USA, ACM Press, July 2006.

[NSMDF] Novosoft Metadata Framework and UML library, http://nsuml.sourceforge.net/, last visited 20.04.2007.

[O'Reilly05] T. O'Reilly. What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html, 2005, last visited 20.04.2007.

[OAW] openArchitectureWare platform, http://www.openarchitectureware.org, last visited 10.08.2006.

[OMG02] Object Management Group (OMG). QVT MOF 2.0 Query/Views/Transformations RFP, October 2002.

[OMG05a] Object Management Group (OMG). Unified Modeling Language (UML), Version 2.0, Superstructure: http://www.omg.org/cgi-bin/doc?formal/05-07-04, Infrastructure: http://www.omg.org/cgi-bin/doc?formal/05-07-05, 2005, last visited 20.04.2007.

[OMG05b] Object Management Group (OMG). MOF Query/ Views/ Transformations, Final Adopted Specification, http://www.omg.org/cgi-bin/doc?ptc/2005-11-01, 2005, last visited 20.04.2007.

[OMG05c] Object Management Group (OMG). MOF 2.0 / XMI Mapping Specification, Version 2.1, http://www.omg.org/cgi-bin/doc?formal/2005-09-01, 2005, last visited 20.04.2007.

[OMG06a] Object Management Group (OMG). Meta Object Facility Core Specification, Version 2.0, http://www.omg.org/cgi-bin/doc?formal/2006-01-01, 2006, last visited 20.04.2007.

[OMG06b] Object Management Group (OMG). Object Constraint Language Specification, Version 2.0. http://www.omg.org/cgi-bin/doc?formal/2006-05-01, 2006, last visited 20.04.2007.

[OMG06c] Object Management Group (OMG). Business Process Modeling Notation Specification. http://www.omg.org/docs/dtc/06-02-01.pdf, 2006, last visited 20.04.2007.

[OptimalJ] Compuware OptimalJ. Model-driven development for Java, http://www.compuware.com/products/optimalj/default.htm, last visited 20.04.2007.

[Pastor01] O. Pastor, J. Gomez, E. Insfran, V. Pelechano. The OO-Method Approach for Information Systems Modelling: From Object-Oriented Conceptual Modeling to Automated Programming. Information Systems 26, pp 507–534, 2001.

[Paternò00] F. Paternò. ConcurTaskTrees and UML: how to marry them?, TUPIS'2000 Workshop at the UML'2000, 2000.

[Poseidon] Gentleware Poseidon for UML, http://www.gentleware.com/, last visited 20.04.2007.

[Priese03] L. Priese, H. Wimmel. Theoretische Informatik - Petri Netze, Springer Verlag, 2003

[QVTP03] QVT Partners. Initial Submission for MOF 2.0 Query/View/Transformations RFP, QVT-Partners, http://qvtp.org/, 2003, last visited 20.04.2007.

[Reenskaug79] T. M. H. Reenskaug. Models - Views – Controllers. Xerox PARC, technical note, December 1979.

[Retalis02] S. Retalis, A. Papasalourus, M. Skordalakis. Towards a generic conceptual design meta-model for web-based educational applications. 2nd. International Workshop on Web oriented Software Technology (IWWOST´02), CYTED, 2002.

[RMI] Sun Microsystems. Remote Method Invocation. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp, last visited 20.04.2007.

[Ruscio06] Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, Alfonso Pierantonio. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs, Hyper Article Online, http://hal.ccsd.cnrs.fr, 2006, last visited 20.04.2007.

[Schauerhuber06a] A. Schauerhuber. aspectUWA: Applying Aspect-Orientation to the Model-Driven Development of Ubiquitous Web Applications. Student Extravaganza: Spring School, AOSD'06, Bonn, Germany, March 19, 2006.

[Schauerhuber06b] A. Schauerhuber, M. Wimmer, E. Kapsammer. Bridging existing Web Modeling Languages to Model-Driven Engineering: A Metamodel for WebML, In Proc. of 2nd Model-Driven Web Engineering Workshop (MDWE 2006), ACM, Palo Alto, USA, July 2006.

[Schmidt06] D.C. Schmidt. Model-Driven Engineering, IEEE Computer 39 (2), 2006.

[Schürr89] A. Schürr. Introduction to PROGRES, an Attribute Graph Grammar Based Specification Language. In Proceedings WG'89 Workshop on Graph-Theoretic Concepts in Computer Science, LNCS 411, Springer Verlag, 1989.

[Schwaabe98] D. Schwabe, G. Rossi. An Object Oriented Approach to Web-based Application Design, Theory and Practice of Object Systems 4(4), Wiley and Sons, New York, 1998.

[Soley97] R. Soley et. al. Object Management Architecture Guide, http://doc.omg.org/ab/97-05-05, 1997, last visited 20.04.2007.

[Spring] Spring Framework, http://www.springframework.org/, last visited 20.04.2007.

[Stiegler02] S. Stiegler. Entwicklung eines Generators zur semiautomatischen Erzeugung von Webanwendungen aus UML Design Modellen, Diplomarbeit, Ludwig-Maximilians-Universität München, 2002

[Sun02] Sun Microsystems, Inc. Sun ONE Architecture Guide, http://www.sun.com/software/sunone/docs/arch/, 2002, last visited 20.04.2007.

[Sun06a] Sun Microsystems, Inc. JavaBeans Specification, http://java.sun.com/products/javabeans/docs/spec.html, 2006, last visited 20.04.2007.

[Szyperski02] C. Szyperski. Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 2002

[Tekinerdogan04] B. Tekinerdoğan, S. Bilir, C. Abatlevi. Integrating Platform Selection Rules in the Model Driven Architecture Approach, In Proceedings of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden, 2004.

[Thomas06] D. Thomas, D. Heinemeier Hansson, L. Breedt. Agile Web Development with Rails, Pragmatic Programmers, 2006.

[Torres06] V. Torres, V. Pelechano, P. Giner. Generación de aplicaciones Web basadas en procesos de negocio mediante transformación de modelos, Jornadas de Ingeniería de Software y Base de Datos (JISBD), XI, Barcelona, Spain, 2006.

[UWE] UML-based Web Engineering approach (UWE) homepage, http://www.pst.ifi.lmu.de/projekte/uwe/, last visited 20.04.2007.

[Valderas05] P. Valderas, J. Fons, V. Pelechano. From Web Requirements to Navigational Design – A Transformational Approach. International Conference on Web Engineering (ICWE 2005), LNCS 3579, pp. 506-511, Sydney, 2005.

[VIATRA] VIATRA 2 Model Transformation Framework User's Guide, http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/VIATRA2/index.html, May 2006, last visited 20.04.2007.

[W3C02] W3C Web Services Activity, http://www.w3.org/2002/ws/, 2002, last visited 20.04.2007.

[Wagelaar05] D. Wagelaar, V. Jonckers. Explicit Platform Models for MDA. In Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005). Springer-Verlag, Montego Bay, Jamaica, pages 367—381. 2005.

[Weis04] T. Weis. Model-Driven Development of QoS-Enabled Distributed Applications, PhD Thesis, Technische Universität Berlin, 2004

[Willink03] E. D. Willink. UMLX: A graphical transformation language for MDA. In Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications, University of Twente, Enschede, The Netherlands, CTIT Technical Report TR–CTIT–03–27, University of Twente, http://trese.cs.utwente.nl/mdafa2003, 2003, last visited 20.04.2007.

[XDE] Rational XDE, http://www.rational.com/products/xde, last visited 20.04.2007.

[XSLT] W3C. XSL Transformations (XSLT) Version 1.0, http://www.w3.org/TR/xslt, last visited 20.04.2007.

[Zhang05] G. Zhang, H. Baumeister, N. Koch, A. Knapp. Aspect-Oriented Modeling of Access Control in Web Applications. In Proc. 6th Int. Wsh. Aspect Oriented Modeling (WAOM'05), Chicago, 2005.

A UML PROFILE

This section comprises the UML profile definition for platform independent analysis and design of Web applications. This profile is used as a notation for the metamodel presented in Chapter 4, see 2.2.4. First, a tabular overview of all stereotypes is presented. Then the profile definition is presented separately for each concern of a Web application by using the diagrammatic notation of the UML itself as described in [OMG05a]. Additionally, notation shortcuts are defined, in order to ease the construction of models and to improve the readability of diagrams using the profile notation.

The metamodel was mapped to the profile following these guidelines:

- Each metaclass is mapped to a stereotype. The name of the metaclass is mapped to the name of the stereotype by converting the name of the metaclass to a lower case representation and inserting spaces for each new word (indicated by a change from a lower case to an upper letter with the exception of acronyms) within the name of the metaclass.

- Inheritance between metaclasses is mapped to inheritance between stereotypes

- The base class of a stereotype (extension relationship denoted by a filled arrow head) is derived from the metamodel definition where the base UML metaclass was defined

- An attribute of a metaclass is mapped to an attribute of a stereotype, maintaining the defined multiplicities and ordering. Visibilities are not relevant at the metamodel level.

- The type of an attribute of a metaclass is mapped to the same type for the attribute of the stereotype for primitive types

- Non primitive attributes of a metaclass, i.e. meta association ends, are mapped to strings. The string value refers to the full qualified name of the referenced element as defined in the UML specification. Note that although it is also possible to use

stereotyped dependencies for references between model elements, in most modeling tools it is not possible to "draw" such a dependency between arbitrary types of elements. For instance, in the majority of cases it is not possible to establish a dependency relationship between the ends of an association.

- Derived meta attributes are not mapped to the profile

A.1 Tabular Overview

For each metaclass the corresponding base class from the UML metamodel and the assigned stereotype is listed in the following table:

Metaclass	Baseclass	Stereotype	Remarks
Anchor	**Class**	**«anchor»**	
Edit	**UseCase**	**«edit»**	
EnumerationInput	**Class**	**«enumeration input»**	
GuidedTour	**Class**	**«guided tour»**	
Image	**Class**	**«image»**	
Index	**Class**	**«index»**	
Menu	**Class**	**«menu»**	
Navigation	**Class**	**«navigation»**	
NavigationClass	**Class**	**«navigation class»**	
NavigationLink	**Association**	**«navigation link»**	**Can be omitted**
NavigationProperty	**Property**	**«navigation property»**	**Can be omitted**
PresentationClass	**Class**	**«presentation class »**	
PresentationProperty	**Property**	**«presentation property»**	**Can be omitted**
ProcessActivity	**Activity**	**«process activity»**	

ProcessClass	**Class**	**«process class»**	
ProcessLink	**Association**	**«process link»**	
ProcessProperty	**Property**	**«process property»**	**Can be omitted**
Query	**Class**	**«query»**	
Selection	**Class**	**«selection»**	
SimpleProcess	**UseCase**	**«simple process»**	
StaticImage	**Class**	**«static image»**	
StaticText	**Class**	**«static text»**	
Text	**Class**	**«text»**	
TextInput	**Class**	**«text input»**	
TransformationTrace	**Abstraction**	**«transformation trace»**	
UserAction	**Action**	**«user action»**	
WebProcess	**UseCase**	**«web process»**	

A.2 Trace

The stereotypes for trace modeling are depicted in Figure 155. The filled arrow head denotes the extension relationship between a metaclass and a stereotype.

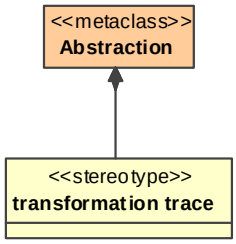


Figure 155. UML Profile for trace modeling

A.3 Requirements

The stereotypes for requirements modeling are depicted in Figure 156.

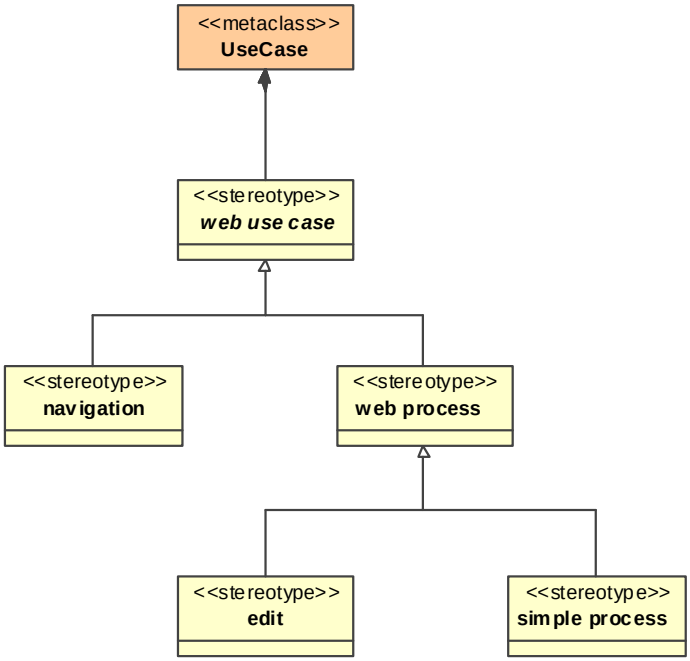


Figure 156. UML Profile for requirements modeling

A.4 Navigation

The stereotypes for navigation modeling are depicted in Figure 157.

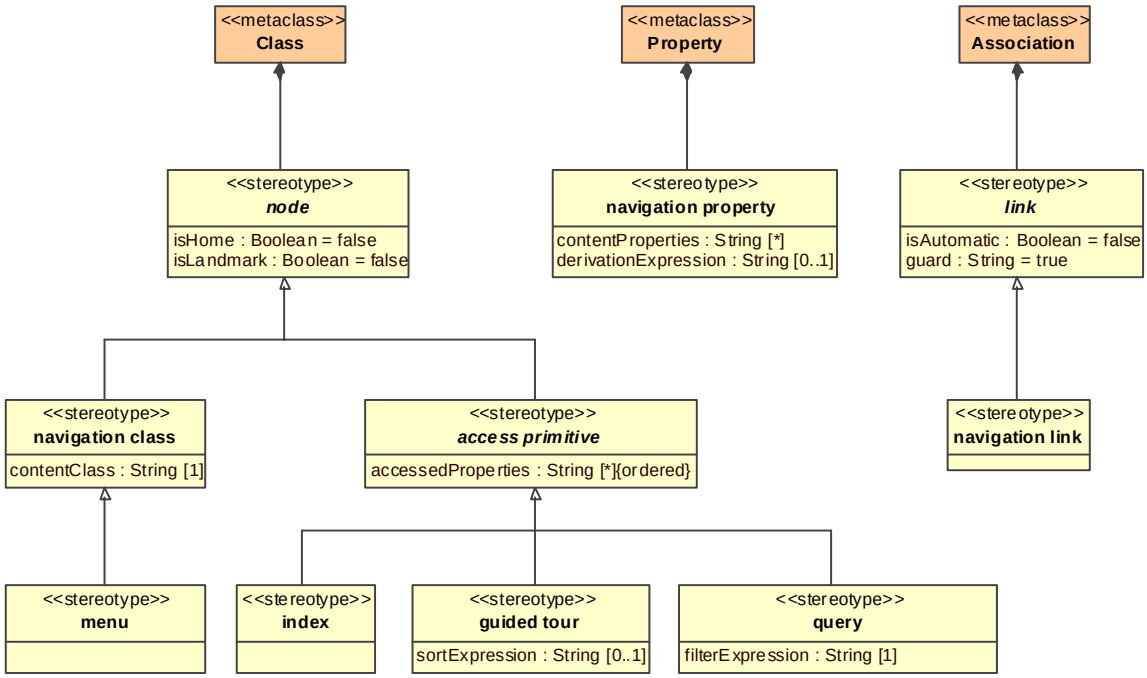


Figure 157. UML Profile for navigation modeling

Notation Shortcuts

- The attribute *contentClass* of the stereotype *navigation class* may be omitted if the content class has the same unique name as the navigation class
- The stereotype *navigation property* may be omitted. Additionally, the attribute *contentProperties* may then be omitted if a property with the same name exists in the corresponding content class.

A.5 Process

The stereotypes for trace modeling are depicted in Figure 158.

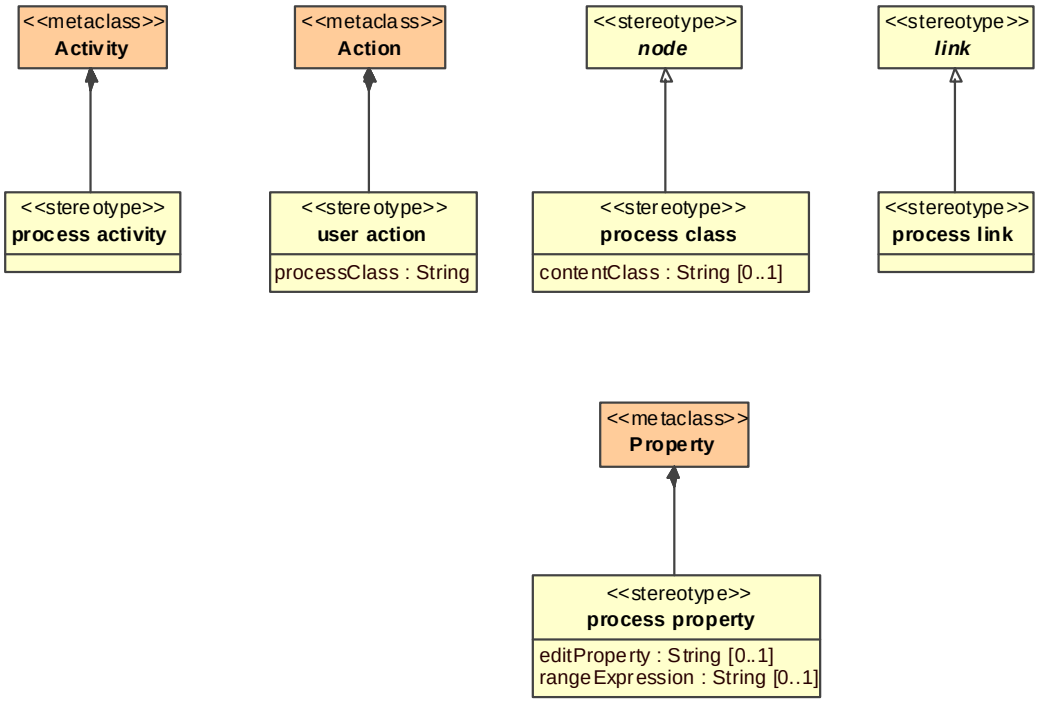


Figure 158. UML Profile for process modeling

Notation Shortcuts

- The attribute *processClass* of the stereotype *user action* may be omitted if a process class with the same name is contained in the main process class associated to the process activity
- The stereotype *process property* may be omitted. Additionally, if the attribute *editProperty* is omitted and a property with the same name exists in the corresponding content class (if any) then this property is assumed to be the edit property.

A.6 Presentation

The stereotypes for trace modeling are depicted in Figure 159 to Figure 161.

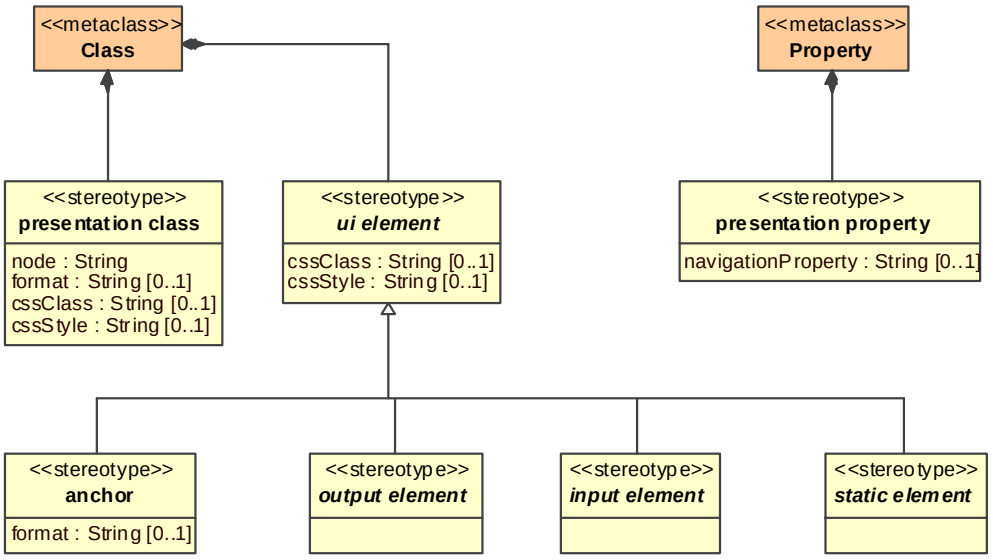


Figure 159. UML Profile for presentation modeling (backbone)

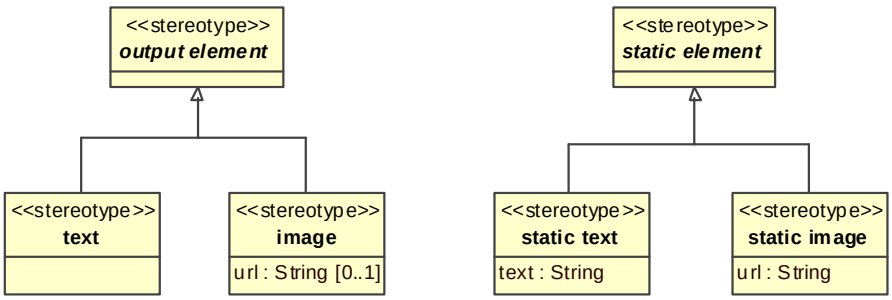


Figure 160. UML Profile for presentation modeling (output and static elements)

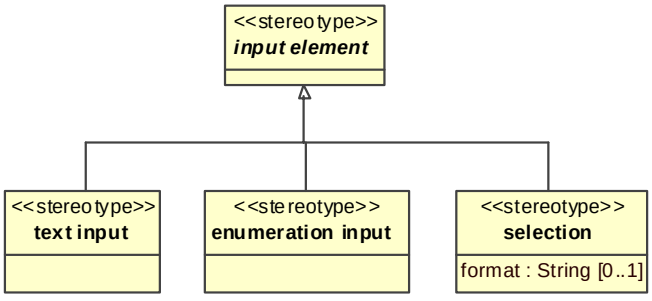


Figure 161. UML Profile for presentation modeling (input elements)

Notation Shortcuts

- The attribute *node* of the stereotype *presentation class* may be omitted if the presentation class has the same unique name as the node

- The stereotype *presentation property* may be omitted. Additionally, if the attribute *navigationProperty* is omitted and a property with the same name exists in the corresponding node (if any) then this property is assumed to be the navigation property.

