



FAKULTÄT FÜR **INFORMATIK**

# Visualization and Manipulation of Diagrams on The Web

—

## Developing e-Learning Support for Teaching UML in The Large

MAGISTERARBEIT

zur Erlangung des akademischen Grades

**Magister der Sozial- und Wirtschaftswissenschaften**

im Rahmen des Studiums

**Informatikmanagement**

eingereicht von

**Johannes Murth**

Matrikelnummer 0427416

an der

Fakultät für Informatik der Technischen Universität Wien

Betreuung:

Betreuer/Betreuerin: O.Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel

Mitwirkung: Univ.-Ass. Dr. Manuel Wimmer

Wien, 07.12.2009

\_\_\_\_\_  
(Unterschrift Verfasser/in)

\_\_\_\_\_  
(Unterschrift Betreuer/in)



# Erklärung zur Verfassung der Arbeit

Johannes Murth, Siebenbrunnengasse 8/12, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit einschließlich Tabellen, Karten und Abbildungen, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 07. Dezember 2009

---

Johannes Murth



# Danksagung

Ich bedanke mich für die Betreuung durch Prof. Dr. Gerti Kappel und Univ.Ass. Dr. Manuel Wimmer, die mir stets freundlich und mit großer Kompetenz zur Seite gestanden sind. Herzlicher Dank gilt meinen Eltern Walter und Elfi und der gesamten Familie, die mich auf meinem Weg immer unterstützt haben — im Speziellen meinem Bruder Martin, der mir durch seine fachkundige Hilfe viele gute Impulse geben konnte.



# Abstract

The ability of software engineers to develop clear and comprehensive models according to a specific problem domain has become an essential factor for the success of software projects. The trend of Model Driven Software Engineering (MDSE) even increases this factor. Universities have to qualitatively educate their students in this field to prepare them for those challenges. At the Vienna University of Technology (VUT), the course “Object Oriented Modeling” teaches the basics of modeling and the Unified Modeling Language (UML). Due to the high number of students, various e-Learning elements have already been established, mainly for teaching the theory behind UML.

The goal of this thesis was to develop a web-based modeling tool, that is integrated into the e-Learning platform of the VUT using Rich Internet Application technologies. Thus, students can solve practical modeling exercises comfortably and directly using the e-Learning platform. Therefore, a generic approach shall be used by utilizing meta-modeling: Visualization and manipulation of two-dimensional diagrams is based on certain patterns. A framework was created, that allows for creating complete diagram editors with drag-and-drop functionality, by specifying the structure of the diagram elements (abstract syntax) and the visual notation elements (concrete syntax). This enables to easily create various UML editors and to align the notation elements with those of the course. Within this thesis, three editors (class diagram, state diagram, and sequence diagram) have been created with the aid of the framework and integrated into the e-Learning platform.



# Kurzfassung

Die Fähigkeit von Software-Entwicklern, zu konkreten Problemstellungen klare und umfassende Modelle zu erstellen, erweist sich als essenzieller Faktor für den Erfolg von Software-Projekten. Die neuen Entwicklungen im Bereich von modellgetriebener Software-Entwicklung verstärken diesen Faktor darüber hinaus. Universitäten haben die Aufgabe, ihre Studenten in diesem Bereich auf hohem Niveau auszubilden, um sie auf diese Herausforderungen vorzubereiten. An der Technischen Universität Wien werden die Grundlagen von Modellierung und der Unified Modeling Language (UML) im Kurs “Objektorientierte Modellierung” gelehrt. Aufgrund der hohen Studentenzahl wurden bereits zahlreiche e-Learning Elemente eingeführt, vor allem im Bereich der theoretischen Grundlagen.