B ATL TRANSFORMATIONS

In this chapter of the appendix the technical details about the transformations presented in this work are given. First, the details about the technical setup of the transformation environment is presented. Then, all of the used metamodels are presented in the KM3 representation (see 2.3.3.4) as used in the transformation environment, together with the implementation of the constraint checking queries and the serialization queries for the platform specific metamodels. Finally, all of the PIM2PIM and the PIM2PSM transformations presented in this work are listed with all technical details.

B.1 Transformation Environment Setup

This section lists the exact setup of the ATL transformation environment. It is important to stick to the exact versions of all participating software components. The transformation environment is based on Eclipse v3.1 and the following plug-ins have to be installed. Eclipse and all of the plug-in are available at http://www.eclipse.org.

- EMF v2.1
- UML2 v1.1.0

For the ATL environment the following modules have to be checked out from the public CVS repository :pserver:anonymous@dev.eclipse.org:/cvsroot/technology for the date 12.12.2006:

- org.atl.eclipse.adt.builder
- org.atl.eclipse.adt.debug
- org.atl.eclipse.adt.doc.developer
- org.atl.eclipse.adt.doc.user

- org.atl.eclipse.adt.editor
- org.atl.eclipse.adt.perspective
- org.atl.eclipse.adt.wizard
- org.atl.eclipse.engine
- org.atl.eclipse.km3
- org.atl.eclipse.mgm
- org.atl.engine.repositories.emf4atl
- org.atl.engine.repositories.mdr4atl
- org.atl.engine.vm
- org.eclipse.am3.core
- org.eclipse.am3.tools.tge
- org.eclipse.am3.ui
- org.eclipse.am3.zoos.atlantic
- org.eclipse.gmt.atl.atl2006
- org.eclipse.gmt.atl.oclquery.core

Further official installation instructions are available at http://www.eclipse.org/m2m/atl/.

B.2 Metamodels

In the following the "implementation" for the metamodels used in this work is presented. This comprises the UWE metamodel for the platform independent analysis and design described in Chapter 4 as well as the metamodels representing technologies for the platform specific implementation described in Chapter 5. For each metamodel the corresponding KM3 representation is given. The Kernel MetaMetaModel (KM3) [ATL05d] allows the definition of metamodels in an easy Java-like textual notation, and a number of standard

bridges allow the conversion to other metamodel formats. Additionally, the corresponding constraint checking ATL query is presented. Finally, for all platform specific metamodels the corresponding serialization query to code is listed.

B.2.1 UWE Metamodel

This section comprises the "implementation" of the UWE metamodel for the platform independent analysis and design described in Chapter 4.

B.2.1.1 KM3 Metamodel

For the definition of the UWE metamodel in the KM3 format first the UML 2 Ecore metamodel was translated to a KM3 representation. Ecore corresponds directly to a subset of MOF called Essential MOF or EMOF [OMG06a], see also 2.2.2. Then this metamodel was extended in a conservative way with UWE specific constructs. Conservative means that the UML 2 part of the metamodel was not changed. The following KM3 source lists the UWE specific elements. Note that KM3 does not allow to specify initialization values.

```
package UWE
{
    -- UWE SPECIFIC ELEMENTS

    -- TRACE

    class TransformationTrace extends Abstraction
    {
    }

    -- REQUIREMENTS

    abstract class WebUseCase extends UseCase
    {
    }

    class Navigation extends WebUseCase
    {
    }

    class WebProcess extends WebUseCase
    {
    }

    class Edit extends WebProcess
```

```
{
}

class SimpleProcess extends WebProcess
{
}

-- CONTENT

-- NAVIGATION

abstract class NavigationNode extends Class
{
    attribute isHome : Boolean;
    attribute isLandmark : Boolean;
}

class NavigationClass extends NavigationNode
{
    reference contentClass : Class;
}

class NavigationProperty extends Property
{
    reference contentProperties[*] : Property;
    attribute derivationExpression[0-1] : String;
}

abstract class Link extends Association
{
    attribute isAutomatic : Boolean;
    attribute guard : String;
}

class NavigationLink extends Link
{
}

class Menu extends NavigationClass
{
}

abstract class AccessPrimitive extends NavigationNode
{
}
```

```
class Index extends AccessPrimitive
{
}

class GuidedTour extends AccessPrimitive
{
}

class Query extends AccessPrimitive
{
}

-- PROCESS

class ProcessClass extends NavigationNode
{
    reference contentClass[0-1] : Class;
}

class ProcessProperty extends Property
{
    reference editProperty[0-1] : Property;
    attribute rangeExpression[0-1] : String;
}

class ProcessLink extends Link
{
}

class ProcessActivity extends Activity
{
}

class UserAction extends Action
{
    reference processClass : ProcessClass;
}

-- PRESENTATION

class PresentationClass extends Class
{
    attribute format[0-1] : String;
    attribute cssClass : String;
    attribute cssStyle : String;
    reference node : NavigationNode;
```

```
}

class PresentationProperty extends Property
{
    reference navigationProperty : Property;
}

class UIElement extends Class
{
    attribute cssClass : String;
    attribute cssStyle : String;
}

class Anchor extends UIElement
{
    attribute format[0-1] : String;
}

abstract class OutputElement extends UIElement
{
}

class Text extends OutputElement
{
}

class Image extends OutputElement
{
    attribute url[0-1] : String;
}

abstract class InputElement extends UIElement
{
}

class TextInput extends InputElement
{
}

class EnumerationInput extends InputElement
{
}

class Selection extends InputElement
{
    attribute format[0-1] : String;
```

```
    }

    abstract class StaticElement extends UIElement
    {
    }

    class StaticText extends StaticElement
    {
        attribute text : String;
    }

    class StaticImage extends StaticElement
    {
        attribute url : String;
    }

}
```

B.2.1.2 Constraint Checking Query

This section comprises the implementation of the constraint checking query for checking the well-formedness of UWE models as described in 4.1.1. Diagrammatic (or implicit) and explicit constraints are distinguished. Diagrammatic constraints have been defined implicitly in the corresponding diagrams representing the metamodel. On the other hand explicit constraints have been defined using OCL class invariants.

```
query CheckConstraints =

-- REQUIREMENTS

-- diagrammatic constraints

-- explicit constraints

    UWE!WebUseCase.allInstances()->forAll( x | x.assert(
        x.check_WebUseCaseContentClass(), 'WebUseCaseContentClass' ) ) and
    UWE!WebUseCase.allInstances()->forAll( x | x.assert(
        x.check_WebUseCaseTarget(), 'WebUseCaseTarget' ) ) and
    UWE!Navigation.allInstances()->forAll( x | x.assert(
        x.check_NavigationTarget(), 'NavigationTarget' ) ) and
    UWE!Edit.allInstances()->forAll( x | x.assert(
        x.check_EditTarget(), 'EditTarget' ) ) and

-- NAVIGATION
```

```
-- diagrammatic constraints

    UWE!NavigationClass.allInstances()->forAll( x | x.assert(
        x.check_NavigationClassContentClassDefined(),
        'NavigationClassContentClassDefined' ) ) and
    UWE!NavigationLink.allInstances()->forAll( x | x.assert(
        x.check_LinkSource(), 'LinkSource' ) ) and
    UWE!NavigationLink.allInstances()->forAll( x | x.assert(
        x.check_LinkTarget(), 'LinkTarget' ) ) and

-- explicit constraints

    UWE!Namespace.allInstances()->forAll( x | x.assert(
        x.check_NamespaceUniqueHomeNode(), 'NamespaceUniqueHomeNode' ) ) and
    UWE!NavigationNode.allInstances()->forAll( x | x.assert(
        x.check_NodeHomeOrLandmark(), 'NodeHomeOrLandmark' ) ) and
    UWE!NavigationNode.allInstances()->forAll( x | x.assert(
        x.check_NodeReachability(), 'NodeReachability' ) ) and
    UWE!NavigationNode.allInstances()->forAll( x | x.assert(
        x.check_NodeInheritance(), 'NodeInheritance' ) ) and
    UWE!NavigationClass.allInstances()->forAll( x | x.assert(
        x.check_NavigationClassOwnedAttributeType(),
        'NavigationClassOwnedAttributeType' ) ) and
    UWE!NavigationProperty.allInstances()->forAll( x | x.assert(
        x.check_NavigationPropertyType(), 'NavigationPropertyType' ) ) and
    UWE!Link.allInstances()->forAll( x | x.assert(
        x.check_LinkMembers(), 'LinkMembers' ) ) and
    UWE!AccessPrimitive.allInstances()->forAll( x | x.assert(
        x.check_AccessPrimitiveIncoming(), 'AccessPrimitiveIncoming' ) ) and
    UWE!AccessPrimitive.allInstances()->forAll( x | x.assert(
        x.check_AccessPrimitiveOutgoing(), 'AccessPrimitiveOutgoing' ) ) and
    UWE!Index.allInstances()->forAll( x | x.assert(
        x.check_IndexOutgoing(), 'IndexOutgoing' ) ) and
    UWE!GuidedTour.allInstances()->forAll( x | x.assert(
        x.check_GuidedTourOutgoing(), 'GuidedTourOutgoing' ) ) and
    UWE!Query.allInstances()->forAll( x | x.assert(
        x.check_QueryOutgoing(), 'QueryOutgoing' ) ) and

-- PROCESS

-- PROCESS INTEGRATION

-- diagrammatic constraints

-- explicit constraints
```

```
    UWE!ProcessClass.allInstances()->forAll( x | x.assert(
        x.check_ProcessClassLinkTypes(), 'ProcessClassLinkTypes' ) ) and
    UWE!ProcessClass.allInstances()->forAll( x | x.assert(
        x.check_ProcessClassLinkCount(), 'ProcessClassLinkCount' ) ) and
    UWE!ProcessClass.allInstances()->forAll( x | x.assert(
        x.check_ProcessClassWebProcess(), 'ProcessClassWebProcess' ) ) and
    UWE!ProcessLink.allInstances()->forAll( x | x.assert(
        x.check_ProcessLinkEnds(), 'ProcessLinkEnds' ) ) and

-- PROCESS DATA AND FLOW

-- diagrammatic constraints

    UWE!ProcessActivity.allInstances()->forAll( x | x.assert(
        x.check_ProcessActivityContext(), 'ProcessActivityContext' ) ) and
    UWE!UserAction.allInstances()->forAll( x | x.assert(
        x.check_UserActionProcessClassDefined(), 'UserActionProcessClassDefined' ) ) and

-- explicit constraints

    UWE!ProcessProperty.allInstances()->forAll( x | x.assert(
        x.check_ProcessPropertyType(), 'ProcessPropertyType' ) ) and
    UWE!ProcessProperty.allInstances()->forAll( x | x.assert(
        x.check_ProcessPropertyEditProperty(), 'ProcessPropertyEditProperty' ) ) and
    UWE!ProcessActivity.allInstances()->forAll( x | x.assert(
        x.check_ProcessActivityProcessClass(), 'ProcessActivityProcessClass' ) ) and
    UWE!ProcessActivity.allInstances()->forAll( x | x.assert(
        x.check_ProcessActivityParameter(), 'ProcessActivityParameter' ) ) and
    UWE!ProcessActivity.allInstances()->forAll( x | x.assert(
        x.check_ProcessActivityInputParameter(), 'ProcessActivityInputParameter' ) ) and
    UWE!ProcessActivity.allInstances()->forAll( x | x.assert(
        x.check_ProcessActivityOutputParameter(), 'ProcessActivityOutputParameter' ) ) and
    UWE!ProcessActivity.allInstances()->forAll( x | x.assert(
        x.check_ProcessActivityFinalNode(), 'ProcessActivityFinalNode' ) ) and
    UWE!ProcessActivity.allInstances()->forAll( x | x.assert(
        x.check_ProcessActivityInitialNode(), 'ProcessActivityInitialNode' ) ) and
    UWE!UserAction.allInstances()->forAll( x | x.assert(
        x.check_UserActionInput(), 'UserActionInput' ) ) and
    UWE!UserAction.allInstances()->forAll( x | x.assert(
        x.check_UserActionOutput(), 'UserActionOutput' ) ) and

-- PRESENTATION

-- diagrammatic constraints
```

```
    UWE!PresentationClass.allInstances()->forAll( x | x.assert(
        x.check_PresentationClassNodeDefined(), 'PresentationClassNodeDefined' ) ) and
    UWE!PresentationProperty.allInstances()->forAll( x | x.assert(
        x.check_PresentationPropertyContainment(), 'PresentationPropertyContainment' ) ) and

    -- explicit constraints

    UWE!NavigationNode.allInstances()->forAll( x | x.assert(
        x.check_NodePresentationClassDefined(), 'NodePresentationClassDefined' ) ) and
    UWE!PresentationClass.allInstances()->forAll( x | x.assert(
        x.check_PresentationClassInheritance(), 'PresentationClassInheritance' ) ) and
    UWE!PresentationProperty.allInstances()->forAll( x | x.assert(
        x.check_PresentationPropertyType(), 'PresentationPropertyType' ) ) and
    UWE!PresentationProperty.allInstances()->forAll( x | x.assert(
        x.check_PresentationPropertyNavigationProperty(),
        'PresentationPropertyNavigationProperty' ) ) and
    UWE!UIElement.allInstances()->forAll( x | x.assert(
        x.check_UIElementInheritance(), 'UIElementInheritance' ) ) and
    UWE!UIElement.allInstances()->forAll( x | x.assert(
        x.check_UIElementContainment(), 'UIElementContainment' ) )

    ;

uses UWEHelpers;

helper context UWE!NamedElement def : assert( checkResult : Boolean, constraintName : String )
: Boolean =
    if checkResult then true else
        false.debug( self.oclType().toString() + ' ' + self.fullName() + ' Constraint ' + constraintName )
    endif;

-- REQUIREMENTS

-- diagrammatic constraints

-- explicit constraints

helper context UWE!WebUseCase def : check_WebUseCaseContentClass() : Boolean =
    self.subject->select( c | c.oclIsTypeOf( UWE!Class ) )->size() = 1;

helper context UWE!WebUseCase def : check_WebUseCaseTarget() : Boolean =
    UWE!Association.allInstances()->
        select( a | a.endType->size() = 2 and a.endType->includes( self ) )->
        collect( a | a.endType->excluding( self )->first() )->
        select( t | t.oclIsTypeOf( UWE!Class ) )->size() <= 1;
```

```
helper context UWE!Navigation def : check_NavigationTarget() : Boolean =
    let target : UWE!Class = self.target() in
        if target.oclIsUndefined() then false else
            let contentClass : UWE!Class = self.contentClass() in
                if contentClass.oclIsUndefined() then true else -- other constraint violated
                    contentClass.ownedAttribute->exists( p | p.type = target )
                endif
        endif;

helper context UWE!Edit def : check_EditTarget() : Boolean =
    self.target().oclIsUndefined();

-- NAVIGATION

-- diagrammatic constraints

helper context UWE!NavigationClass def : check_NavigationClassContentClassDefined()
    : Boolean =
    not self.contentClass.oclIsUndefined();

helper context UWE!Link def : check_LinkSource() : Boolean = not self.source().oclIsUndefined();

helper context UWE!Link def : check_LinkTarget() : Boolean = not self.target().oclIsUndefined();

-- explicit constraints

helper context UWE!Namespace def : check_NamespaceUniqueHomeNode() : Boolean =
    self.member->select( e | e.oclIsKindOf( UWE!NavigationNode ) )->
    select( n | n.isHome )->size() <= 1;

helper context UWE!NavigationNode def : check_NodeHomeOrLandmark() : Boolean =
    self.isHome or self.isLandmark implies self.oclIsKindOf( UWE!NavigationClass );

helper context UWE!NavigationNode def : check_NodeReachability() : Boolean =
    not ( self.isHome or self.isLandmark )  implies (
        let allNodes : Set( UWE!NavigationNode ) = self.allParents()->including( self ) in
            UWE!NavigationNode.allInstances()->exists( n | n.ownedAttribute->
                exists( p | allNodes->includes( p.type ) ) ) );

helper context UWE!NavigationNode def : check_NodeInheritance() : Boolean =
    if self.oclIsKindOf( UWE!NavigationClass ) then
        self.parents()->forAll( sn | if sn.oclType() = self.oclType() then
            if sn.contentClass.oclIsUndefined() or self.contentClass.oclIsUndefined() then
                true else -- other constraints already fails then
                self.contentClass.conformsTo( sn.contentClass )
            endif
```

```
            else false endif )
    else self.parents()->isEmpty() endif;

helper context UWE!NavigationClass def : check_NavigationClassOwnedAttributeType()
    : Boolean =
    self.ownedAttribute->forAll( p | p.oclIsKindOf( UWE!NavigationProperty ) );

helper context UWE!NavigationProperty def : check_NavigationPropertyType() : Boolean =
    self.type.oclIsKindOf( UWE!DataType ) or self.type.oclIsKindOf( UWE!NavigationNode );

helper context UWE!Link def : check_LinkMembers() : Boolean =
    self.memberEnd->size() = 2 and self.ownedEnd->size() = 1 and
    self.memberEnd->forAll( p | p.type.oclIsKindOf( UWE!NavigationNode ) );

helper context UWE!AccessPrimitive def : check_AccessPrimitiveIncoming() : Boolean =
    let ps : Set( UWE!Property ) = UWE!NavigationNode.allInstances()->
        collect( n | n.ownedAttribute )->flatten()->select( p |
        p.association.oclIsKindOf( UWE!Link ) and p.type = self ) in
            ps->forAll( p | p.lower = 1 and p.upper = 1 );

helper context UWE!AccessPrimitive def : check_AccessPrimitiveOutgoing() : Boolean =
    self.outLinks()->size() = 1;

helper context UWE!Index def : check_IndexOutgoing() : Boolean =
    self.ownedAttribute->forAll( p | p.association.oclIsKindOf( UWE!Link ) implies
        p.isMultivalued() and p.type.oclIsKindOf( UWE!NavigationClass ) );

helper context UWE!GuidedTour def : check_GuidedTourOutgoing() : Boolean =
    self.ownedAttribute->forAll( p | p.association.oclIsKindOf( UWE!Link ) implies
        p.isMultivalued() and p.type.oclIsKindOf( UWE!NavigationClass ) );

helper context UWE!Query def : check_QueryOutgoing() : Boolean =
    self.ownedAttribute->forAll( p | p.association.oclIsKindOf( UWE!Link ) implies
        p.lower = 1 and p.upper = 1 and p.type.oclIsKindOf( UWE!Index ) );

-- PROCESS

-- PROCESS INTEGRATION

-- diagrammatic constraints

-- explicit constraints

helper context UWE!ProcessClass def : check_ProcessClassLinkTypes() : Boolean =
    self.inLinks()->forAll( pl | pl.oclIsTypeOf( UWE!ProcessLink ) ) and
    self.outLinks()->forAll( pl | pl.oclIsTypeOf( UWE!ProcessLink ) );
```

```
helper context UWE!ProcessClass def : check_ProcessClassLinkCount() : Boolean =
    self.inLinks()->size() <= 1 and self.outLinks()->size() <= 1;

helper context UWE!ProcessClass def : check_ProcessClassWebProcess() : Boolean =
    self.inLinks()->notEmpty() implies self.useCase->select( uc |
        uc.oclIsKindOf( UWE!WebProcess ) )->size() = 1;

helper context UWE!ProcessLink def : check_ProcessLinkEnds() : Boolean =
    ( self.source().oclIsKindOf( UWE!NavigationClass ) and
        self.target().oclIsTypeOf( UWE!ProcessClass ) ) or
    ( self.source().oclIsTypeOf( UWE!ProcessClass ) and
        self.target().oclIsKindOf( UWE!NavigationClass ) );

-- PROCESS DATA AND FLOW

-- diagrammatic constraints

helper context UWE!ProcessActivity def : check_ProcessActivityContext() : Boolean =
    if self."context".oclIsKindOf( UWE!ProcessClass ) then
        self."context".ownedBehavior->size() = 1
    else false endif;

helper context UWE!UserAction def : check_UserActionProcessClassDefined() : Boolean =
    not self.processClass.oclIsUndefined();

-- explicit constraints

helper context UWE!ProcessProperty def : check_ProcessPropertyType() : Boolean =
    self.type.oclIsKindOf( UWE!DataType ) or self.type.oclIsTypeOf( UWE!Class );

helper context UWE!ProcessProperty def : check_ProcessPropertyEditProperty() : Boolean =
    not self.editProperty.oclIsUndefined() implies
        if self.processClass().oclIsUndefined() then false else
            if self.processClass().contentClass.oclIsUndefined() then false else
                self.processClass().contentClass.allOwnedAttribute()->includes( self.editProperty )
            endif
        endif;

helper context UWE!ProcessActivity def : check_ProcessActivityProcessClass() : Boolean =
    if self.processClass().oclIsUndefined() then false else self.processClass().inLinks()->notEmpty()
endif;

helper context UWE!ProcessActivity def : check_ProcessActivityParameter() : Boolean =
    self.parameter->select( p | p.direction = #"in" )->size() = 1 and
    self.parameter->select( p | p.direction = #out )->size() <= 1 and
```

```
    self.parameter->select( p | p.direction = #inout )->size() = 0 and
    self.parameter->select( p | p.direction = #return )->size() = 0;

helper context UWE!ProcessActivity def : check_ProcessActivityInputParameter() : Boolean =
    if self.processClass().oclIsUndefined() then false else
        let ls : Set( UWE!Link ) = self.processClass().inLinks() in
            if ls->isEmpty() then false else
                let source : UWE!NavigationNode = ls->asSequence()->first()->source() in
                    if source.oclIsKindOf( UWE!NavigationClass ) then
                        let inputParameter : UWE!Parameter = self.parameter->select( p |
                            p.direction = #"in" )->first() in
                            if inputParameter.oclIsUndefined() then false else
                                source.contentClass.conformsTo( inputParameter.type ) endif
                    else false endif
            endif
    endif;

helper context UWE!ProcessActivity def : check_ProcessActivityOutputParameter() : Boolean =
    if self.processClass().oclIsUndefined() then false else
        if self.processClass().outLinks()->size() = 1 then
            self.parameter->select( p | p.direction = #out )->size() = 1 and
            not self.node->exists( n | n.oclIsKindOf( UWE!ActivityFinalNode ) ) and (
            let target : UWE!NavigationNode = self.processClass().outLinks()->asSequence()->
                first()->target() in
                if target.oclIsKindOf( UWE!NavigationClass ) then
                    let outputParameter : UWE!Parameter = self.parameter->select( p |
                        p.direction = #out )->first() in
                        if outputParameter.oclIsUndefined() then false else
                            outputParameter.type.conformsTo( target.contentClass ) endif
                else false endif )
        else true endif
    endif;

helper context UWE!ProcessActivity def : check_ProcessActivityFinalNode() : Boolean =
    if self.processClass().oclIsUndefined() then false else
        if self.processClass().outLinks()->isEmpty() then
            self.node->exists( n | n.oclIsKindOf( UWE!ActivityFinalNode ) )
        else true endif
    endif;

helper context UWE!ProcessActivity def : check_ProcessActivityInitialNode() : Boolean =
    not self.node->exists( n | n.oclIsKindOf( UWE!InitialNode ) );

helper context UWE!UserAction def : check_UserActionInput() : Boolean =
    if self.processClass.oclIsUndefined() then true else
        -- because then other constraints are violated
```

```
        if self.processClass.contentClass.oclIsUndefined() then
            self.input->isEmpty() and self.incoming->notEmpty()
        else
            self.input->size() = 1 and self.input->forAll( pin |
                if pin.type.oclIsUndefined() then false else
                pin.type.conformsTo( self.processClass.contentClass ) endif )
        endif
    endif;

helper context UWE!UserAction def : check_UserActionOutput() : Boolean =
    if self.processClass.oclIsUndefined() then true else
        -- because then other constraints are violated
        let pps : Sequence( UWE!ProcessProperty ) = self.processClass.processProperties() in
            self.output->forAll( pin | pps->exists( p | if pin.type.oclIsUndefined() or
                p.type.oclIsUndefined() then false else
                p.name = pin.name and p.type.conformsTo( pin.type ) endif ) )
    endif;

-- PRESENTATION

-- diagrammatic constraints

helper context UWE!PresentationClass def : check_PresentationClassNodeDefined() : Boolean =
    not self.node.oclIsUndefined();

helper context UWE!PresentationProperty def : check_PresentationPropertyContainment()
    : Boolean =
    self.class_.oclIsKindOf( UWE!PresentationClass );

-- explicit constraints

helper context UWE!NavigationNode def : check_NodePresentationClassDefined() : Boolean =
    not self.isAbstract and ( self.oclIsKindOf( UWE!ProcessClass ) implies
        self.inLinks()->isEmpty() )
            implies UWE!PresentationClass.allInstances()->select( pc | pc.node = self )->size() = 1;

helper context UWE!PresentationClass def : check_PresentationClassInheritance() : Boolean =
    self.parents()->isEmpty();

helper context UWE!PresentationProperty def : check_PresentationPropertyType() : Boolean =
    self.type.oclIsKindOf( UWE!UIElement ) or self.type.oclIsKindOf( UWE!PresentationClass );

helper context UWE!PresentationProperty def : check_PresentationPropertyNavigationProperty()
    : Boolean =
    if not self.class_.oclIsKindOf( UWE!PresentationClass ) then true else
        -- because then other constraints are violated
```

```
        if self.class_.node.oclIsUndefined() then true else
            -- because then other constraints are violated
            if self.type.oclIsKindOf( UWE!StaticElement ) then
                self.navigationProperty.oclIsUndefined() else
                if self.class_.node.oclIsKindOf( UWE!ProcessClass ) then
                        self.navigationProperty.oclIsKindOf( UWE!ProcessProperty ) else
                    if self.class_.node.oclIsKindOf( UWE!AccessPrimitive ) then
                        self.navigationProperty.oclIsUndefined() else
                        self.navigationProperty.oclIsKindOf( UWE!NavigationProperty )
                    endif
                endif
            endif
        endif
    endif;

helper context UWE!UIElement def : check_UIElementInheritance() : Boolean =
    self.parents()->isEmpty();

helper context UWE!UIElement def : check_UIElementContainment() : Boolean =
    let ps : Set( UWE!PresentationProperty ) =
        UWE!PresentationProperty.allInstances()->select( p | p.type = self ) in
        ps->size() = 1 and ps->forAll( p | p.isComposite and
        p.class_.oclIsKindOf( UWE!PresentationClass ) );
```

B.2.2 Java Metamodel

This section comprises the “implementation” of the metamodel for Java presented in 5.2.

B.2.2.1 KM3 Metamodel

```
package JAVA
{

    abstract class JavaElement
    {
        attribute name : String;
    }

    class Package extends JavaElement
    {
        reference classes[*] container : JavaClass oppositeOf "package";
        reference enumerations[*] container : Enumeration oppositeOf "package";
        attribute isImported : Boolean;
    }
```

```
abstract class ClassMember extends JavaElement
{
    attribute isStatic : Boolean;
    attribute isPublic : Boolean;

    reference owner : JavaClass oppositeOf members;
    reference type[0-1] : Type;
}

class Field extends ClassMember
{
    attribute initializer[0-1] : String;
}

abstract class Type extends JavaElement
{
}

class JavaClass extends Type
{
    attribute isAbstract : Boolean;
    attribute isPublic : Boolean;
    attribute isInterface : Boolean;

    reference superClasses[*] : JavaClass;
    reference actualTypeParameters[*] ordered : JavaClass;
    reference "package" : Package oppositeOf classes;
    reference members[*] container : ClassMember oppositeOf owner;
}

class Method extends ClassMember
{
    attribute body : String;

    reference parameters[*] ordered container : MethodParameter oppositeOf method;
    reference exceptions[*] : JavaClass;
}

class PrimitiveType extends Type
{
}

class Enumeration extends Type
{
    reference "package" : Package oppositeOf enumerations;
```

```
        reference enumerationLiterals[*] ordered container :
            EnumerationLiteral oppositeOf "enumeration";
    }

    class EnumerationLiteral extends JavaElement
    {
        reference "enumeration" : Enumeration oppositeOf enumerationLiterals;
    }

    class MethodParameter extends JavaElement
    {
        reference type : Type;
        reference method : Method oppositeOf parameters;
    }
}

package PrimitiveTypes
{
    datatype String;
    datatype Integer;
    datatype Boolean;
}
```

B.2.2.2 Constraint Checking Query

```
query CheckConstraints_Java =

    JAVA!JavaClass.allInstances()->forAll( x |
        x.assert( x.check_InterfaceMembersAndBody(), 'InterfaceMembersAndBody' ) ) and
    JAVA!JavaClass.allInstances()->forAll( x |
        x.assert( x.check_InterfaceSuperClasses(), 'InterfaceSuperClasses' ) ) and
    JAVA!JavaClass.allInstances()->forAll( x |
        x.assert( x.check_ClassSuperClasses(), 'ClassSuperClasses' ) ) and
    JAVA!Field.allInstances()->forAll( x |
        x.assert( x.check_FieldType(), 'FieldType' ) )
    ;

helper context JAVA!JavaElement def : assert( checkResult : Boolean, constraintName : String ) :
    Boolean =
    if checkResult then true else
        false.debug( self.oclType().toString() + ' ' + self.name + ' Constraint ' + constraintName )
    endif;

helper context JAVA!JavaClass def : check_InterfaceMembersAndBody() : Boolean =
    self.isInterface implies self.members->forAll( m |
```

```
        not m.oclIsTypeOf( JAVA!Field ) and
        m.oclIsTypeOf( JAVA!Method ) implies m.body->isEmpty() );

helper context JAVA!JavaClass def : check_InterfaceSuperClasses() : Boolean =
    self.isInterface implies self.superClasses->forAll( sc | sc.isInterface );

helper context JAVA!JavaClass def : check_ClassSuperClasses() : Boolean =
    not self.isInterface implies self.superClasses->select( sc | not sc.isInterface )->size() <= 1;

helper context JAVA!Field def : check_FieldType() : Boolean =
    self.type->notEmpty();
```

B.2.2.3 Serialization Query

```
query Java2Code = JAVA!Type.allInstances()->
    select( e | if e.oclIsTypeOf( JAVA!JavaClass ) or e.oclIsTypeOf( JAVA!Enumeration ) then
            if e.package.oclIsUndefined() then false else not e.package.isImported endif
        else false endif )->collect(x | x.toString().writeTo(
            'src/' + x.package.name.replaceAll('.', '/') + '/' + x.name + '.java'));

helper context JAVA!ClassMember def: visibility() : String =
    if self.isPublic then
        'public '
    else
        'private '
    endif;

helper context JAVA!JavaClass def: visibility() : String =
    if self.isPublic then
        'public '
    else
        'private '
    endif;

helper context JAVA!ClassMember def: scope() : String =
    if self.isStatic then
        'static '
    else
        ''
    endif;

helper context JAVA!JavaClass def: modifierAbstract() : String =
    if self.isAbstract then
        'abstract '
    else
```

```
        ''
    endif;

helper context JAVA!Type def : fullName() : String = self.name;

helper context JAVA!JavaClass def : fullName() : String =
    if self.package.oclIsUndefined() then self.name else self.package.name + '.' + self.name endif +
    if self.actualTypeParameters->isEmpty() then '' else
        '<' + self.actualTypeParameters->iterate( tp; acc : String = '' | acc +
        if acc = '' then '' else ',' endif + tp.fullName() ) + '>'
    endif;

helper context JAVA!Enumeration def : fullName() : String =
    if self.package.oclIsUndefined() then self.name else self.package.name + '.' + self.name endif;

helper context JAVA!Package def: toString() : String =
    'package ' + self.name + ';\n\n';

helper context JAVA!JavaClass def: toString() : String =
    self.package.toString() + self.visibility() +
    self.modifierAbstract() +
    if self.isInterface then 'interface ' else 'class ' endif + self.name +
    self.superClasses->select( sc | not sc.isInterface or self.isInterface )->
      iterate( sc; acc : String = '' |
        acc +
        if acc = '' then
            ' extends '
        else
            ', '
        endif +
        sc.fullName()
    ) +
    self.superClasses->select( sc | not self.isInterface and sc.isInterface )->
      iterate( sc; acc : String = '' |
        acc +
        if acc = '' then
            ' implements '
        else
            ', '
        endif +
        sc.fullName()
    ) +
    ' {\n' +
    self.members->iterate(i; acc : String = '' |
        acc + i.toString()
    ) +
```

```
    '\n}\n\n';

helper context JAVA!Enumeration def: toString() : String =
    self.package.toString() +
    'public enum ' + self.name +
    ' {\n\t' +
    self.enumerationLiterals->iterate( el; res : String = '' |
        if res = '' then el.name else res + ', ' + el.name endif
    ) +
    ';\n}\n\n';

helper context JAVA!PrimitiveType def: toString() : String =
    if self.name = 'Integer' then
        'int '
    else if self.name = 'Boolean' then
        'boolean '
    else if self.name = 'String' then
        'java.lang.String '
    else if self.name = 'Long' then
        'long '
    else
        'void '
    endif endif endif endif;

helper context JAVA!Field def: toString() : String =
    '\t' + self.visibility() + self.scope() +
    if self.type.oclIsUndefined() then '???' else self.type.fullName() endif
    + ' ' + self.name + if self.initializer.oclIsUndefined() then '' else ' = ' + self.initializer endif + ';\n';

helper context JAVA!Method def: toString() : String =
    '\t' + self.visibility() + self.scope() +
    if self.type.oclIsUndefined() then 'void' else self.type.fullName() endif
    + ' ' + self.name + '(' +
    self.parameters->iterate(i; acc : String = '' |
        acc +
        if acc = '' then
            ''
        else
            ', '
        endif +
        i.toString()
    ) +
    ')' + if self.exceptions->size() > 0 then ' throws ' +
        self.exceptions->iterate( e; acc : String = '' |
            acc + if acc = '' then '' else ', ' endif + e.fullName() )
    else '' endif + if self.body.oclIsUndefined() then ';\n' else ' {\n\t\t' + self.body + '\n\t}\n' endif;
```

```
helper context JAVA!MethodParameter def: toString() : String =
    self.type.fullName()
    + ' ' + self.name;
```

B.2.3 XML Metamodel

This section comprises the “implementation” of the metamodel for XML which is used for generating configuration data for the runtime environment as presented in 5.1.3.

B.2.3.1 KM3 Metamodel

```
package XML
{
    abstract class Node
    {
        attribute name : String;
        attribute value : String;
        reference parent[0-1] : Element oppositeOf children;
    }

    class Attribute extends Node
    {
    }

    class TextNode extends Node
    {
    }

    class Element extends Node
    {
        reference children[*] ordered container : Node oppositeOf parent;
    }

    class Root extends Element
    {
        attribute documentName : String;
    }
}

package PrimitiveTypes
{
    datatype Boolean;
    datatype Integer;
```

```
    datatype String;
}
```

B.2.3.2 Constraint Checking Query

```
query CheckConstraints_XML =

    XML!Node.allInstances()->forAll( x |
        x.assert( x.check_RootParent(), 'RootParent' ) )
    ;

helper context XML!Node def : assert( checkResult : Boolean, constraintName : String ) :
    Boolean =
    if checkResult then true else
        false.debug( self.oclType().toString() + ' ' + self.name + ' Constraint ' + constraintName )
    endif;

helper context XML!Node def : check_RootParent() : Boolean =
    self.parent->isEmpty() implies self.oclIsTypeOf( XML!Root ) and
    self.oclIsKindOf( XML!Root ) implies self.parent->isEmpty();
```

B.2.3.3 Serialization Query

```
query XML2Code = XML!Root.allInstances()->collect( n | n.getChildren()->
    iterate( n; acc : String = '<?xml version=\"1.0\" encoding=\"UTF-8\"?>' + '\n' +
            '<!DOCTYPE beans PUBLIC \"-//SPRING//DTD BEAN//EN\"' +
            '\"http://www.springframework.org/dtd/spring-beans.dtd\">' + '\n\n'
        | acc + n.toCode() ).writeTo( 'conf/' + n.documentName ) );

helper context XML!Element def : getAttribute( name : String ) : String =
    self.children->select( an | an.oclIsKindOf( XML!Attribute ) and an.name = 'name' )->first().value;

helper context XML!Element def : getAttributes() : Sequence(XML!Attribute) =
    self.children->select( cn | cn.oclIsKindOf( XML!Attribute ) );

helper context XML!Element def : getChildren() : Sequence(XML!Node) =
    self.children->select( cn | not cn.oclIsKindOf( XML!Attribute ) );

helper context XML!Element def : toCode() : String =
    '<' + self.name + self.getAttributes()->iterate( n; acc : String = '' | acc + ' ' + n.name + '=\"' +
n.value + '\"' ) + '>'
    + self.getChildren()->iterate( n; acc : String = '' | acc + n.toCode() )
    + '</' + self.name + '>\n';
```

```
helper context XML!TextNode def : toCode() : String =
    self.value;
```

B.2.4 JSP Metamodel

This section comprises the "implementation" of the metamodel for Java Server Pages presented in 5.6.

B.2.4.1 KM3 Metamodel

```
package JSP
{
    abstract class Node
    {
        attribute name : String;
        attribute value : String;
        reference parent[0-1] : Element oppositeOf children;
    }

    class Attribute extends Node
    {
    }

    class TextNode extends Node
    {
    }

    class Element extends Node
    {
        reference children[*] ordered container : Node oppositeOf parent;
    }

    class Root extends Element
    {
        attribute documentName : String;
    }

    abstract class JSPNode extends Node
    {
    }

    class JSPDirective extends JSPNode
    {
```

```
    }
}

package PrimitiveTypes
{
    datatype Boolean;
    datatype Integer;
    datatype String;
}
```

B.2.4.2 Constraint Checking Query

```
query CheckConstraints_JSP =

    JSP!Node.allInstances()->forAll( x |
        x.assert( x.check_RootParent(), 'RootParent' ) )
    ;

helper context JSP!Node def : assert( checkResult : Boolean, constraintName : String ) :
    Boolean =
    if checkResult then true else
        false.debug( self.oclType().toString() + ' ' + self.name + ' Constraint ' + constraintName )
    endif;

helper context JSP!Node def : check_RootParent() : Boolean =
    self.parent->isEmpty() implies self.oclIsTypeOf( JSP!Root ) and
    self.oclIsKindOf( JSP!Root ) implies self.parent->isEmpty();
```

B.2.4.3 Serialization Query

```
query JSP2Code = JSP!Root.allInstances()->collect( n | n.getChildren()->
    iterate( n; acc : String = '' | acc + n.toCode() ).writeTo( ‘jsp/’ + n.documentName ) );

helper context JSP!Element def : getAttributes() : Sequence( JSP!Attribute) =
    self.children->select( cn | cn.oclIsKindOf( JSP!Attribute ) );

helper context JSP!Element def : getChildren() : Sequence( JSP!Node) =
    self.children->select( cn | not cn.oclIsKindOf( JSP!Attribute ) );

helper context JSP!Element def : toCode() : String =
    '<' + self.name + self.getAttributes()->iterate( n; acc : String = '' | acc + ' ' + n.name + '=\"' +
    n.value + '\"' ) + '>\n' + self.getChildren()->iterate( n; acc : String = '' | acc + n.toCode() )
    + '</' + self.name + '>\n';
```

```
helper context JSP!TextNode def : toCode() : String =
    self.value;

helper context JSP!JSPDirective def : toCode() : String =
    '<%@ ' + self.name + ' ' + self.value + ' %>\n';
```

B.3 PIM2PIM Transformations

This section comprises the implementation of the PIM2PIM transformations for the systematic model evolution as presented in Chapter 4. All PIM2PIM transformations are refining transformations as described in 2.3.3.3. At the time of writing the actual version of the ATL compiler, called ATL 2006, did not yet support refining transformations. Therefore, the older version, called ATL 2004, had to be used. Unfortunately, the older version did not support rule inheritance yet, thus some of the transformations had to be written in a more verbose way. For activating the ATL 2004 compiler, the following code line has to be placed at the beginning of a transformation:

```
-- @atlcompiler atl2004
```

The transformation code presented in the two following sections has to be included by all PIM2PIM transformations presented in the rest of this section.