Ziel dieser Arbeit ist es, mithilfe von Rich Internet Application-Technologien ein web-basiertes Modellierungstool zu erstellen, das in die Lernplattform der TU Wien integriert werden kann. Mit diesem Tool können Studenten praktische Modellierungsaufgaben einfach und direkt auf der Lernplattform lösen. Dabei soll ein generischer, Metamodell-basierter Ansatz gewählt werden: Die Visualisierung und Manipulierung von 2D-Diagrammen basiert auf bestimmten Mustern. Ein Framework soll es ermöglichen, nur durch die Angabe der Struktur der Diagrammelemente (abstrakte Syntax) und der visuellen Notationselemente (konkrete Syntax) einen kompletten Editor mit Drag-and-drop-Funktionalität zu erstellen. Damit wird es ermöglicht, verschiedene UML-Editoren zu erstellen und die Notationselemente mit denen des Kurses abzustimmen. Im Zuge der Arbeit werden prototypisch drei UML-Editoren (Klassendiagramm, Zustandsdiagramm, Sequenzdiagramm) mithilfe des Frameworks erstellt und in die Lernplattform integriert.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Problem Description . . . . .	1
1.3	Approach . . . . .	2
1.4	Structure . . . . .	2
<b>2</b>	<b>Basic Concepts</b>	<b>5</b>
2.1	Metamodeling . . . . .	5
2.1.1	Core Concepts . . . . .	5
2.1.2	Modeling Languages . . . . .	8
2.1.3	Eclipse Modeling Project (EMP) . . . . .	9
2.2	E-Learning . . . . .	11
2.2.1	Learning Management Systems . . . . .	11
2.2.2	Object-Oriented Modeling Course at the VUT . . . . .	12
2.2.3	Perspectives of a Course Centered Around a Modeling Tool . . . . .	14
2.3	Rich Internet Applications . . . . .	15
2.3.1	Basics . . . . .	15
2.3.2	RIA Technologies . . . . .	16
2.3.3	Evaluation and Choice . . . . .	20
<b>3</b>	<b>Developing the Web Modeling Framework</b>	<b>25</b>
3.1	Requirements . . . . .	25
3.1.1	Visualization and Manipulation of Diagrams . . . . .	26
3.1.2	Specification of Diagram Types . . . . .	28
3.1.3	Import and Export Interfaces . . . . .	28
3.2	Implementation Technology: Adobe Flex . . . . .	28
3.2.1	Visual Components . . . . .	29
3.2.2	Model-View-Controller Pattern . . . . .	29
3.3	Definition of Diagram Types . . . . .	32
3.3.1	Stencil Set DSL . . . . .	32
3.3.2	Stencil View DSL . . . . .	35
3.4	Generic Diagram Editing . . . . .	38
3.4.1	Common Data Structure for Arranging Diagram Elements . . . . .	38
3.4.2	Visual UI Components for Visualization and Manipulation . . . . .	39
3.5	Framework Utilization . . . . .	41
3.5.1	Design of a Typical WebMF Application . . . . .	42
3.5.2	Interplay of the Functionality Parts . . . . .	42
<b>4</b>	<b>Sample Application</b>	<b>45</b>
4.1	Abstract Syntax . . . . .	45
4.2	Concrete Syntax . . . . .	46
4.3	Building the Stencil Set Library . . . . .	46
4.3.1	Creating an Flex Library Project . . . . .	46

4.3.2	Creating a Stencil Set . . . . .	48
4.3.3	Creating Stencil Views . . . . .	51
4.4	Building an Editor Application . . . . .	54
4.4.1	Creating an Flex Application . . . . .	54
4.4.2	Creating and Connecting Components . . . . .	54
4.4.3	Appearance of the Created Application . . . . .	55
<b>5</b>	<b>Developing e-Learning Support with WebMF</b>	<b>57</b>
5.1	UML Class Diagram Editor . . . . .	57
5.1.1	Abstract Syntax . . . . .	57
5.1.2	Concrete Syntax . . . . .	57
5.1.3	Implementation . . . . .	59
5.1.4	Issues . . . . .	61
5.2	UML State Diagram Editor . . . . .	63
5.2.1	Abstract Syntax . . . . .	63
5.2.2	Concrete Syntax . . . . .	63
5.2.3	Implementation . . . . .	63
5.2.4	Issues . . . . .	67
5.3	UML Sequence Diagram Editor . . . . .	68
5.3.1	Abstract Syntax . . . . .	68
5.3.2	Concrete Syntax . . . . .	68
5.3.3	Implementation . . . . .	69
5.3.4	Issues . . . . .	71
5.4	Integration of Editors Into Moodle . . . . .	73
5.4.1	Enabling Import and Export Functions . . . . .	74
5.4.2	Developing the WebMF Assignment Plug-in for Moodle . . . . .	74
5.4.3	Creating a Sample Assignment . . . . .	76
<b>6</b>	<b>Related Work</b>	<b>81</b>
6.1	Oryx Editor . . . . .	81
6.1.1	Metamodel . . . . .	81
6.1.2	Technologies . . . . .	82
6.1.3	Communication and Architecture . . . . .	82
6.1.4	Usability . . . . .	83
6.1.5	Rating . . . . .	85
6.2	SLIM . . . . .	85
6.2.1	Metamodel . . . . .	85
6.2.2	Technologies . . . . .	85
6.2.3	Communication and Architecture . . . . .	86
6.2.4	Usability . . . . .	86
6.2.5	Rating . . . . .	88
6.3	Web 2.0 Metamodel Browser . . . . .	88
6.3.1	Metamodel . . . . .	88
6.3.2	Technologies . . . . .	88
6.3.3	Communication and Architecture . . . . .	88
6.3.4	Usability . . . . .	89
6.3.5	Rating . . . . .	89
6.4	2D Meta Model Browser . . . . .	89
6.4.1	Metamodel . . . . .	89
6.4.2	Technologies . . . . .	90
6.4.3	Communication and Architecture . . . . .	91