B.3.1 Refinement Header

For refining transformations a trigger rule has to be defined, which either directly or indirectly references all model elements that should implicitly be copied from the source to the target model. Therefore, the following rule *Model2Model* has to be included by all PIM2PIM transformations. It copies all *Model* elements from the source model to the target model and triggers the implicit copying of all directly or indirectly owned members.

```
rule Model2Model
{
    from p : UWE!Model
    to tp : UWE!Model
    (
        name <- p.name,
        owner <- p.owner,
        ownedMember <- p.ownedMember
    )
}
```

B.3.2 Trace Header

The following transformation rules and helpers have to be included by the PIM2PIM transformations, in order to maintain transformation traces as presented in 4.1.2.

```
helper def : tracePackageExists : Boolean = false;

entrypoint rule Init()
{
    do
    {
        thisModule.tracePackageExists <- ( UWE!TransformationTrace.allInstances()->size() > 0 );
    }
}

unique lazy rule CreatePackage
{
    from name : String
    to p : UWE!Package
    (
        name <- name,
        owningPackage <- UWE!Model.allInstances()->asSequence()->first()
    )
}

rule CreateTrace( sourceEl : UWE!NamedElement, targetEl : UWE!NamedElement,
    ruleName : String )
{
    to t : UWE!TransformationTrace
    (
        name <- ruleName,
        supplier <- Set { sourceEl },
        client <- Set { targetEl },
        owningPackage <- if thisModule.tracePackageExists then
                UWE!TransformationTrace.allInstances()->asSequence()->first().owningPackage
            else thisModule.CreatePackage( 'Trace' ) endif
    )
}

helper context UWE!NamedElement def : getTraceSource( ruleName : String )
    : UWE!NamedElement =
    let ts : Set( UWE!NamedElement ) = UWE!TransformationTrace.allInstances()->
        select( t | t.name = ruleName and t.client->includes( self ) )->
        collect( t | t.supplier )->flatten() in
```

```
        if ts->size() > 0 then ts->asSequence()->first() else OclUndefined endif;

helper context UWE!NamedElement def : hasTraceSource( ruleName : String ) : Boolean =
    not self.getTraceSource( ruleName ).oclIsUndefined();

helper context UWE!NamedElement def : getTraceTarget( ruleName : String ) :
    UWE!NamedElement =
    let ts : Set( UWE!NamedElement ) = UWE!TransformationTrace.allInstances()->
        select( t | t.name = ruleName and t.supplier->includes( self ) )->
        collect( t | t.client )->flatten() in
            if ts->size() > 0 then ts->asSequence()->first() else OclUndefined endif;

helper context UWE!NamedElement def : hasTraceTarget( ruleName : String ) : Boolean =
    not self.getTraceTarget( ruleName ).oclIsUndefined();
```

B.3.3 Transformation Requirements2Content

```
module Requirements2Content;
create OUT : UWE refining IN : UWE;

uses UWEHelpers;
uses strings;

-- INCLUDE Refinement Header HERE
-- INCLUDE Trace Header HERE

rule ContentClass2ContentClassWithOperations
{
    from c : UWE!Class ( c.oclIsTypeOf( UWE!Class ) )
    to tc : UWE!Class
    (
        name <- c.name,
        isAbstract <- c.isAbstract,
        owner <- c.owner,
        ownedAttribute <- c.ownedAttribute,
        ownedOperation <- c.ownedOperation->union(
            c.useCase->select( uc | uc.oclIsKindOf( UWE!SimpleProcess ) and
                not uc.hasTraceTarget( 'SimpleProcess2Operation' ) )->collect( uc |
                    thisModule.resolveTemp( uc, 'o' ) ) ),
        ownedBehavior <- c.ownedBehavior,
        nestedClassifier <- c.nestedClassifier,
        useCase <- c.useCase,
        ownedUseCase <- c.ownedUseCase,
        generalization <- c.generalization
    )
```

```
}

rule SimpleProcess2Operation
{
    from sp : UWE!SimpleProcess ( not sp.hasTraceTarget( 'SimpleProcess2Operation' ) )
    using
    {
        target : UWE!Class = sp.target();
    }
    to tsp : UWE!SimpleProcess
    (
        name <- sp.name,
        subject <- sp.subject,
        include <- sp.include,
        extend <- sp.extend,
        extensionPoint <- sp.extensionPoint,
        ownedBehavior <- sp.ownedBehavior,
        attribute <- sp.attribute,
        ownedBehavior <- sp.ownedBehavior
    ),
    o : UWE!Operation
    (
        name <- sp.name.regexReplaceAll( ' ', '' ).firstToLower(),
        type <- target,
        ownedParameter <- if target.oclIsUndefined() then Sequence {} else
            Sequence { thisModule.Type2ReturnParameter( target ) } endif
    )
    do
    {
        thisModule.CreateTrace( sp, o, 'SimpleProcess2Operation' );
    }
}

lazy rule Type2ReturnParameter
{
    from t : UWE!Type
    to p : UWE!Parameter
    (
        type <- t,
        direction <- #return
    )
}
```

B.3.4 Transformation RequirementsAndContent2Navigation

```
module RequirementsAndContent2Navigation;
create OUT : UWE refining IN : UWE;

uses UWEHelpers;
uses strings;

-- INCLUDE Refinement Header HERE
-- INCLUDE Trace Header HERE

helper context UWE!Class def : isContentClass() : Boolean =
    self.oclIsTypeOf( UWE!Class );

helper context UWE!Package def : isContentPackage() : Boolean =
    self.ownedMember->select( el | el.oclIsKindOf( UWE!Class ) )->exists( c |
        c.isContentClass() ) or
    self.ownedMember->select( el | el.oclIsKindOf( UWE!Package ) )->exists( p |
        p.isContentPackage() );

helper context UWE!Element def : isRelevantForNavigation() : Boolean =
    UWE!WebUseCase.allInstances()->exists( uc | uc.contentClass() = self or uc.target() = self );

-- rule for copying package structure from content to navigation, not outlined in 4.4.2.1

rule ContentPackage2NavigationPackage
{
    from p : UWE!Package ( p.oclIsTypeOf( UWE!Package ) and p.isContentPackage() and
        not p.hasTraceTarget( 'ContentPackage2NavigationPackage' ) )
    using
    {
        owningPackage : UWE!Package = let traceP : UWE!Package =
            p.owningPackage.getTraceTarget( 'ContentPackage2NavigationPackage' ) in
            if traceP.oclIsUndefined() then
                if p.owningPackage.oclIsTypeOf( UWE!Model ) then p.owningPackage
                else thisModule.resolveTemp( p.owningPackage, 'np' ) endif
            else traceP endif;
    }
    to tp : UWE!Package
    (
        name <- p.name,
        owner <- p.owner,
        ownedMember <- p.ownedMember
    ),
    np : UWE!Package
    (
        name <- if p.owningPackage.oclIsTypeOf( UWE!Model ) then 'Navigation' else p.name endif,
        owner <- owningPackage,
```

```
        owningPackage <- owningPackage
    )
    do
    {
        thisModule.CreateTrace( p, np, 'ContentPackage2NavigationPackage' );
    }
}

rule ContentClass2NavigationClass
{
    from c : UWE!Class ( c.isRelevantForNavigation() and
        not c.hasTraceTarget( 'ContentClass2NavigationClass' ) )
    using
    {
        owningPackage : UWE!Package =
            if c.owningPackage.oclIsUndefined() then OclUndefined else
                let traceP : UWE!Package = c.owningPackage.getTraceTarget(
                    'ContentPackage2NavigationPackage' ) in
                    if traceP.oclIsUndefined() then thisModule.resolveTemp( c.owningPackage, 'np' )
                    else traceP endif
            endif;
    }
    to tc : UWE!Class
    (
        name <- c.name,
        isAbstract <- c.isAbstract,
        owner <- c.owner,
        ownedAttribute <- c.ownedAttribute,
        ownedOperation <- c.ownedOperation,
        ownedBehavior <- c.ownedBehavior,
        nestedClassifier <- c.nestedClassifier,
        useCase <- c.useCase,
        ownedUseCase <- c.ownedUseCase,
        generalization <- c.generalization
    ),
    nc : UWE!NavigationClass
    (
        name <- c.name,
        isAbstract <- c.isAbstract,
        contentClass <- tc,
        owner <- owningPackage,
        owningPackage <- owningPackage,
        generalization <- ng,
        ownedAttribute <- c.ownedAttribute->select( p | p.type.oclIsKindOf( UWE!DataType )
            or ( p.type.oclIsTypeOf( UWE!Class ) and not p.association.oclIsUndefined() and
            c.useCase->exists( uc | uc.oclIsKindOf( UWE!Navigation ) and uc.target() = p.type ) ) )->
```

```
                collect( p | thisModule.resolveTemp( p, 'np' ) )
    ),
    ng : distinct UWE!Generalization foreach ( g in c.generalization )
    (
        general <- let traceC : UWE!Class = g.general.getTraceTarget(
            'ContentClass2NavigationClass' ) in
                if traceC.oclIsUndefined() then thisModule.resolveTemp( g.general, 'nc' )
                else traceC endif
    )
    do
    {
        thisModule.CreateTrace( c, nc, 'ContentClass2NavigationClass' );
    }
}

rule Property2NavigationProperty
{
    from p : UWE!Property ( if p.class_.oclIsUndefined() or p.type.oclIsUndefined() then false else
            p.oclIsTypeOf( UWE!Property ) and
            p.class_.isRelevantForNavigation() and p.association.oclIsUndefined() and
            p.type.oclIsKindOf( UWE!DataType ) and
            not p.hasTraceTarget( 'Property2NavigationProperty' )
        endif )
    using
    {
        nc : UWE!NavigationClass = let traceC : UWE!NavigationClass =
            p.class_.getTraceTarget( 'ContentClass2NavigationClass' ) in
            if traceC.oclIsUndefined() then thisModule.resolveTemp( p.class_, 'nc' ) else traceC endif;
    }
    to tp : UWE!Property
    (
        name <- p.name,
        owner <- p.owner,
        class_ <- p.class_,
        type <- p.type,
        aggregation <- p.aggregation,
        upper <- p.upper,
        lower <- p.lower,
        isOrdered <- p.isOrdered,
        isUnique <- p.isUnique,
        isStatic <- p.isStatic,
        isComposite <- p.isComposite,
        isDerived <- p.isDerived,
        isReadOnly <- p.isReadOnly
    ),
    np : UWE!NavigationProperty
```

```
    (
        name <- p.name,
        owner <- nc,
        class_ <- nc,
        type <- p.type,
        aggregation <- p.aggregation,
        upper <- p.upper,
        lower <- p.lower,
        isOrdered <- p.isOrdered,
        isUnique <- p.isUnique,
        isStatic <- p.isStatic,
        isComposite <- p.isComposite,
        contentProperties <- Sequence { p }
    )
    do
    {
        thisModule.CreateTrace( p, np, 'Property2NavigationProperty' );
    }
}

rule AssociationProperty2NavigationLink
{
    from p : UWE!Property ( if p.class_.oclIsUndefined() or p.type.oclIsUndefined() then false else
            if p.class_.oclIsTypeOf( UWE!Class ) then
                p.oclIsTypeOf( UWE!Property ) and
                not p.association.oclIsUndefined() and
                p.class_.useCase->exists( uc |
                    uc.oclIsKindOf( UWE!Navigation ) and uc.target() = p.type )
            else false endif and
            not p.hasTraceTarget( 'AssociationProperty2NavigationLink' )
        endif )
    using
    {
        source : UWE!NavigationClass = let traceC : UWE!NavigationClass =
            p.class_.getTraceTarget( 'ContentClass2NavigationClass' ) in
            if traceC.oclIsUndefined() then thisModule.resolveTemp( p.class_, 'nc' ) else traceC endif;
        target : UWE!NavigationClass = let traceC : UWE!NavigationClass =
            p.type.getTraceTarget( 'ContentClass2NavigationClass' ) in
            if traceC.oclIsUndefined() then thisModule.resolveTemp( p.type, 'nc' ) else traceC endif;
        owningPackage : UWE!Package = if p.class_.owningPackage.oclIsUndefined() then
                OclUndefined else
                let traceP : UWE!Package = p.class_.owningPackage.getTraceTarget(
                    'ContentPackage2NavigationPackage' ) in
                    if traceP.oclIsUndefined() then thisModule.resolveTemp(
                        p.class_.owningPackage, 'np' ) else traceP endif
            endif;
```

```
}
to tp : UWE!Property
(
    name <- p.name,
    owner <- p.owner,
    class_ <- p.class_,
    type <- p.type,
    aggregation <- p.aggregation,
    upper <- p.upper,
    lower <- p.lower,
    isOrdered <- p.isOrdered,
    isUnique <- p.isUnique,
    isStatic <- p.isStatic,
    isComposite <- p.isComposite,
    isDerived <- p.isDerived,
    isReadOnly <- p.isReadOnly
),
nl : UWE!NavigationLink
(
    name <- source.name + ' -> ' + target.name,
    owner <- owningPackage,
    owningPackage <- owningPackage
),
nps : UWE!Property
(
    association <- nl,
    owner <- nl,
    owningAssociation <- nl,
    type <- source
),
np : UWE!NavigationProperty
(
    name <- p.name,
    owner <- source,
    class_ <- source,
    type <- target,
    association <- nl,
    contentProperties <- Set { p },
    aggregation <- p.aggregation,
    upper <- p.upper,
    lower <- p.lower,
    isOrdered <- p.isOrdered,
    isUnique <- p.isUnique,
    isStatic <- p.isStatic,
    isComposite <- p.isComposite
)
```

```
    do
    {
        thisModule.CreateTrace( p, nl, 'AssociationProperty2NavigationLink' );
    }
}
```

B.3.5 Transformation AddIndices

```
module AddIndices;
create OUT : UWE refining IN : UWE;

uses UWEHelpers;
uses strings;

-- INCLUDE Refinement Header HERE
-- INCLUDE Trace Header HERE

rule NavigationProperty2Index
{
    from np : UWE!NavigationProperty ( np.isMultivalued() and
        np.association.oclIsKindOf( UWE!Link ) and
        np.class_.oclIsKindOf( UWE!NavigationClass ) and
        np.type.oclIsKindOf( UWE!NavigationClass ) and
        not np.hasTraceTarget( 'AddIndices' ) )
    to tnp : UWE!NavigationProperty
    (
        name <- np.name,
        owner <- np.owner,
        class_ <- np.class_,
        type <- index,
        association <- np.association,
        aggregation <- np.aggregation,
        upper <- 1,
        lower <- 1,
        isOrdered <- np.isOrdered,
        isUnique <- np.isUnique,
        isStatic <- np.isStatic,
        isComposite <- np.isComposite,
        derivationExpression <- np.derivationExpression,
        contentProperties <- np.contentProperties
    ),
    index : UWE!Index
    (
        name <- if UWE!Property.allInstances()->select( p | p.isMultivalued() and
                p.association.oclIsKindOf( UWE!Link ) and p.type = np.type )->size() > 1 then
```

```
                np.class_.name else '' endif + np.type.name + 'Index',
        owner <- np.class_.owner,
        package <- np.class_.package,
        owningPackage <- np.class_.owningPackage,
        ownedAttribute <- Sequence { npt }
    ),
    nl : UWE!NavigationLink
    (
        owner <- np.class_.owner,
        package <- np.class_.package,
        owningPackage <- np.class_.owningPackage
    ),
    nps : UWE!Property
    (
        owner <- nl,
        association <- nl,
        owningAssociation <- nl,
        type <- index,
        lower <- 1,
        upper <- 1
    ),
    npt : UWE!Property
    (
        owner <- index,
        association <- nl,
        class_ <- index,
        type <- np.type,
        aggregation <- np.aggregation,
        isComposite <- np.isComposite,
        lower <- 0,
        upper <- 0-1 -- ATL bug: -1 for unlimited gives error
    )
    do
    {
        thisModule.CreateTrace( np, index, 'NavigationProperty2Index' );
    }
}
```

B.3.6 Transformation AddMenus

```
module AddMenus;
create OUT : UWE refining IN : UWE;

uses UWEHelpers;
uses strings;
```

```
-- INCLUDE Refinement Header HERE
-- INCLUDE Trace Header HERE

rule NavigationClass2NavigationClassWithMenu
{
    from nc : UWE!NavigationClass ( nc.oclIsTypeOf( UWE!NavigationClass ) and
        ( nc.ownedAttribute->select( p | not p.isComposite and
            p.association.oclIsKindOf( UWE!Link ) and
            p.type.oclIsKindOf( UWE!NavigationNode ) and
            not p.type.oclIsTypeOf( UWE!Menu ) )->size() > 0
        or nc.contentClass.useCase->exists( uc | uc.oclIsKindOf( UWE!WebProcess ) ) ) and
        not nc.hasTraceTarget( 'NavigationClass2Menu' ) )
    using
    {
        menuNps : Sequence( UWE!Property ) = nc.ownedAttribute->select( p |
            not p.isComposite and p.association.oclIsKindOf( UWE!Link ) and
            p.type.oclIsKindOf( UWE!NavigationNode ) and not p.type.oclIsTypeOf( UWE!Menu ) );
        otherNps : Sequence( UWE!Property ) =
            ( nc.ownedAttribute->asSet() - menuNps->asSet() )->asSequence();
    }
    to tnc : UWE!NavigationClass
    (
        name <- nc.name,
        isAbstract <- nc.isAbstract,
        owner <- nc.owner,
        feature <- otherNps->including( apt )->union( nc.ownedOperation ),
        ownedAttribute <- otherNps->including( apt ),
        ownedOperation <- nc.ownedOperation,
        ownedBehavior <- nc.ownedBehavior,
        nestedClassifier <- nc.nestedClassifier,
        package <- nc.package,
        generalization <- nc.generalization,
        isHome <- nc.isHome,
        isLandmark <- nc.isLandmark,
        useCase <- nc.useCase,
        contentClass <- nc.contentClass
    ),
    menu : UWE!Menu
    (
        name <- nc.name + 'Menu',
        isAbstract <- nc.isAbstract,
        owner <- nc.owner,
        package <- nc.package,
        owningPackage <- nc.owningPackage,
        feature <- menuNps,
```

```
        ownedAttribute <- menuNps,
        package <- nc.package,
        generalization <- mg,
        contentClass <- nc.contentClass
    ),
    a : UWE!Association
    (
        owner <- nc.owner,
        package <- nc.package,
        owningPackage <- nc.owningPackage
    ),
    aps : UWE!Property
    (
        owner <- a,
        association <- a,
        owningAssociation <- a,
        type <- nc,
        lower <- 1,
        upper <- 1
    ),
    apt : UWE!NavigationProperty
    (
        owner <- nc,
        association <- a,
        class_ <- nc,
        type <- menu,
        aggregation <- #composite,
        isComposite <- true,
        lower <- 1,
        upper <- 1
    ),
    mg : distinct UWE!Generalization foreach ( g in nc.generalization )
    (
        general <- thisModule.resolveTemp( g.general, 'menu' ),
        specific <- menu
    )
    do
    {
        thisModule.CreateTrace( nc, menu, 'NavigationClass2Menu' );
    }
}

rule NavigationProperty2MenuProperty
{
    from np : UWE!NavigationProperty ( np.class_.oclIsTypeOf( UWE!NavigationClass ) and
        np.type.oclIsKindOf( UWE!NavigationNode ) and not np.type.oclIsTypeOf( UWE!Menu ) and
```

```
        not np.isComposite and np.association.oclIsKindOf( UWE!Link ) and
        not np.hasTraceTarget( 'NavigationProperty2MenuProperty' ) )
    to tnp : UWE!NavigationProperty
    (
        name <- np.name,
        owner <- thisModule.resolveTemp( np.class_, 'menu' ),
        class_ <- thisModule.resolveTemp( np.class_, 'menu' ),
        type <- np.type,
        association <- np.association,
        aggregation <- np.aggregation,
        upper <- np.upper,
        lower <- np.lower,
        isOrdered <- np.isOrdered,
        isUnique <- np.isUnique,
        isStatic <- np.isStatic,
        isComposite <- np.isComposite,
        derivationExpression <- np.derivationExpression,
        contentProperties <- np.contentProperties
    )
    do
    {
        thisModule.CreateTrace( np, tnp, 'NavigationProperty2MenuProperty' );
    }
}
```

B.3.7 Transformation ProcessIntegration

```
module ProcessIntegration;
create OUT : UWE refining IN : UWE;

uses UWEHelpers;
uses strings;

-- INCLUDE Refinement Header HERE
-- INCLUDE Trace Header HERE

rule Menu2IntegratedMenu
{
    from nc : UWE!Menu (
        nc.contentClass.useCase->exists( uc | uc.oclIsKindOf( UWE!WebProcess )
        and nc.hasTraceSource( 'NavigationClass2Menu' ) )
    )
    using
    {
        wps : Sequence( UWE!WebProcess ) = nc.contentClass.useCase->select( uc |
```

```
        uc.oclIsKindOf( UWE!WebProcess ) )->
        select( wp | not wp.subject->exists( c |
        c.oclIsTypeOf( UWE!ProcessClass ) ) )->asSequence();
    wpsWithTarget : Sequence( UWE!WebProcess ) = wps->select( wp |
        not wp.target().oclIsUndefined() );
    wpsWithoutTarget : Sequence( UWE!WebProcess ) = wps->select( wp |
        wp.target().oclIsUndefined() );
    wpsOrdered : Sequence( UWE!WebProcess ) = wpsWithTarget->union( wpsWithoutTarget );
}
to tnc : UWE!Menu
(
    name <- nc.name,
    owner <- nc.owner,
    feature <- nc.feature->including( npt ),
    ownedAttribute <- nc.ownedAttribute->including( npt ),
    ownedOperation <- nc.ownedOperation,
    ownedBehavior <- nc.ownedBehavior,
    nestedClassifier <- nc.nestedClassifier,
    package <- nc.package,
    generalization <- nc.generalization,
    isHome <- nc.isHome,
    isLandmark <- nc.isLandmark,
    useCase <- nc.useCase,
    contentClass <- nc.contentClass
),
pc : distinct UWE!ProcessClass foreach ( wp in wpsOrdered )
(
    name <- let n : String = wp.name.regexReplaceAll( ' ', '' ).firstToUpper() in
        if UWE!WebProcess.allInstances()->select( uc |
            uc.name.regexReplaceAll( ' ', '' ).firstToUpper() = n )->size() > 2 then
                nc.getTraceSource( 'NavigationClass2Menu' ).name else '' endif + n,
    owner <- nc.owner,
    package <- nc.package,
    owningPackage <- nc.owningPackage,
    useCase <- wp,
    ownedAttribute <- wpsOrdered->iterate( wp; res : Sequence(OclAny) = Sequence{} | res->
        including( if wpsWithTarget->includes( wp ) then
            enpt->at( wpsWithTarget->indexOf( wp ) ) else Sequence {} endif ) )
),
pl : distinct UWE!ProcessLink foreach ( wp in wpsOrdered )
(
    name <- wp.name.regexReplaceAll( ' ', '' ).firstToUpper() + 'Entry',
    owner <- nc.owner,
    package <- nc.package,
    owningPackage <- nc.owningPackage
),
```

```
    nps : distinct UWE!Property foreach ( wp in wpsOrdered )
    (
        owner <- pl,
        association <- pl,
        owningAssociation <- pl,
        type <- tnc,
        lower <- 1,
        upper <- 1
    ),
    npt : distinct UWE!NavigationProperty foreach ( wp in wpsOrdered )
    (
        owner <- tnc,
        association <- pl,
        class_ <- tnc,
        type <- pc,
        lower <- 1,
        upper <- 1
    ),

    -- exit link
    epl : distinct UWE!ProcessLink foreach ( wp in wpsWithTarget )
    (
        name <- wp.name.regexReplaceAll( ' ', '' ).firstToUpper() + 'Exit',
        owner <- nc.owner,
        package <- nc.package,
        owningPackage <- nc.owningPackage
    ),
    enps : distinct UWE!Property foreach ( wp in wpsWithTarget )
    (
        owner <- epl,
        association <- epl,
        owningAssociation <- epl,
        type <- pc,
        lower <- 1,
        upper <- 1
    ),
    enpt : distinct UWE!Property foreach ( wp in wpsWithTarget )
    (
        owner <- pc,
        association <- epl,
        class_ <- pc,
        type <- wp.target().getTraceTarget( 'ContentClass2NavigationClass' ),
        lower <- 1,
        upper <- 1
    )
}
```

B.3.8 Transformation CreateProcessDataAndFlow

```
module CreateProcessDataAndFlow;
create OUT : UWE refining IN : UWE;

uses UWEHelpers;
uses strings;

-- INCLUDE Refinement Header HERE
-- INCLUDE Trace Header HERE

rule CreateProcessDataAndFlowForWebProcess
{
    from pc : UWE!ProcessClass ( pc.ownedBehavior->isEmpty() and
        pc.webProcess().oclIsTypeOf( UWE!WebProcess ) )
    using
    {
        source : UWE!NavigationClass = let ls : Set( UWE!Link ) = pc.inLinks() in
            if ls->isEmpty() then OclUndefined else ls->asSequence()->first().source() endif;
        target : UWE!NavigationClass = let ls : Set( UWE!Link ) = pc.outLinks() in
            if ls->isEmpty() then OclUndefined else ls->asSequence()->first().target() endif;
        targetSeq : Sequence( Boolean ) = if target.oclIsUndefined() then Sequence {} else
            Sequence { true } endif;
        nTargetSeq : Sequence( Boolean ) = if target.oclIsUndefined() then Sequence { true } else
            Sequence {} endif;
    }
    to tpc : UWE!ProcessClass
    (
        name <- pc.name,
        isHome <- pc.isHome,
        isLandmark <- pc.isLandmark,
        owner <- pc.owner,
        feature <- pc.feature,
        ownedAttribute <- pc.ownedAttribute,
        ownedOperation <- pc.ownedOperation,
        ownedBehavior <- Sequence { pa },
        nestedClassifier <- pc.nestedClassifier,
        package <- pc.package,
        generalization <- pc.generalization,
        isHome <- pc.isHome,
        isLandmark <- pc.isLandmark,
        useCase <- pc.useCase
    ),
    pa : UWE!ProcessActivity
```

```
(
    name <- pc.name,
    owner <- pc,
    useCase <- pc.useCase,
    parameter <- Sequence { entryPar }->union( exitPar )
),

-- create input parameter and activity parameter node
entryAPN : UWE!ActivityParameterNode
(
    name <- source.contentClass.name,
    owner <- pa,
    activity <- pa,
    type <- source.contentClass,
    parameter <- entryPar
),
entryPar : UWE!Parameter
(
    name <- source.contentClass.name.firstToLower(),
    direction <- #"in",
    type <- source.contentClass
),

-- conditionally create output parameter and activity parameter node
exitAPN : distinct UWE!ActivityParameterNode foreach( b in targetSeq )
(
    name <- target.contentClass.name,
    owner <- pa,
    activity <- pa,
    type <- target.contentClass,
    parameter <- exitPar
),
exitPar : distinct UWE!Parameter foreach( b in targetSeq )
(
    name <- target.contentClass.name.firstToLower(),
    direction <- #out,
    type <- target.contentClass
),

-- conditionally create activity final node
finalNode : distinct UWE!ActivityFinalNode foreach( b in nTargetSeq )
(
    name <- '',
    owner <- pa,
    activity <- pa
)
```

```
}

rule CreateProcessDataAndFlowForEdit
{
    from pc : UWE!ProcessClass ( pc.ownedBehavior->isEmpty() and
        pc.webProcess().oclIsTypeOf( UWE!Edit ) )
    using
    {
        source : UWE!NavigationClass = let ls : Set( UWE!Link ) = pc.inLinks() in
            if ls->isEmpty() then OclUndefined else ls->asSequence()->first().source() endif;
    }
    to tpc : UWE!ProcessClass
    (
        name <- pc.name,
        isHome <- pc.isHome,
        isLandmark <- pc.isLandmark,
        owner <- pc.owner,
        feature <- pc.feature,
        ownedAttribute <- pc.ownedAttribute->including( inputPCProperty ),
        ownedOperation <- pc.ownedOperation,
        ownedBehavior <- Sequence { pa },
        nestedClassifier <- pc.nestedClassifier,
        package <- pc.package,
        generalization <- pc.generalization,
        isHome <- pc.isHome,
        isLandmark <- pc.isLandmark,
        useCase <- pc.useCase
    ),
    inputPCProperty : UWE!Property
    (
        class_ <- tpc,
        owner <- tpc,
        type <- inputPC,
        isComposite <- true,
        aggregation <- #composite,
        lower <- 1,
        upper <- 1
    ),
    pa : UWE!ProcessActivity
    (
        name <- pc.name,
        owner <- pc,
        useCase <- pc.useCase,
        parameter <- Sequence { entryPar }
    ),
```

```
-- create input parameter and activity parameter node
entryAPN : UWE!ActivityParameterNode
(
    name <- source.contentClass.name,
    owner <- pa,
    activity <- pa,
    type <- source.contentClass,
    parameter <- entryPar
),
entryPar : UWE!Parameter
(
    name <- source.contentClass.name.firstToLower(),
    direction <- #"in",
    type <- source.contentClass
),

-- create user action
userAction : UWE!UserAction
(
    name <- pc.name + 'Input',
    owner <- pa,
    activity <- pa,
    processClass <- inputPC,
    input <- Sequence { inputPin }
),
inputPin : UWE!InputPin
(
    name <- source.contentClass.name.firstToLower(),
    owner <- userAction,
    type <- source.contentClass
),

-- create object flow from input activity parameter node to input pin of user action
inputObjectFlow : UWE!ObjectFlow
(
    name <- '',
    owner <- pa,
    activity <- pa,
    source <- entryAPN,
    target <- inputPin
),

-- create process data class
inputPC : UWE!ProcessClass
(
    name <- pc.name + 'Input',
```

```
        owner <- pc.owningPackage,
        package <- pc.owningPackage,
        owningPackage <- pc.owningPackage,
        contentClass <- source.contentClass,
        ownedAttribute <- pp
    ),
    pp : distinct UWE!ProcessProperty foreach( cp in source.contentClass.allOwnedAttribute()->
        select( p | p.type.oclIsKindOf( UWE!DataType ) and not p.isMultivalued() ) )
    (
        name <- cp.name,
        class_ <- inputPC,
        owner <- inputPC,
        type <- cp.type,
        lower <- cp.lower,
        upper <- cp.upper,
        editProperty <- cp
    ),

    -- create activity final node
    finalNode : UWE!ActivityFinalNode
    (
        name <- '',
        owner <- pa,
        activity <- pa
    ),

    -- create control flow to activity final node
    finalFlow : UWE!ControlFlow
    (
        name <- '',
        owner <- pa,
        activity <- pa,
        source <- userAction,
        target <- finalNode
    )
}

rule CreateProcessDataAndFlowForSimpleProcess
{
    from pc : UWE!ProcessClass ( pc.ownedBehavior->isEmpty() and
        pc.webProcess().oclIsTypeOf( UWE!SimpleProcess ) and
        pc.webProcess().hasTraceTarget( 'SimpleProcess2Operation' ) )
    using
    {
        source : UWE!NavigationClass = let ls : Set( UWE!Link ) = pc.inLinks() in
            if ls->isEmpty() then OclUndefined else ls->asSequence()->first().source() endif;
```

```
    target : UWE!NavigationClass = let ls : Set( UWE!Link ) = pc.outLinks() in
        if ls->isEmpty() then OclUndefined else ls->asSequence()->first().target() endif;
    targetSeq : Sequence( Boolean ) = if target.oclIsUndefined() then Sequence {} else
        Sequence { true } endif;
    nTargetSeq : Sequence( Boolean ) = if target.oclIsUndefined() then Sequence { true } else
        Sequence {} endif;
    o : UWE!Operation = pc.webProcess().getTraceTarget( 'SimpleProcess2Operation' );
    inputPar : Sequence( UWE!Parameter ) = o.ownedParameter->select( p |
        p.direction <> #return );
    parSeq : Sequence( Boolean ) = if inputPar->isEmpty() then Sequence {} else
        Sequence { true } endif;
    nParSeq : Sequence( Boolean ) = if inputPar->notEmpty() then Sequence {} else
        Sequence { true } endif;
    typeSeq : Sequence( Boolean ) = if o.type.oclIsUndefined() then Sequence {} else
        Sequence { true } endif;
    nTypeSeq : Sequence( Boolean ) = if o.type.oclIsUndefined() then Sequence { true } else
        Sequence {} endif;
}
to tpc : UWE!ProcessClass
(
    name <- pc.name,
    isHome <- pc.isHome,
    isLandmark <- pc.isLandmark,
    owner <- pc.owner,
    feature <- pc.feature,
    ownedAttribute <- pc.ownedAttribute->union( inputPCProperty ),
    ownedOperation <- pc.ownedOperation,
    ownedBehavior <- Sequence { pa },
    nestedClassifier <- pc.nestedClassifier,
    package <- pc.package,
    generalization <- pc.generalization,
    isHome <- pc.isHome,
    isLandmark <- pc.isLandmark,
    useCase <- pc.useCase
),
pa : UWE!ProcessActivity
(
    name <- pc.name,
    owner <- pc,
    useCase <- pc.useCase,
    parameter <- Sequence { entryPar }->union( exitPar )
),

-- create input parameter and activity parameter node
entryAPN : UWE!ActivityParameterNode
(
```

```
        name <- source.contentClass.name,
        owner <- pa,
        activity <- pa,
        type <- source.contentClass,
        parameter <- entryPar,
        outgoing <- flowToForkNode->union( directTargetFlow )
    ),
    entryPar : UWE!Parameter
    (
        name <- source.contentClass.name.firstToLower(),
        direction <- #"in",
        type <- source.contentClass
    ),

    -- create call operation action with target and input pins
    coa : UWE!CallOperationAction
    (
        name <- o.name,
        owner <- pa,
        operation <- o,
        activity <- pa,
        input <- inputPin->including( targetPin ),
        output <- resultPin,
        target <- targetPin
    ),
    targetPin : UWE!InputPin
    (
        name <- 'target',
        owner <- coa,
        type <- o.class_,
        incoming <- flowFromForkNode2->union( directTargetFlow )
    ),
    inputPin : distinct UWE!InputPin foreach ( p in inputPar )
    (
        name <- p.name,
        owner <- coa,
        type <- p.type
    ),

    -- create user action and process data class if operation has parameters
    userAction : distinct UWE!UserAction foreach ( b in parSeq )
    (
        name <- pc.name + 'Input',
        owner <- pa,
        activity <- pa,
        processClass <- inputPC
```

```
),
inputPC : distinct UWE!ProcessClass foreach ( b in parSeq )
(
    name <- pc.name + 'Input',
    owner <- pc.owningPackage,
    package <- pc.owningPackage,
    owningPackage <- pc.owningPackage
),
inputPCProperty : distinct UWE!Property foreach ( b in parSeq )
(
    isComposite <- true,
    aggregation <- #composite,
    type <- inputPC,
    class_ <- pc,
    owner <- pc,
    lower <- 1,
    upper <- 1
),

--  create properties, output pins and object flows from the user action to the
    call operation action for all parameters
pp : distinct UWE!ProcessProperty foreach( p in inputPar )
(
    name <- p.name,
    type <- p.type,
    lower <- p.lower,
    upper <- p.upper
),
outputPin : distinct UWE!OutputPin foreach ( p in inputPar )
(
    name <- p.name,
    type <- p.type
),
outputObjectFlow : distinct UWE!ObjectFlow foreach ( p in inputPar )
(
    name <- '',
    owner <- pa,
    activity <- pa,
    source <- outputPin,
    target <- inputPin
),

-- create output pin and object flow if operation has a return type
resultPin : distinct UWE!OutputPin foreach ( b in typeSeq )
(
    name <- 'result',
```

```
    owner <- coa,
    type <- o.type
),
resultObjectFlow : distinct UWE!ObjectFlow foreach ( b in typeSeq )
(
    name <- '',
    owner <- pa,
    activity <- pa,
    source <- resultPin,
    target <- exitAPN
),

-- conditionally create output parameter and activity parameter node
exitAPN : distinct UWE!ActivityParameterNode foreach( b in targetSeq )
(
    name <- target.contentClass.name,
    owner <- pa,
    activity <- pa,
    type <- target.contentClass,
    parameter <- exitPar
),
exitPar : distinct UWE!Parameter foreach( b in targetSeq )
(
    name <- target.contentClass.name.firstToLower(),
    direction <- #out,
    type <- target.contentClass
),

-- conditionally create fork node and flows if operation has parameters
forkNode : distinct UWE!ForkNode foreach ( b in parSeq )
(
    name <- '',
    owner <- pa,
    activity <- pa
),
flowToForkNode : distinct UWE!ObjectFlow foreach ( b in parSeq )
(
    name <- '',
    owner <- pa,
    activity <- pa,
    target <- forkNode
),
flowFromForkNode1 : distinct UWE!ControlFlow foreach ( b in parSeq )
(
    name <- '',
    owner <- pa,
```

```
    activity <- pa,
    source <- forkNode,
    target <- userAction
),
flowFromForkNode2 : distinct UWE!ObjectFlow foreach ( b in parSeq )
(
    name <- '',
    owner <- pa,
    activity <- pa,
    source <- forkNode
),

-- conditionally object flow if operation has no parameters
directTargetFlow : distinct UWE!ObjectFlow foreach ( b in nParSeq )
(
    name <- '',
    owner <- pa,
    activity <- pa
),

-- conditionally create activity final node and control flow
finalNode : distinct UWE!ActivityFinalNode foreach( b in nTargetSeq )
(
    name <- '',
    owner <- pa,
    activity <- pa
),
finalControlFlow : distinct UWE!ControlFlow foreach ( b in nTypeSeq )
(
    name <- '',
    owner <- pa,
    activity <- pa,
    source <- coa,
    target <- finalNode
)
do
{
    for( ipc in inputPC )
    {
        ipc.ownedAttribute <- pp;
        for( p in pp  )
        {
            p.class_ <- ipc;
            p.owner <- ipc;
        }
    }
```

```
        for( ua in userAction )
        {
            ua.output <- outputPin;
            for( op in outputPin )
            {
                op.owner <- ua;
            }
        }
        for( rp in resultPin )
        {
            rp.owner <- coa;
        }
    }
}
```

B.3.9 Transformation NavigationAndProcess2Presentation

```
module NavigationAndProcess2Presentation;
create OUT : UWE refining IN : UWE;

uses UWEHelpers;
uses strings;

-- INCLUDE Refinement Header HERE
-- INCLUDE Trace Header HERE

helper context UWE!Package def : isNavigationPackage() : Boolean =
    self.ownedMember->exists( el | el.oclIsKindOf( UWE!NavigationNode ) ) or
    self.ownedMember->select( el | el.oclIsKindOf( UWE!Package ) )->exists( p |
p.isNavigationPackage() );

helper context UWE!Package def : isNavigationPackage() : Boolean =
    self.ownedMember->exists( el | el.oclIsKindOf( UWE!NavigationNode ) ) or
    self.ownedMember->select( el | el.oclIsKindOf( UWE!Package ) )->exists( p |
p.isNavigationPackage() );

-- rule for copying package structure from navigation to presentation, not outlined in 4.6.2

rule NavigationPackage2PresentationPackage
{
    from p : UWE!Package ( p.oclIsTypeOf( UWE!Package ) and p.isNavigationPackage() and
        not p.hasTraceTarget( 'NavigationPackage2PresentationPackage' ) )
    using
    {
        owningPackage : UWE!Package = let traceP : UWE!Package =
```

```
            p.owningPackage.getTraceTarget( 'NavigationPackage2PresentationPackage' ) in
            if traceP.oclIsUndefined() then
                if p.owningPackage.oclIsTypeOf( UWE!Model ) then p.owningPackage
                else thisModule.resolveTemp( p.owningPackage, 'pp' ) endif
            else traceP endif;
    }
    to tp : UWE!Package
    (
        name <- p.name,
        owner <- p.owner,
        ownedMember <- p.ownedMember
    ),
    pp : UWE!Package
    (
        name <- if p.owningPackage.oclIsTypeOf( UWE!Model ) then 'Presentation'
            else p.name endif,
        owner <- owningPackage,
        owningPackage <-owningPackage
    )
    do
    {
        thisModule.CreateTrace( p, pp, 'NavigationPackage2PresentationPackage' );
    }
}

rule NavigationClass2PresentationClass
{
    from nn : UWE!NavigationClass ( not nn.isAbstract and
        nn.oclIsTypeOf( UWE!NavigationClass ) and
        not nn.hasTraceTarget( 'Node2PresentationClass' ) )
    using
    {
        owningPackage : UWE!Package =
            if nn.owningPackage.oclIsUndefined() then OclUndefined else
                let traceP : UWE!Package = nn.owningPackage.getTraceTarget(
                    'NavigationPackage2PresentationPackage' ) in
                    if traceP.oclIsUndefined() then thisModule.resolveTemp( nn.owningPackage, 'pp' )
                    else traceP endif
            endif;
        allOwnedAttribute : Sequence( UWE!Property ) = nn.allOwnedAttribute();
        textAttribute : Sequence( UWE!Property ) = allOwnedAttribute->select( p |
            p.type.oclIsKindOf( UWE!DataType ) );
        anchorAttribute : Sequence( UWE!Property ) = allOwnedAttribute->select( p |
            p.association.oclIsKindOf( UWE!Link ) );
        pcAttribute : Sequence( UWE!Property ) = allOwnedAttribute->select( p | p.isComposite and
            p.type.oclIsKindOf( UWE!NavigationNode ) and not p.type.isAbstract and
```

```
            not p.association.oclIsKindOf( UWE!Link ) );
    }
    to tnn : UWE!NavigationClass
    (
        -- general from Node2PresentationClass
        name <- nn.name,
        isHome <- nn.isHome,
        isLandmark <- nn.isLandmark,
        owner <- nn.owner,
        ownedAttribute <- nn.ownedAttribute,
        ownedOperation <- nn.ownedOperation,
        ownedBehavior <- nn.ownedBehavior,
        nestedClassifier <- nn.nestedClassifier,
        useCase <- nn.useCase,
        generalization <- nn.generalization,
        -- specific
        contentClass <- nn.contentClass
    ),
    pc : UWE!PresentationClass
    (
        -- general from Node2PresentationClass
        name <- nn.name,
        node <- nn,
        owningPackage <- owningPackage,
        -- specific
        ownedAttribute <- textPps->union( anchorPps )->union( pcPps ),
        nestedClassifier <- textUis->union( anchorUis )
    ),
    textPps : distinct UWE!PresentationProperty foreach ( p in textAttribute )
    (
        name <- '',
        owner <- pc,
        class_ <- pc,
        aggregation <- #composite,
        isComposite <- true,
        navigationProperty <- p,
        type <- textUis,
        lower <- p.lower,
        upper <- p.upper
    ),
    textUis : distinct UWE!Text foreach ( p in textAttribute )
    (
        name <- p.name.firstToUpper(),
        owner <- pc
    ),
    anchorPps : distinct UWE!PresentationProperty foreach ( p in anchorAttribute )
```

```
    (
        name <- '',
        owner <- pc,
        class_ <- pc,
        aggregation <- #composite,
        isComposite <- true,
        navigationProperty <- p,
        type <- anchorUis,
        lower <- p.lower,
        upper <- p.upper
    ),
    anchorUis : distinct UWE!Anchor foreach ( p in anchorAttribute )
    (
        name <- p.type.name,
        owner <- pc
    ),
    pcPps : distinct UWE!PresentationProperty foreach ( p in pcAttribute )
    (
        name <- '',
        owner <- pc,
        class_ <- pc,
        aggregation <- #composite,
        isComposite <- true,
        navigationProperty <- p,
        type <- let traceT : UWE!PresentationClass =
            nn.getTraceTarget( 'Node2PresentationClass' ) in
            if traceT.oclIsUndefined() then thisModule.resolveTemp( p.type, 'pc' ) else traceT endif,
        lower <- p.lower,
        upper <- p.upper
    )
    do
    {
        thisModule.CreateTrace( nn, pc, 'Node2PresentationClass' );
    }
}

rule Menu2PresentationClass
{
    from nn : UWE!Menu ( not nn.isAbstract and
        not nn.hasTraceTarget( 'Node2PresentationClass' ) )
    using
    {
        owningPackage : UWE!Package =
            if nn.owningPackage.oclIsUndefined() then OclUndefined else
                let traceP : UWE!Package = nn.owningPackage.getTraceTarget(
                'NavigationPackage2PresentationPackage' ) in
```

```
                if traceP.oclIsUndefined() then thisModule.resolveTemp( nn.owningPackage, 'pp' )
                else traceP endif
        endif;
    allOwnedAttribute : Sequence( UWE!Property ) = nn.allOwnedAttribute();
    textAttribute : Sequence( UWE!Property ) = allOwnedAttribute->select( p |
        p.type.oclIsKindOf( UWE!DataType ) );
    anchorAttribute : Sequence( UWE!Property ) = allOwnedAttribute->select( p |
        p.association.oclIsKindOf( UWE!Link ) );
    pcAttribute : Sequence( UWE!Property ) = allOwnedAttribute->select( p | p.isComposite and
        p.type.oclIsKindOf( UWE!NavigationNode ) and not p.type.isAbstract and
        not p.association.oclIsKindOf( UWE!Link ) );
}
to tnn : UWE!Menu
(
    -- general from Node2PresentationClass
    name <- nn.name,
    isHome <- nn.isHome,
    isLandmark <- nn.isLandmark,
    owner <- nn.owner,
    ownedAttribute <- nn.ownedAttribute,
    ownedOperation <- nn.ownedOperation,
    ownedBehavior <- nn.ownedBehavior,
    nestedClassifier <- nn.nestedClassifier,
    useCase <- nn.useCase,
    generalization <- nn.generalization,
    -- specific
    contentClass <- nn.contentClass
),
pc : UWE!PresentationClass
(
    -- general from Node2PresentationClass
    name <- nn.name,
    node <- nn,
    owningPackage <- owningPackage,
    -- specific
    ownedAttribute <- textPps->union( anchorPps )->union( pcPps ),
    nestedClassifier <- textUis->union( anchorUis )
),
textPps : distinct UWE!PresentationProperty foreach ( p in textAttribute )
(
    name <- '',
    owner <- pc,
    class_ <- pc,
    aggregation <- #composite,
    isComposite <- true,
    navigationProperty <- p,
```

```
        type <- textUis,
        lower <- p.lower,
        upper <- p.upper
    ),
    textUis : distinct UWE!Text foreach ( p in textAttribute )
    (
        name <- p.name.firstToUpper(),
        owner <- pc
    ),
    anchorPps : distinct UWE!PresentationProperty foreach ( p in anchorAttribute )
    (
        name <- '',
        owner <- pc,
        class_ <- pc,
        aggregation <- #composite,
        isComposite <- true,
        navigationProperty <- p,
        type <- anchorUis,
        lower <- p.lower,
        upper <- p.upper
    ),
    anchorUis : distinct UWE!Anchor foreach ( p in anchorAttribute )
    (
        name <- p.type.name,
        owner <- pc
    ),
    pcPps : distinct UWE!PresentationProperty foreach ( p in pcAttribute )
    (
        name <- '',
        owner <- pc,
        class_ <- pc,
        aggregation <- #composite,
        isComposite <- true,
        navigationProperty <- p,
        type <- let traceT : UWE!PresentationClass =
            nn.getTraceTarget( 'Node2PresentationClass' ) in
            if traceT.oclIsUndefined() then thisModule.resolveTemp( p.type, 'pc' ) else traceT endif,
        lower <- p.lower,
        upper <- p.upper
    )
    do
    {
        thisModule.CreateTrace( nn, pc, 'Node2PresentationClass' );
    }
}
```

```
rule Index2PresentationClass
{
    from nn : UWE!Index ( not nn.isAbstract and
        not nn.hasTraceTarget( 'Node2PresentationClass' ) )
    using
    {
        owningPackage : UWE!Package =
            if nn.owningPackage.oclIsUndefined() then OclUndefined else
                let traceP : UWE!Package = nn.owningPackage.getTraceTarget(
                'NavigationPackage2PresentationPackage' ) in
                    if traceP.oclIsUndefined() then thisModule.resolveTemp( nn.owningPackage, 'pp' )
                    else traceP endif
            endif;
    }
    to tnn : UWE!Index
    (
        -- general from Node2PresentationClass
        name <- nn.name,
        isHome <- nn.isHome,
        isLandmark <- nn.isLandmark,
        owner <- nn.owner,
        ownedAttribute <- nn.ownedAttribute,
        ownedOperation <- nn.ownedOperation,
        ownedBehavior <- nn.ownedBehavior,
        nestedClassifier <- nn.nestedClassifier,
        useCase <- nn.useCase,
        generalization <- nn.generalization
        -- specific
    ),
    pc : UWE!PresentationClass
    (
        -- general from Node2PresentationClass
        name <- nn.name,
        node <- nn,
        owningPackage <- owningPackage,
        -- specific
        ownedAttribute <- Sequence { anchorPp },
        nestedClassifier <- Sequence { anchorUi }
    ),
    anchorPp : UWE!PresentationProperty
    (
        name <- '',
        owner <- pc,
        class_ <- pc,
        aggregation <- #composite,
        isComposite <- true,
```

```
        type <- anchorUi,
        lower <- 0,
        upper <- 0-1
    ),
    anchorUi : UWE!Anchor
    (
        name <- nn.outLinks()->first().target().name,
        owner <- pc
    )
    do
    {
        thisModule.CreateTrace( nn, pc, 'Node2PresentationClass' );
    }
}

rule ProcessClass2PresentationClass
{
    from nn : UWE!ProcessClass ( not nn.isAbstract and nn.inLinks()->isEmpty() and
        not nn.hasTraceTarget( 'Node2PresentationClass' ) )
    using
    {
        owningPackage : UWE!Package =
            if nn.owningPackage.oclIsUndefined() then OclUndefined else
                let traceP : UWE!Package = nn.owningPackage.getTraceTarget(
                    'NavigationPackage2PresentationPackage' ) in
                    if traceP.oclIsUndefined() then thisModule.resolveTemp( nn.owningPackage, 'pp' )
                    else traceP endif
            endif;
        allOwnedAttribute : Sequence( UWE!Property ) = nn.allOwnedAttribute();
        textInputAttribute : Sequence( UWE!Property ) = allOwnedAttribute->select( p |
            p.type.oclIsKindOf( UWE!PrimitiveType ) );
        enumerationInputAttribute : Sequence( UWE!Property ) = allOwnedAttribute->select( p |
            p.type.oclIsKindOf( UWE!Enumeration ) );
        selectionAttribute : Sequence( UWE!Property ) = allOwnedAttribute->select( p |
            p.type.oclIsTypeOf( UWE!Class ) );
    }
    to tnn : UWE!ProcessClass
    (
        -- general from Node2PresentationClass
        name <- nn.name,
        isHome <- nn.isHome,
        isLandmark <- nn.isLandmark,
        owner <- nn.owner,
        ownedAttribute <- nn.ownedAttribute,
        ownedOperation <- nn.ownedOperation,
        ownedBehavior <- nn.ownedBehavior,
```

```
        nestedClassifier <- nn.nestedClassifier,
        useCase <- nn.useCase,
        generalization <- nn.generalization,
        -- specific
        contentClass <- nn.contentClass
    ),
    pc : UWE!PresentationClass
    (
        -- general from Node2PresentationClass
        name <- nn.name,
        node <- nn,
        owningPackage <- owningPackage,
        -- specific
        ownedAttribute <- textInputPps->union( enumerationInputPps )->union( selectionPps ),
        nestedClassifier <- textInputUis->union( enumerationInputUis )->union( selectionUis )
    ),
    textInputPps : distinct UWE!PresentationProperty foreach ( p in textInputAttribute )
    (
        name <- '',
        owner <- pc,
        class_ <- pc,
        aggregation <- #composite,
        isComposite <- true,
        navigationProperty <- p,
        type <- textInputUis,
        lower <- p.lower,
        upper <- p.upper
    ),
    textInputUis : distinct UWE!TextInput foreach ( p in textInputAttribute )
    (
        name <- p.name.firstToUpper(),
        owner <- pc
    ),
    enumerationInputPps : distinct UWE!PresentationProperty
        foreach ( p in enumerationInputAttribute )
    (
        name <- '',
        owner <- pc,
        class_ <- pc,
        aggregation <- #composite,
        isComposite <- true,
        navigationProperty <- p,
        type <- enumerationInputUis,
        lower <- p.lower,
        upper <- p.upper
    ),
```

```
    enumerationInputUis : distinct UWE!EnumerationInput foreach ( p in enumerationInputAttribute )
    (
        name <- p.name.firstToUpper(),
        owner <- pc
    ),
    selectionPps : distinct UWE!PresentationProperty foreach ( p in selectionAttribute )
    (
        name <- '',
        owner <- pc,
        class_ <- pc,
        aggregation <- #composite,
        isComposite <- true,
        navigationProperty <- p,
        type <- selectionUis,
        lower <- p.lower,
        upper <- p.upper
    ),
    selectionUis : distinct UWE!Selection foreach ( p in selectionAttribute )
    (
        name <- p.name.firstToUpper(),
        owner <- pc,
        format <- p.type.name
    )
    do
    {
        thisModule.CreateTrace( nn, pc, 'Node2PresentationClass' );
    }
}
```

B.4 PIM2PSM Transformations

This section comprises the implementation of the PIM2PSM transformations for the transformation of the platform independent design models to the platform specific implementation models as presented in Chapter 5. All PIM2PSM transformations are regular, i.e. non refining, transformations using the actual version of the ATL compiler, called ATL 2006, which supports rule inheritance. For activating the ATL 2006 compiler, the following code line has to be placed at the beginning of a transformation:

```
-- @atlcompiler atl2006
```

B.4.1 Configuration Header

The following transformation rules and helpers are included by the transformations *Navigation2Conf* and *Process2Conf* in order to generate XML configuration data for the runtime environment as presented in 5.1.3.

```
helper context OclAny def : strippedTypeName() : String =
    let n : String = self.oclType().toString() in
        let i : Integer = n.indexOf( '!' ) in
            if i < 0 then n else n.substring( i+2, n.size() ) endif;

helper context OclAny def : isPrimitive() : Boolean =
    self.oclIsTypeOf( Boolean ) or self.oclIsTypeOf( Integer ) or self.oclIsTypeOf( Real )
    or self.oclIsTypeOf( String );

helper def : processPackageName : String = 'uwe.runtime.process';

helper def : confIdCounter : Integer = 0;
helper def : confIdMap : Map( UWE!NamedElement, String ) = Map{};
helper def : beansNode : XML!Element = OclUndefined;

helper context UWE!NamedElement def : getId() : String =
    let mapId : String = thisModule.confIdMap->get( self ) in
        if mapId.oclIsUndefined() then let qn : String = self.qualifiedId() in
            if qn.oclIsUndefined() or qn = '' then
                thisModule.CreateId( self )
            else self.strippedTypeName() + '_' + qn endif
        else mapId endif;

rule CreateId( el : UWE!NamedEl )
{
    do
    {
        -- increase id counter
        thisModule.confIdCounter <- thisModule.confIdCounter + 1;

        -- rememberid  in global map
        thisModule.confIdMap <- thisModule.confIdMap->including( el, 'id_' +
            thisModule.confIdCounter.toString() );
        'id_' + thisModule.confIdCounter.toString();
    }
}

abstract rule NamedElement2Conf
{
```

```
    from el : UWE!NamedElement
    to beanEl : XML!Element
    (
        name <- 'bean',
        parent <- thisModule.beansNode
    ),
    idAttr : XML!Attribute
    (
        name <- 'id',
        value <- el.getId(),
        parent <- beanEl
    )
}

rule CreateConfProperty( parent : XML!Element, name : String, value : OclAny )
{
    to propertyEl : XML!Element
    (
        name <- 'property',
        parent <- parent
    ),
    nameAttr : XML!Attribute
    (
        name <- 'name',
        value <- name,
        parent <- propertyEl
    )
    do
    {
        thisModule.CreateConfPropertyValue( propertyEl, value );
    }
}

rule CreateConfPropertyValue( parent : XML!Element, value : OclAny )
{
    do
    {
        if( value.isPrimitive() )
        {
            thisModule.CreateConfPropertyPrimitiveValue( parent, value );
        }
        else
        {
            if( value.oclIsKindOf( UWE!NamedElement ) )
            {
                thisModule.CreateConfPropertyRefValue( parent, value );
```

```
            }
            else
            {
                if( value.oclIsKindOf( Set( OclAny ) ) )
                {
                    thisModule.CreateConfPropertySetValue( parent, value );
                }
                else
                {
                    if( value.oclIsKindOf( Sequence( OclAny ) ) )
                    {
                        thisModule.CreateConfPropertySequenceValue( parent, value );
                    }
                    else
                    {
                        value.debug( 'Property value cannot be converted to conf' );
                    }
                }
            }
        }
    }
}

rule CreateConfPropertyPrimitiveValue( parent : XML!Element, value : OclAny )
{
    to valueEl : XML!Element
    (
        name <- 'value',
        parent <- parent
    ),
    stringValue : XML!TextNode
    (
        value <- value.toString(),
        parent <- valueEl
    )
}

rule CreateConfPropertyRefValue( parent : XML!Element, value : UWE!NamedElement )
{
    to refEl : XML!Element
    (
        name <- 'ref',
        parent <- parent
    ),
    beanAttr : XML!Attribute
    (
```

```
        name <- 'bean',
        value <- value.getId(),
        parent <- refEl
    )
}

rule CreateConfPropertySetValue( parent : XML!Element, value : Set( OclAny ) )
{
    to setEl : XML!Element
    (
        name <- 'set',
        parent <- parent
    )
    do
    {
        for( v in value )
        {
            thisModule.CreateConfPropertyValue( setEl, v );
        }
    }
}

rule CreateConfPropertySequenceValue( parent : XML!Element,
    value : Sequence( OclAny ) )
{
    to listEl : XML!Element
    (
        name <- 'list',
        parent <- parent
    )
    do
    {
        for( v in value )
        {
            thisModule.CreateConfPropertyValue( listEl, v );
        }
    }
}
```

B.4.2 Transformation Content2JavaBeans

```
module Content2JavaBeans;
create OUT : JAVA from IN : UWE;

uses strings;
```

```
uses Java;
uses UWEHelpers;

helper def : utilPck : JAVA!Package = OclUndefined;

helper context UWE!Class def : fullJavaName() : String =
    if self.owningPackage.oclIsUndefined() then '' else self.owningPackage.fullJavaName() + '.'
    endif + self.name;

helper context UWE!Package def : fullJavaName() : String =
    let qn : String = self.qualifiedNameBySeparator( '.' ) in
        if qn.oclIsUndefined() then '' else qn.toLower() + '.' endif + 'beans';

entrypoint rule CreateSingletons()
{
 to utilPck : JAVA!Package
 (
    name <- 'java.util',
    isImported <- true
 )
 do
 {
  thisModule.utilPck <- utilPck;
 }
}

rule Package2Package
{
    from p : UWE!Package ( p.oclIsTypeOf( UWE!Package ) andp.ownedMember->exists( el |
        el.oclIsTypeOf( UWE!Class ) ) )
    to jp : JAVA!Package
    (
        name <- p.fullJavaName(),
        isImported <- false
    )
}

rule Class2Class
{
    from c : UWE!Class ( c.oclIsTypeOf( UWE!Class ) )
    to jc : JAVA!JavaClass
    (
        name <- c.name,
        package <- c.package,
        superClasses <- c.generalization->collect( g | g.general ),
        isAbstract <- false,
```

```
        isPublic <- true,
        isInterface <- false
    )
}

rule PrimitiveType2PrimitiveType
{
    from pt : UWE!PrimitiveType
    to jpt : JAVA!PrimitiveType
    (
        name <- pt.name
    )
}

rule Enumeration2Enumeration
{
    from e : UWE!Enumeration ( UWE!Property.allInstances()->exists( p |
        p.class_.oclIsTypeOf( UWE!Class ) and p.type = e ) )
    to je : JAVA!Enumeration
    (
        name <- e.name,
        package <- e.package,
        enumerationLiterals <- e.ownedLiteral->collect( el |
            thisModule.EnumerationLiteral2EnumerationLiteral( el ) )
    )
}

lazy rule EnumerationLiteral2EnumerationLiteral
{
    from el : UWE!EnumerationLiteral
    to jel : JAVA!EnumerationLiteral
    (
        name <- el.name
    )
}

rule Property2ClassMembers
{
    from p : UWE!Property ( p.class_.oclIsTypeOf( UWE!Class ) and
        ( p.type.oclIsKindOf( UWE!DataType ) or p.type.oclIsTypeOf( UWE!Class ) ) and
        not p.isDerived )
    to field : JAVA!Field
    (
        owner <- p.class_,
        name <- '_' + p.name,
        type <- if p.isMultivalued() then
```

```
            if p.isOrdered.oclIsUndefined() then thisModule.Class2ParameterizedList( p.type )
            else if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
                thisModule.Class2ParameterizedSet( p.type ) endif
            endif
        else p.type endif,
    isPublic <- false,
    isStatic <- false,
    initializer <- if p.type.name = 'String' then """" else
            if p.isMultivalued() then
                if p.isOrdered.oclIsUndefined() then 'new java.util.ArrayList<' +
                p.type.fullJavaName() + '>()' else
                    if p.isOrdered then 'new java.util.ArrayList<' + p.type.fullJavaName() + '>()'
                    else 'new java.util.HashSet<' + p.type.fullJavaName() + '>()' endif
                endif
            else OclUndefined endif
        endif
),
getter : JAVA!Method
(
    owner <- p.class_,
    name <- 'get' + p.name.stringFirstToUpper(),
    type <- if p.isMultivalued() then
            if p.isOrdered.oclIsUndefined() then thisModule.Class2ParameterizedList( p.type )
            else if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
                thisModule.Class2ParameterizedSet( p.type ) endif
            endif
        else p.type endif,
    isPublic <- true,
    isStatic <- false,
    body <- 'return ' + '_' + p.name + ';'
),
setter : JAVA!Method
(
    owner <- p.class_,
    name <- 'set' + p.name.stringFirstToUpper(),
    isPublic <- true,
    isStatic <- false,
    parameters <- Sequence { setterParameter },
    body <- 'this.' + '_' + p.name + ' = ' + '_' + p.name + ';'
),
setterParameter : JAVA!MethodParameter
(
    name <- '_' + p.name,
    type <- if p.isMultivalued() then
            if p.isOrdered.oclIsUndefined() then thisModule.Class2ParameterizedList( p.type )
                else if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
```

```
                    thisModule.Class2ParameterizedSet( p.type ) endif
                endif
            else p.type endif
    )
}

rule DerivedProperty2ClassMembers
{
    from p : UWE!Property ( p.class_.oclIsTypeOf( UWE!Class ) and
        ( p.type.oclIsKindOf( UWE!DataType ) or p.type.oclIsTypeOf( UWE!Class ) ) and
        p.isDerived )
    to getter : JAVA!Method
    (
        owner <- p.class_,
        name <- 'get' + p.name.stringFirstToUpper(),
        type <- if p.isMultivalued() then
                if p.isOrdered.oclIsUndefined() then thisModule.Class2ParameterizedList( p.type )
                else if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
                    thisModule.Class2ParameterizedSet( p.type ) endif
                endif
            else p.type endif,
        isPublic <- true,
        isStatic <- false,
        body <- if p.type.oclIsUndefined() then '' else
            if p.type.oclIsKindOf( UWE!DataType ) then
                if p.type.name = 'void' then '' else
                    if p.type.name = 'Boolean' then 'return false;' else
                        'return (' + p.type.name + ')0;'
                    endif
                endif
            else
                'return null;'
            endif
        endif
    )
}

rule Operation2Method
{
    from o : UWE!Operation ( o.class_.oclIsTypeOf( UWE!Class ) )
    using
    {
        formalParameters : Sequence ( UWE!Parameter ) = o.ownedParameter->select( op |
            op.direction <> #return );
    }
    to m : JAVA!Method
```

```
    (
        name <- o.name,
        owner <- o.class_,
        isPublic <- true,
        isStatic <- false,
        parameters <- parameters,
        type <- if o.type.oclIsKindOf( UWE!PrimitiveType ) then
            thisModule.PrimitiveType2PrimitiveType( o.type ) else o.type endif,
        body <- if o.type.oclIsUndefined() then '' else
            if o.type.oclIsKindOf( UWE!DataType ) then
                if o.type.name = 'void' then '' else
                    if o.type.name = 'Boolean' then 'return false;' else
                        'return (' + o.type.name + ')0;'
                    endif
                endif
            else
                'return null;'
            endif
        endif
    ),
    parameters : distinct JAVA!MethodParameter foreach ( p in formalParameters )
    (
        name <- '_' + p.name,
        type <- p.type
    )
}

unique lazy rule Class2ParameterizedSet
{
    from c : UWE!Class
    to s : JAVA!JavaClass
    (
        name <- 'Set',
        package <- thisModule.utilPck,
        isAbstract <- false,
        isPublic <- true,
        isInterface <- true,
        actualTypeParameters <- Sequence { c }
    )
}

unique lazy rule Class2ParameterizedList
{
    from c : UWE!Class
    to s : JAVA!JavaClass
    (
```

```
        name <- 'List',
        package <- thisModule.utilPck,
        isAbstract <- false,
        isPublic <- true,
        isInterface <- true,
        actualTypeParameters <- Sequence { c }
    )
}
```

B.4.3 Transformation Content2RMIInterfaces

```
module Content2RMIInterfaces;
create OUT : JAVA from IN : UWE;

uses strings;
uses Java;
uses UWEHelpers;

helper def : utilPck : JAVA!Package = OclUndefined;
helper def : rmiPck : JAVA!Package = OclUndefined;
helper def : remoteClass : JAVA!Cass = OclUndefined;
helper def : remoteException : JAVA!Cass = OclUndefined;

helper context UWE!Class def : fullJavaName() : String =
    if self.owningPackage.oclIsUndefined() then ''
    else self.owningPackage.fullJavaName() + '.' endif + self.name;

helper context UWE!Package def : fullJavaName() : String =
    let qn : String = self.qualifiedNameBySeparator( '.' ) in
        if qn.oclIsUndefined() then '' else qn.toLower() + '.' endif + 'rmi';

entrypoint rule CreateSingletons()
{
  to utilPck : JAVA!Package
  (
    name <- 'java.util',
    isImported <- true
  ),
  rmiPck : JAVA!Package
  (
    name <- 'java.rmi',
    isImported <- true
  ),
  remoteClass : JAVA!JavaClass
  (
```

```
    name <- 'Remote',
    package <- rmiPck,
    isAbstract <- false,
    isPublic <- true,
    isInterface <- true
  ),
  remoteException : JAVA!JavaClass
  (
    name <- 'RemoteException',
    package <- rmiPck,
    isAbstract <- false,
    isPublic <- true,
    isInterface <- false
  )
  do
  {
    thisModule.utilPck <- utilPck;
    thisModule.rmiPck <- rmiPck;
    thisModule.remoteClass <- remoteClass;
    thisModule.remoteException <- remoteException;
  }
}

rule Package2Package
{
    from p : UWE!Package ( p.oclIsTypeOf( UWE!Package ) andp.ownedMember->exists( el |
        el.oclIsTypeOf( UWE!Class ) ) )
    to jp : JAVA!Package
    (
        name <- p.fullJavaName(),
        isImported <- false
    )
}

rule Class2Interfaces
{
    from c : UWE!Class ( c.oclIsTypeOf( UWE!Class ) )
    to jc : JAVA!JavaClass
    (
        name <- c.name,
        package <- c.package,
        superClasses <- c.generalization->collect( g | g.general )->including(
            thisModule.remoteClass ),
        isAbstract <- false,
        isPublic <- true,
        isInterface <- true
```

```
    )
}

rule PrimitiveType2PrimitiveType
{
    from pt : UWE!PrimitiveType
    to jpt : JAVA!PrimitiveType
    (
        name <- pt.name
    )
}

rule Enumeration2Enumeration
{
    from e : UWE!Enumeration ( UWE!Property.allInstances()->exists( p |
        p.class_.oclIsTypeOf( UWE!Class ) and p.type = e ) )
    to je : JAVA!Enumeration
    (
        name <- e.name,
        package <- e.package,
        enumerationLiterals <- e.ownedLiteral->collect( el |
            thisModule.EnumerationLiteral2EnumerationLiteral( el ) )
    )
}

lazy rule EnumerationLiteral2EnumerationLiteral
{
    from el : UWE!EnumerationLiteral
    to jel : JAVA!EnumerationLiteral
    (
        name <- el.name
    )
}

rule Property2ClassMembers
{
    from p : UWE!Property ( p.class_.oclIsTypeOf( UWE!Class ) and
        ( p.type.oclIsKindOf( UWE!DataType ) or p.type.oclIsTypeOf( UWE!Class ) ) and
        not p.isDerived )
    to getter : JAVA!Method
    (
        owner <- p.class_,
        name <- 'get' + p.name.stringFirstToUpper(),
        type <- if p.isMultivalued() then
                if p.isOrdered.oclIsUndefined() then thisModule.Class2ParameterizedList( p.type )
                else if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
```

```
                thisModule.Class2ParameterizedSet( p.type ) endif
            endif
        else p.type endif,
    isPublic <- true,
    isStatic <- false,
    exceptions <- Set { thisModule.remoteException }
),
setter : JAVA!Method
(
    owner <- p.class_,
    name <- 'set' + p.name.stringFirstToUpper(),
    isPublic <- true,
    isStatic <- false,
    parameters <- Sequence { setterParameter },
    exceptions <- Set { thisModule.remoteException }
),
setterParameter : JAVA!MethodParameter
(
    name <- '_' + p.name,
    type <- if p.isMultivalued() then
            if p.isOrdered.oclIsUndefined() then thisModule.Class2ParameterizedList( p.type )
            else if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
                thisModule.Class2ParameterizedSet( p.type ) endif
            endif
        else p.type endif
)
}

rule DerivedProperty2ClassMembers
{
    from p : UWE!Property ( p.class_.oclIsTypeOf( UWE!Class ) and
        ( p.type.oclIsKindOf( UWE!DataType ) or p.type.oclIsTypeOf( UWE!Class ) ) and
        p.isDerived )
    to getter : JAVA!Method
    (
        owner <- p.class_,
        name <- 'get' + p.name.stringFirstToUpper(),
        type <- if p.isMultivalued() then
                if p.isOrdered.oclIsUndefined() then thisModule.Class2ParameterizedList( p.type )
                else if p.isOrdered then thisModule.Class2ParameterizedList( p.type ) else
                    thisModule.Class2ParameterizedSet( p.type ) endif
                endif
            else p.type endif,
        isPublic <- true,
        isStatic <- false,
        exceptions <- Set { thisModule.remoteException }
```

```
    )
}

rule Operation2Method
{
    from o : UWE!Operation ( o.class_.oclIsTypeOf( UWE!Class ) )
    using
    {
        formalParameters : Sequence ( UWE!Parameter ) = o.ownedParameter->select( op |
            op.direction <> #return );
    }
    to m : JAVA!Method
    (
        name <- o.name,
        owner <- o.class_,
        isPublic <- true,
        isStatic <- false,
        parameters <- parameters,
        type <- if o.type.oclIsKindOf( UWE!PrimitiveType ) then
            thisModule.PrimitiveType2PrimitiveType( o.type ) else o.type endif,
        exceptions <- Set { thisModule.remoteException }
    ),
    parameters : distinct JAVA!MethodParameter foreach ( p in formalParameters )
    (
        name <- '_' + p.name,
        type <- p.type
    )
}

unique lazy rule Class2ParameterizedSet
{
    from c : UWE!Class
    to s : JAVA!JavaClass
    (
        name <- 'Set',
        package <- thisModule.utilPck,
        isAbstract <- false,
        isPublic <- true,
        isInterface <- true,
        actualTypeParameters <- Sequence { c }
    )
}

unique lazy rule Class2ParameterizedList
{
    from c : UWE!Class
```

```
    to s : JAVA!JavaClass
    (
        name <- 'List',
        package <- thisModule.utilPck,
        isAbstract <- false,
        isPublic <- true,
        isInterface <- true,
        actualTypeParameters <- Sequence { c }
    )
}
```

B.4.4 Transformation Navigation2Conf

```
module Navigation2Conf;
create OUT : XML from IN : UWE;

uses UWEHelpers;

-- INCLUDE Configuration Header HERE

entrypoint rule CreateList()
{
    to beanEl : XML!Element
    (
        name <- 'bean',
        parent <- beansNode
    ),
    idAttr : XML!Attribute
    (
        name <- 'id',
        value <- 'navigationClassInfos',
        parent <- beanEl
    ),
    classAttr : XML!Attribute
    (
        name <- 'class',
        value <- 'uwe.runtime.ListBean',
        parent <- beanEl
    ),
    rootNode : XML!Root
    (
        children <- Sequence { beansNode },
        documentName <- 'navigation-conf.xml'
    ),
    beansNode : XML!Element
```

```
    (
        name <- 'beans'
    )
    do
    {
        thisModule.CreateConfProperty( beanEl, 'list', UWE!NavigationClass.allInstances() );
        thisModule.beansNode <- beansNode;
    }
}

rule NavigationClass2Conf extends NamedElement2Conf
{
    from el : UWE!NavigationClass
    to classAttr : XML!Attribute
    (
        name <- 'class',
        value <- 'uwe.runtime.NavigationClassInfo',
        parent <- beanEl
    )
    do
    {
        thisModule.CreateConfProperty( beanEl, 'name', el.qualifiedId() );
        thisModule.CreateConfProperty( beanEl, 'specific',
            UWE!Generalization.allInstances()->select( g | g.general = el )->collect( g | g.specific ) );
        thisModule.CreateConfProperty( beanEl, 'contentClass', el.contentClass.fullJavaName() );
    }
}
```

B.4.5 Transformation Process2Conf

```
module Process2Conf;
create OUT : XML from IN : UWE;

uses UWEHelpers;

-- INCLUDE Configuration Header HERE

entrypoint rule CreateList()
{
    to beanEl : XML!Element
    (
        name <- 'bean',
        parent <- beansNode
    ),
    idAttr : XML!Attribute
```

```
    (
        name <- 'id',
        value <- 'processActivities',
        parent <- beanEl
    ),
    classAttr : XML!Attribute
    (
        name <- 'class',
        value <- 'uwe.runtime.ListBean',
        parent <- beanEl
    ),
    rootNode : XML!Root
    (
        children <- Sequence { beansNode },
        documentName <- 'process-conf.xml'
    ),
    beansNode : XML!Element
    (
        name <- 'beans'
    )
    do
    {
        thisModule.CreateConfProperty( beanEl, 'list', UWE!ProcessActivity.allInstances() );
        thisModule.beansNode <- beansNode;
    }
}

rule ProcessActivity2Conf extends NamedElement2Conf
{
    from el : UWE!ProcessActivity
    using
    {
        inputParameterNode : UWE!ActivityParameterNode =
            el.node->select( n | n.oclIsTypeOf( UWE!ActivityParameterNode ) and
            n.incoming->size() = 0 )->asSequence()->first();
        outputParameterNode : UWE!ActivityParameterNode =
            let ns : Set( UWE!ActivityParameterNode ) =
            el.node->select( n | n.oclIsTypeOf( UWE!ActivityParameterNode ) and
            n.outgoing->size() = 0 ) in if ns->size() = 0 then OclUndefined else
            ns->asSequence()->first() endif;
        entryNode : UWE!NavigationNode = let ns : Set( UWE!NavigationNode ) =
            UWE!NavigationNode.allInstances()->select( n | n.ownedAttribute->exists( p |
            p.type = el.owner and if p.association.oclIsUndefined() then false else
            p.association.oclIsTypeOf( UWE!ProcessLink ) endif ) ) in
                if ns->size() = 0 then OclUndefined else ns->asSequence()->first() endif;
        exitNode : UWE!NavigationNode = let ns : Set( UWE!NavigationNode ) =
```

```
            el.owner.ownedAttribute->select( p |
            if p.association.oclIsUndefined() then false else p.association.oclIsTypeOf(
                UWE!ProcessLink ) endif )->collect( p |
                p.type )->select( n | n.oclIsKindOf( UWE!NavigationNode ) ) in
                    if ns->size() = 0 then OclUndefined else ns->asSequence()->first() endif;
    }
    to classAttr : XML!Attribute
    (
        name <- 'class',
        value <- thisModule.processPackageName + '.ProcessActivity',
        parent <- beanEl
    )
    do
    {
        thisModule.CreateConfProperty( beanEl, 'name', el.name );
        thisModule.CreateConfProperty( beanEl, 'processClass', el.owner.qualifiedId() );
        if( not entryNode.oclIsUndefined() )
        {
            thisModule.CreateConfProperty( beanEl, 'entryNode',
                ( let ps : Sequence( UWE!Property ) = entryNode.containingPropertyPath() in
                    if ps->isEmpty() then entryNode else ps->first().class_ endif ).qualifiedId() );
        }
        if( not exitNode.oclIsUndefined() )
        {
            thisModule.CreateConfProperty( beanEl, 'exitNode', exitNode.qualifiedId() );
        }
        thisModule.CreateConfProperty( beanEl, 'activityNodes',
                el.node->including( el.node->select( n | n.oclIsKindOf( UWE!Action ) )->
                collect( a | a.output )->flatten() )->flatten()->
                including( el.node->select( n | n.oclIsKindOf( UWE!Action ) )->collect( a | a.input )->
                    flatten() )->flatten() );
        thisModule.CreateConfProperty( beanEl, 'activityEdges', el.edge );
        if( not inputParameterNode.oclIsUndefined() )
        {
            thisModule.CreateConfProperty( beanEl, 'inputParameterNode', inputParameterNode );
        }
        if( not outputParameterNode.oclIsUndefined() )
        {
            thisModule.CreateConfProperty( beanEl, 'outputParameterNode',
                outputParameterNode );
        }
    }
}

rule ActivityNode2Conf extends NamedElement2Conf
{
```

```
    from el : UWE!ActivityNode
    to classAttr : XML!Attribute
    (
        name <- 'class',
        value <- thisModule.processPackageName + '.' + el.strippedTypeName(),
        parent <- beanEl
    )
    do
    {
        thisModule.CreateConfProperty( beanEl, 'name', el.name );
        thisModule.CreateConfProperty( beanEl, 'activity', el.activity );
        thisModule.CreateConfProperty( beanEl, 'incoming', el.incoming );
        thisModule.CreateConfProperty( beanEl, 'outgoing', el.outgoing );
    }
}

rule Pin2Conf extends ActivityNode2Conf
{
    from el : UWE!Pin ( el.owner.oclIsKindOf( UWE!Action ) )
    do
    {
        thisModule.CreateConfProperty( beanEl, 'name', el.name );
        thisModule.CreateConfProperty( beanEl, 'activity', el.owner.owner );
        thisModule.CreateConfProperty( beanEl, 'incoming', el.incoming );
        thisModule.CreateConfProperty( beanEl, 'outgoing', el.outgoing );
    }
}

rule DecisionNodeOrMergeNode2Conf extends ActivityNode2Conf
{
    from el : UWE!ControlNode ( el.oclIsTypeOf( UWE!DecisionNode )
        or el.oclIsTypeOf( UWE!MergeNode ) )
    to classAttr : XML!Attribute
    (
        name <- 'class',
        value <- thisModule.processPackageName + '.DecisionMergeNode',
        parent <- beanEl
    )
    do
    {
        thisModule.CreateConfProperty( beanEl, 'name', el.name );
        thisModule.CreateConfProperty( beanEl, 'activity', el.activity );
        thisModule.CreateConfProperty( beanEl, 'incoming', el.incoming );
        thisModule.CreateConfProperty( beanEl, 'outgoing', el.outgoing );
    }
}
```

```
rule ForkNodeOrJoinNode2Conf extends ActivityNode2Conf
{
    from el : UWE!ControlNode ( el.oclIsTypeOf( UWE!ForkNode ) or
        el.oclIsTypeOf( UWE!JoinNode ) )
    to classAttr : XML!Attribute
    (
        name <- 'class',
        value <- thisModule.processPackageName + '.ForkJoinNode',
        parent <- beanEl
    )
    do
    {
        thisModule.CreateConfProperty( beanEl, 'name', el.name );
        thisModule.CreateConfProperty( beanEl, 'activity', el.activity );
        thisModule.CreateConfProperty( beanEl, 'incoming', el.incoming );
        thisModule.CreateConfProperty( beanEl, 'outgoing', el.outgoing );
    }
}

rule CallOperationAction2Conf extends ActivityNode2Conf
{
    from el : UWE!CallOperationAction
    do
    {
        thisModule.CreateConfProperty( beanEl, 'name', el.name );
        thisModule.CreateConfProperty( beanEl, 'methodName', el.operation.name );
        thisModule.CreateConfProperty( beanEl, 'activity', el.activity );
        thisModule.CreateConfProperty( beanEl, 'incoming', el.incoming );
        thisModule.CreateConfProperty( beanEl, 'outgoing', el.outgoing );
        thisModule.CreateConfProperty( beanEl, 'target', el.target );
        thisModule.CreateConfProperty( beanEl, 'input', el.input );
        thisModule.CreateConfProperty( beanEl, 'output', el.output );
    }
}

rule UserAction2Conf extends ActivityNode2Conf
{
    from el : UWE!UserAction
    to classAttr : XML!Attribute
    (
        name <- 'class',
        value <- thisModule.processPackageName + '.UserAction',
        parent <- beanEl
    )
    do
```

```
    {
        thisModule.CreateConfProperty( beanEl, 'name', el.name );
        thisModule.CreateConfProperty( beanEl, 'activity', el.activity );
        thisModule.CreateConfProperty( beanEl, 'processClass', el.processClass.qualifiedId() );
        thisModule.CreateConfProperty( beanEl, 'incoming', el.incoming );
        thisModule.CreateConfProperty( beanEl, 'outgoing', el.outgoing );
        thisModule.CreateConfProperty( beanEl, 'input', el.input );
        thisModule.CreateConfProperty( beanEl, 'output', el.output );
    }
}

rule ActivityEdge2Conf extends NamedElement2Conf
{
    from el : UWE!ActivityEdge
    to classAttr : XML!Attribute
    (
        name <- 'class',
        value <- thisModule.processPackageName + '.' + el.strippedTypeName(),
        parent <- beanEl
    )
    do
    {
        if( el.guard.oclIsKindOf( UWE!OpaqueExpression ) )
        {
            thisModule.CreateConfProperty( beanEl, 'guard', el.guard.body );
        }
        thisModule.CreateConfProperty( beanEl, 'name', el.name );
        thisModule.CreateConfProperty( beanEl, 'activity', el.activity );
        thisModule.CreateConfProperty( beanEl, 'source', el.source );
        thisModule.CreateConfProperty( beanEl, 'target', el.target );
    }
}
```

B.4.6 Transformation Presentation2JSP

```
module Presentation2JSP;
create OUT : JSP from IN : UWE;

uses strings;
uses UWEHelpers;

helper context String def : formatTypeName() : String =
    self.regexReplaceAll( '([a-z])([A-Z])', '$1 $2' );

helper def : translateELExpr( expr : String, prefixExpr : String ) : String =
```

```
    expr.regexReplaceAll( '[{]', '{' + prefixExpr + '.' );

helper def : createJSTLCOutExpr( expr : String ) : String =
    '<c:out value=\'${' + expr + '}\' />';

helper def : createRawJSTLCOutExpr( expr : String ) : String =
    '<c:out value=\'' + expr + '\' />';

helper def : createJSTLURLExpr( viewName : String, objIDExpr : String ) : String =
    '<c:url value=\'' + viewName + '.uwe\'><c:param name=\'objID\' value=\'${' +
    objIDExpr + '}\' /></c:url>';

helper context UWE!Class def : elPath() : Sequence( String ) =
    self.containingPropertyPath()->iterate( pp; res : Sequence( String ) = Sequence {} |
        if pp.navigationProperty.oclIsKindOf( UWE!NavigationProperty ) then
            if pp.navigationProperty.contentProperties->size() = 1 then
                let pname : String = pp.navigationProperty.contentProperties->first().name in
                    if pname.oclIsUndefined() or pname = '' then res else res->including( pname ) endif
            else res endif
        else res endif
    );

helper context UWE!Class def : elExpression() : String =
    self.elPath()->prepend( 'self' )->iterate( s; res : String = '' |
        if res = '' then s else res + '.' + s endif );

helper context UWE!Class def : elExpressionIt() : String =
    self.elPath()->prepend( 'self' )->iterate( s; res : String = '' |
        if res = '' then s else res + '_' + s endif ) + '_it';

-- RULES

lazy rule RootPresentationClass2JSP
{
    from pc : UWE!PresentationClass
    to bodyNode : JSP!Element
    (
        name <- 'body',
        children <- Sequence { jspIncludeDirective3 }
    ),
    jsp : JSP!Root
    (
        children <- Sequence { jspPageLanguageDirective, jspIncludeDirective1, htmlNode },
        documentName <- pc.node.qualifiedId() + '.jsp'
    ),
    jspPageLanguageDirective : JSP!JSPDirective
```

```
    (
        name <- 'page',
        value <- 'language="java"'
    ),
    jspIncludeDirective1 : JSP!JSPDirective
    (
        name <- 'include',
        value <- 'file="/WEB-INF/jsp/include.jspf"'
    ),
    htmlNode : JSP!Element
    (
        name <- 'html',
        children <- Sequence { headNode, jspIncludeDirective2, bodyNode }
    ),
    headNode : JSP!Element
    (
        name <- 'head',
        children <- Sequence { titleNode }
    ),
    titleNode : JSP!Element
    (
        name <- 'title',
        children <- Sequence { titleTextNode }
    ),
    titleTextNode : JSP!TextNode
    (
        value <- pc.name.formatTypeName()
    ),
    jspIncludeDirective2 : JSP!JSPDirective
    (
        name <- 'include',
        value <- 'file="/WEB-INF/jsp/style.jspf"'
    ),
    jspIncludeDirective3 : JSP!JSPDirective
    (
        name <- 'include',
        value <- 'file="/WEB-INF/jsp/header.jspf"'
    )
}

rule PresentationClass2JSP
{
    from pc : UWE!PresentationClass
    to pcBody : JSP!Element
    (
        name <- 'div',
```

```
        children <- Sequence { cssClassAttr, cssStyleAttr, captionNode, pc.ownedAttribute },
        parent <- if pc.containingClass().oclIsUndefined() then
            thisModule.RootPresentationClass2JSP( pc ) else OclUndefined endif
    ),
    cssClassAttr : JSP!Attribute
    (
        name <- 'class',
        value <- if pc.cssClass.oclIsUndefined() then '' else pc.cssClass endif
    ),
    cssStyleAttr : JSP!Attribute
    (
        name <- 'style',
        value <- if pc.cssStyle.oclIsUndefined() then '' else pc.cssStyle endif
    ),
    captionNode : JSP!Element
    (
        name <- 'h' + ( pc.containingPropertyPath()->size() + 2 ).toString(),
        children <- Sequence { captionTextNode }
    ),
    captionTextNode : JSP!TextNode
    (
        value <- pc.name.formatTypeName()
    )
}

rule PresentationClassForNavigationClass2JSP extends PresentationClass2JSP
{
    from pc : UWE!PresentationClass ( pc.node.oclIsKindOf( UWE!NavigationClass ) )
    using
    {
        staticProperties : Sequence( UWE!PresentationProperty ) = pc.ownedAttribute->
            select( pp | pp.type.oclIsKindOf( UWE!StaticElement ) );
        attributeProperties : Sequence( UWE!PresentationProperty ) = pc.ownedAttribute->
            select( pp | pp.type.oclIsKindOf( UWE!OutputElement ) );
        anchorProperties : Sequence( UWE!PresentationProperty ) = pc.ownedAttribute->
            select( pp | pp.type.oclIsKindOf( UWE!Anchor ) );
        pcProperties : Sequence( UWE!PresentationProperty ) = pc.ownedAttribute->
            select( pp | pp.type.oclIsKindOf( UWE!PresentationClass ) );
    }
    to pcBody : JSP!Element
    (
        children <- Sequence { captionNode, staticProperties, tableNode, anchorDivNodes,
            pcProperties }->flatten()
    ),
    tableNode : JSP!Element
    (
```

```
        name <- 'table',
        children <- trNodes
    ),
    trNodes : distinct JSP!Element foreach ( p in attributeProperties )
    (
        name <- 'tr'
    ),
    col1Nodes : distinct JSP!Element foreach ( p in attributeProperties )
    (
        name <- 'td',
        parent <- trNodes,
        children <- labelNodes
    ),
    labelNodes : distinct JSP!TextNode foreach ( p in attributeProperties )
    (
        value <- p.type.name.formatTypeName() + ':'
    ),
    col2Nodes : distinct JSP!Element foreach ( p in attributeProperties )
    (
        name <- 'td',
        parent <- trNodes,
        children <- p
    ),
    anchorDivNodes : distinct JSP!Element foreach ( p in anchorProperties )
    (
        name <- 'div',
        children <- p
    )
}

rule PresentationClassForProcessClass2JSP extends PresentationClass2JSP
{
    from pc : UWE!PresentationClass ( pc.node.oclIsTypeOf( UWE!ProcessClass ) )
    using
    {
        staticProperties : Sequence( UWE!PresentationProperty ) = pc.ownedAttribute->
            select( pp | pp.type.oclIsKindOf( UWE!StaticElement ) );
        processProperties : Sequence( UWE!PresentationProperty ) = pc.ownedAttribute->
            select( pp | pp.navigationProperty.oclIsKindOf( UWE!ProcessProperty ) );
    }
    to pcBody : JSP!Element
    (
        children <- Sequence { captionNode, subCaptionNode, staticProperties, formNode }->
            flatten()
    ),
    captionNode : JSP!Element
```

```
(
    name <- 'h2',
    children <- Sequence { captionTextNode }
),
captionTextNode : JSP!TextNode
(
    value <- let c : UWE!Class = pc.node.containingClass() in
        if c.oclIsUndefined() then '' else c.name.formatTypeName() endif
),
subCaptionNode : JSP!Element
(
    name <- 'h3',
    children <- Sequence { subCaptionTextNode }
),
subCaptionTextNode : JSP!TextNode
(
    value <- pc.name.formatTypeName()
),
formNode : JSP!Element
(
    name <- 'form',
    children <- Sequence { actionAttr, methodAttr, tableNode, pNode, sbNode, rbNode }
),
actionAttr : JSP!Attribute
(
    name <- 'action',
    value <- '__processinput__.uwe'
),
methodAttr : JSP!Attribute
(
    name <- 'method',
    value <- 'post'
),
tableNode : JSP!Element
(
    name <- 'table',
    children <- trNodes
),
trNodes : distinct JSP!Element foreach ( p in processProperties )
(
    name <- 'tr'
),
col1Nodes : distinct JSP!Element foreach ( p in processProperties )
(
    name <- 'td',
    parent <- trNodes,
```

```
    children <- labelNodes
),
labelNodes : distinct JSP!TextNode foreach ( p in processProperties )
(
    value <- p.type.name.formatTypeName() + ':'
),
col2Nodes : distinct JSP!Element foreach ( p in processProperties )
(
    name <- 'td',
    parent <- trNodes,
    children <- p
),
pNode : JSP!Element
(
    name <- 'p'
),
sbNode : JSP!Element
(
    name <- 'input',
    children <- Sequence { sbTypeAttr, sbValueAttr }
),
sbTypeAttr : JSP!Attribute
(
    name <- 'type',
    value <- 'submit'
),
sbValueAttr : JSP!Attribute
(
    name <- 'value',
    value <- 'Submit'
),
rbNode : JSP!Element
(
    name <- 'input',
    children <- Sequence { rbTypeAttr, rbValueAttr }
),
rbTypeAttr : JSP!Attribute
(
    name <- 'type',
    value <- 'reset'
),
rbValueAttr : JSP!Attribute
(
    name <- 'value',
    value <- 'Reset'
)
```

```
}

rule PresentationClassForIndex2JSP extends PresentationClass2JSP
{
    from pc : UWE!PresentationClass ( pc.node.oclIsTypeOf( UWE!Index ) )
    using
    {
        anchorProperty : Sequence( UWE!PresentationProperty ) = pc.ownedAttribute->
            select( pp | pp.type.oclIsKindOf( UWE!Anchor ) )->first();
        elExpression : String = pc.elExpression();
        elExpressionIt : String = pc.elExpressionIt();
        target : UWE!NavigationNode = let outLinks : Set( UWE!Link ) = pc.node.outLinks() in
            if outLinks->size() <> 1 then OclUndefined else
                outLinks->asSequence()->first().target()
            endif;
    }
    to pcBody : JSP!Element
    (
        children <- Sequence { captionNode, ulNode }
    ),
    ulNode : JSP!Element
    (
        name <- 'ul',
        children <- Sequence { forEachNode }
    ),
    forEachNode : JSP!Element
    (
        name <- 'c:forEach',
        children <- Sequence { itemsAttr, varAttr, liNode }
    ),
    itemsAttr : JSP!Attribute
    (
        name <- 'items',
        value <- '${' + elExpression + '}'
    ),
    varAttr : JSP!Attribute
    (
        name <- 'var',
        value <- elExpressionIt
    ),
    liNode : JSP!Element
    (
        name <- 'li',
        children <- Sequence { anchorProperty }
    )
}
```

```
rule PresentationProperty2JSP
{
    from pp : UWE!PresentationProperty ( not pp.class_.oclIsUndefined() )
    to spanNode : JSP!Element
    (
        name <- 'span',
        children <- Sequence { pp.type }
    )
}

rule UIElement2JSP
{
    from ui : UWE!UIElement
    to uiBody : JSP!Element
    (
        name <- 'span',
        children <- Sequence { cssClassAttr, cssStyleAttr }
    ),
    cssClassAttr : JSP!Attribute
    (
        name <- 'class',
        value <- if ui.cssClass.oclIsUndefined() then '' else ui.cssClass endif
    ),
    cssStyleAttr : JSP!Attribute
    (
        name <- 'style',
        value <- if ui.cssStyle.oclIsUndefined() then '' else ui.cssStyle endif
    )
}

rule Anchor2JSP extends UIElement2JSP
{
    from ui : UWE!Anchor
    using
    {
        presentationProperty : UWE!PresentationProperty = ui.containingProperty();
        navigationProperty : UWE!NavigationProperty = if presentationProperty.
            navigationProperty.oclIsTypeOf( UWE!NavigationProperty ) then
            presentationProperty.navigationProperty else OclUndefined endif;
        contentProperty : UWE!Property = if navigationProperty.oclIsUndefined() then OclUndefined
            else if navigationProperty.contentProperties->isEmpty() then OclUndefined else
                navigationProperty.contentProperties->first() endif
            endif;
        link : UWE!Link = if presentationProperty.class_.node.
            oclIsKindOf( UWE!AccessPrimitive ) then
```

```
            presentationProperty.class_.node.outLinks()->first()
        else if navigationProperty.oclIsUndefined() then OclUndefined else
            navigationProperty.association endif endif;
    target : UWE!NavigationNode = if link.oclIsUndefined() then OclUndefined
        else link.target() endif;
    elExpression : String = if presentationProperty.class_.node.
        oclIsKindOf( UWE!AccessPrimitive ) then
        ui.elExpressionIt() else ui.elExpression() endif;
}
to uiBody : JSP!Element
(
    children <- Sequence { cssClassAttr, cssStyleAttr, cIfNode }
),
cIfNode : JSP!Element
(
    name <- 'c:if',
    children <- Sequence { cIfTestAttr, cSetNode, aNode }
),
cIfTestAttr : JSP!Attribute
(
    name <- 'test',
    value <- '${not empty ' + elExpression + if link.oclIsUndefined() then '' else
            if link.guard.oclIsUndefined() then '' else ' and ( ' + link.guard + ' )' endif
        endif + '}'
),
cSetNode : JSP!Element
(
    name <- 'c:set',
    children <- Sequence { cSetVarAttr, cSetScopeAttr, cSetValueAttr }
),
cSetVarAttr : JSP!Attribute
(
    name <- 'var',
    value <- 'obj'
),
cSetScopeAttr : JSP!Attribute
(
    name <- 'scope',
    value <- 'request'
),
cSetValueAttr : JSP!Attribute
(
    name <- 'value',
    value <- '${' + elExpression + '}'
),
aNode : JSP!Element
```

```
    (
        name <- 'a',
        children <- Sequence { hrefAttr, anchorTextNode }
    ),
    hrefAttr : JSP!Attribute
    (
        name <- 'href',
        value <- if target.oclIsUndefined() then '' else
                target.qualifiedId() + '.uwe?objID=<%= objID( request ) %>'
            endif
    ),
    anchorTextNode : JSP!TextNode
    (
        value <- if ui.format.oclIsUndefined() or ui.format = '' then ui.name.formatTypeName() else
                thisModule.createRawJSTLCOutExpr( thisModule.translateELExpr(
                    ui.format, elExpression ) )
            endif
    )
}

rule Text2JSP extends UIElement2JSP
{
    from ui : UWE!Text
    using
    {
        presentationProperty : UWE!PresentationProperty = ui.containingProperty();
        navigationProperty : UWE!NavigationProperty = if presentationProperty.navigationProperty.
            oclIsTypeOf( UWE!NavigationProperty ) then
            presentationProperty.navigationProperty else OclUndefined endif;
        contentProperty : UWE!Property = if navigationProperty.oclIsUndefined() then OclUndefined
            else if navigationProperty.contentProperties->isEmpty() then OclUndefined else
                navigationProperty.contentProperties->first() endif
            endif;
        elExpression : String = if contentProperty.oclIsUndefined() then '' else
                if presentationProperty.class_.node.oclIsKindOf( UWE!AccessPrimitive )
                    then 'self_it' else 'self' endif +
                    '.' + contentProperty.name
            endif;
    }
    to uiBody : JSP!Element
    (
        children <- Sequence { cssClassAttr, cssStyleAttr, cOutEl }
    ),
    cOutEl : JSP!Element
    (
        name <- 'c:out',
```

```
        children <- Sequence { valueAttr }
    ),
    valueAttr : JSP!Attribute
    (
        name <- 'value',
        value <- '${' + elExpression + '}'
    )
}

rule Image2JSP extends UIElement2JSP
{
    from ui : UWE!Image
    using
    {
        presentationProperty : UWE!PresentationProperty = ui.containingProperty();
        navigationProperty : UWE!NavigationProperty = if presentationProperty.navigationProperty.
            oclIsTypeOf( UWE!NavigationProperty ) then
            presentationProperty.navigationProperty else OclUndefined endif;
        contentProperty : UWE!Property = if navigationProperty.oclIsUndefined() then OclUndefined
            else if navigationProperty.contentProperties->isEmpty() then OclUndefined else
                navigationProperty.contentProperties->first() endif
            endif;
        elExpression : String = if contentProperty.oclIsUndefined() then '' else
            if presentationProperty.class_.node.oclIsKindOf( UWE!AccessPrimitive ) then 'self_it'
            else 'self' endif + '.' + contentProperty.name endif;
    }
    to uiBody : JSP!Element
    (
        name <- 'img',
        children <- Sequence { cssClassAttr, cssStyleAttr, srcAttr }
    ),
    srcAttr : JSP!Attribute
    (
        name <- 'src',
        value <- if ui.url.oclIsUndefined() then '' else ui.url + '/' endif +
            thisModule.createRawJSTLCOutExpr( '${' + elExpression + '}' )
    )
}

rule TextInput2JSP extends UIElement2JSP
{
    from ui : UWE!TextInput
    using
    {
        processProperty : UWE!ProcessProperty = let p : UWE!Property =
            ui.containingProperty().navigationProperty in
```

```
            if p.oclIsTypeOf( UWE!ProcessProperty ) then p else OclUndefined endif;
        editProperty : UWE!Property = if processProperty.oclIsUndefined() then OclUndefined
            else processProperty.editProperty endif;
    }
    to uiBody : JSP!Element
    (
        name <- 'input',
        children <- Sequence { cssClassAttr, cssStyleAttr, typeAttr, nameAttr, valueAttr }
    ),
    typeAttr : JSP!Attribute
    (
        name <- 'type',
        value <- 'text'
    ),
    nameAttr : JSP!Attribute
    (
        name <- 'name',
        value <- '__' + processProperty.name
    ),
    valueAttr : JSP!Attribute
    (
        name <- 'value',
        value <- if editProperty.oclIsUndefined() then '' else
                thisModule.createJSTLCOutExpr( 'self.' + editProperty.name )
            endif
    )
}

rule EnumerationInput2JSP extends UIElement2JSP
{
    from ui : UWE!EnumerationInput
    using
    {
        processProperty : UWE!ProcessProperty = let p : UWE!Property =
            ui.containingProperty().navigationProperty in
            if p.oclIsTypeOf( UWE!ProcessProperty ) then p else OclUndefined endif;
        enumLiterals : Sequence( UWE!EnumerationLiteral ) = let p : UWE!Property =
            if  processProperty.editProperty.oclIsUndefined() then processProperty else
            processProperty.editProperty endif in
                if p.type.oclIsKindOf( UWE!Enumeration ) then p.type.ownedLiteral else Sequence {}
            endif;
    }
    to uiBody : JSP!Element
    (
        name <- 'select',
        children <- Sequence { cssClassAttr, cssStyleAttr, nameAttr, optionNodes }
```

```
    ),
    nameAttr : JSP!Attribute
    (
        name <- 'name',
        value <- '__' + processProperty.name
    ),
    optionNodes : distinct JSP!Element foreach ( el in enumLiterals )
    (
        name <- 'option',
        children <- optionValues->iterate( e; res : Sequence( OclAny ) = Sequence { Sequence {
            optionValues->at( enumLiterals->indexOf( el ) ), optionTextValues->
            at( enumLiterals->indexOf( el ) ) } } |
                if optionValues->indexOf( e ) < enumLiterals->indexOf( el )
                    then res.prepend( Sequence {} ) else res endif )
    ),
    optionValues : distinct JSP!Attribute foreach ( el in enumLiterals )
    (
        name <- 'value',
        value <- el.name
    ),
    optionTextValues : distinct JSP!TextNode foreach ( el in enumLiterals )
    (
        value <- el.name.firstToUpper().formatTypeName()
    )
}

rule Selection2JSP extends UIElement2JSP
{
    from ui : UWE!Selection
    using
    {
        processProperty : UWE!ProcessProperty = let p : UWE!Property =
            ui.containingProperty().navigationProperty in
            if p.oclIsTypeOf( UWE!ProcessProperty ) then p else OclUndefined endif;
        elExpression : String = 'self.' + if processProperty.rangeExpression.oclIsUndefined() then
            processProperty.name else processProperty.rangeExpression endif;
        elExpressionIt : String = 'self_it';
    }
    to uiBody : JSP!Element
    (
        name <- 'select',
        children <- Sequence { cssClassAttr, cssStyleAttr, nameAttr }->union(
            if processProperty.lower = 0 then Sequence {
                thisModule.CreateNoneOptionNode( false ), forEachNode }
            else Sequence { forEachNode } endif )
    ),
```

```
nameAttr : JSP!Attribute
(
    name <- 'name',
    value <- '__' + processProperty.name
),
forEachNode : JSP!Element
(
    name <- 'c:forEach',
    children <- Sequence { itemsAttr, varAttr, cSetNode, chooseNode }
),
itemsAttr : JSP!Attribute
(
    name <- 'items',
    value <- '${' + elExpression + '}'
),
varAttr : JSP!Attribute
(
    name <- 'var',
    value <- elExpressionIt
),
cSetNode : JSP!Element
(
    name <- 'c:set',
    children <- Sequence { cSetVarAttr, cSetScopeAttr, cSetValueAttr }
),
cSetVarAttr : JSP!Attribute
(
    name <- 'var',
    value <- 'obj'
),
cSetScopeAttr : JSP!Attribute
(
    name <- 'scope',
    value <- 'request'
),
cSetValueAttr : JSP!Attribute
(
    name <- 'value',
    value <- '${' + elExpressionIt + '}'
),
chooseNode : JSP!Element
(
    name <- 'c:choose',
    children <- Sequence { whenNode, otherwiseNode }
),
whenNode : JSP!Element
```

```
    (
        name <- 'c:when',
        children <- Sequence { whenTestAttr,
            thisModule.CreateOptionNode( true, thisModule.createRawJSTLCOutExpr(
            thisModule.translateELExpr( ui.format, elExpressionIt ) ) ) }
    ),
    whenTestAttr : JSP!Attribute
    (
        name <- 'test',
        value <- '${' +
                if processProperty.editProperty.oclIsUndefined() then 'false' else
                elExpressionIt + ' == self.' + processProperty.editProperty.name endif +
            '}'
    ),
    otherwiseNode : JSP!Element
    (
        name <- 'c:otherwise',
        children <- Sequence {
            thisModule.CreateOptionNode( false, thisModule.createRawJSTLCOutExpr(
            thisModule.translateELExpr( ui.format, elExpressionIt ) ) ) }
    )
}

rule CreateOptionNode( selected : Boolean, text : String )
{
    to optionNode : JSP!Element
    (
        name <- 'option',
        children <- Sequence { optionValueAttr, optionTextNode }
    ),
    optionValueAttr : JSP!Attribute
    (
        name <- 'value',
        value <- '$<%= objID( request ) %>'
    ),
    optionTextNode : JSP!TextNode
    (
        value <- text
    )
    do
    {
        if( selected )
        {
            thisModule.CreateSelectedAttribute().parent <- optionNode;
        }
        optionNode;
```

```
    }
}

rule CreateNoneOptionNode( selected : Boolean )
{
    to optionNode : JSP!Element
    (
        name <- 'option',
        children <- Sequence { optionValueAttr, optionTextNode }
    ),
    optionValueAttr : JSP!Attribute
    (
        name <- 'value',
        value <- '$null'
    ),
    optionTextNode : JSP!TextNode
    (
        value <- '--- none ---'
    )
    do
    {
        if( selected )
        {
            thisModule.CreateSelectedAttribute().parent <- optionNode;
        }
        optionNode;
    }
}

rule CreateSelectedAttribute()
{
    to selAttr : JSP!Attribute
    (
        name <- 'selected',
        value <- 'selected'
    )
    do
    {
        selAttr;
    }
}

rule StaticText2JSP extends UIElement2JSP
{
    from ui : UWE!StaticText
    to uiBody : JSP!Element
```

```
    (
        name <- 'p',
        children <- Sequence { cssClassAttr, cssStyleAttr, textNode }
    ),
    textNode : JSP!TextNode
    (
        value <- ui.text
    )
}

rule StaticImage2JSP extends UIElement2JSP
{
    from ui : UWE!StaticImage
    to uiBody : JSP!Element
    (
        name <- 'img',
        children <- Sequence { cssClassAttr, cssStyleAttr, srcAttr }
    ),
    srcAttr : JSP!Attribute
    (
        name <- 'src',
        value <- ui.url
    )
}
```

Curriculum Vitae

Andreas Kraus

geboren am 29. Januar 1974 in Starnberg

seit 02.2001	Promotion am Lehrstuhl für Programmierung und Softwaretechnik des Departments für Informatik an der Ludwigs-Maximilians-Universität München
	Thema der Doktorarbeit: Model Driven Software Engineering for Web Applications
01.2000	Abschluß des Physikstudiums mit Note sehr gut
	Thema der Diplomarbeit: Getriebene nichtlineare nanomechanische Resonatoren
11.1995	Vordiplom in Informatik
11.1995	Beginn des Informatikstudiums (Diplom) an der Ludwigs-Maximilians-Universität München
05.1995	Vordiplom in Physik
11.1993	Beginn des Physikstudiums (Diplom) an der Ludwigs-Maximilians-Universität München
07.1993	Abitur am Kurt-Huber-Gymnasium in Gräfelfing