6.4.4	Usability . . . . .	91
6.4.5	Rating . . . . .	91
6.5	Comparison . . . . .	91
<b>7</b>	<b>Evaluation</b>	<b>93</b>
7.1	Implementation Effort for WebMF Applications . . . . .	93
7.2	Evaluation of the Requirements Catalog . . . . .	94
7.2.1	Diagram . . . . .	94
7.2.2	Tool bar . . . . .	95
7.2.3	Properties Editor . . . . .	96
7.2.4	Import / Export . . . . .	96
7.3	Lessons Learned . . . . .	96
7.3.1	Recommendations for Enhancement . . . . .	96
7.3.2	Limitations . . . . .	97
7.4	Summary . . . . .	97
<b>8</b>	<b>Conclusion</b>	<b>99</b>
8.1	Summary . . . . .	99
8.2	Future Work . . . . .	99
	<b>Bibliography</b>	<b>101</b>

# 1 Introduction

This chapter gives an overview on the thesis. First, Section 1.1 explains the thesis' context. The problem description is stated in section 1.2 which is tackled by the approach presented in Section 1.3. Finally, the thesis' structure is outlined in Section 1.4.

## 1.1 Motivation

Object Oriented Modeling (OOM) is one of the most powerful tools in modern software engineering. Its de-facto standard, the Unified Modeling Language (UML) [22] is an expressive graphical language for specifying all artifacts used and produced in the software development process. It facilitates Software specification, Software construction, Software visualization, and Software documentation [38].

The trends to Model Engineering (ME), including Model Driven Engineering (MDE) and Model Driven Architecture (MDA), have even increased the importance of OOM and UML. Commonly, they see modeling as primary approach in the software development process and aim at an automation of the processes themselves [17] [6].

As a result, the ability of software engineers to develop clear and comprehensive models according to a specific problem domain has become an essential factor for the success of software projects. To cope with these challenges, engineers need to be familiar with OOM and UML. They have to formulate parts of the problem domain and their relationships in semiformal natural language and produce models on the basis of these formulations. They also have to know relevant design patterns and use state of the art visual model editors for practical modeling tasks. Further, they need to implement the modeled artifacts and use modeling as a central approach for realizing software projects.

To keep up-to-date, universities (and other educational institutes) have to provide professional training of these skills. One goal of this thesis is to provide qualitative eLearning support for teaching UML.

## 1.2 Problem Description

The course “Object Oriented Modeling (OOM)” at the Vienna University of Technology (VUT) introduces modeling and UML for about 1000 students a year [7]. To cope with this large number, e-Learning elements have been established. While the theoretical aspects are covered quite well by the current e-Learning elements, there are some lacks when it comes to practical modeling exercises.

Currently, the practical modeling exercises are handled only partially on the e-Learning platform: The problem description of the exercise is presented online. The student models a diagram apart from the platform (by hand or with a proprietary modeling tool). Then he answers questions about his modeled solution or compares it to the standard solution that is presented on the platform. Both ways are problematical: Modeling by hand on a sheet of paper is very inconvenient due to the lack of automation, especially when there are many diagram elements. On the other side, proprietary

## 1 Introduction

modeling tools are difficult to understand for novices and often do not conform to the UML standards that are taught in the lecture.

In the future, students shall be able to model their solution by the aid of a software modeling tool that conforms to the standards of the lecture, and the whole process shall be integrated in the e-Learning platform.

### 1.3 Approach

The mentioned problems are tackled by creating a web-based modeling tool that is able to be integrated into the e-Learning platform of the VUT using Rich Internet Application technologies. Thus, students can solve practical modeling exercises comfortable direct on the e-Learning platform. Therefore, a generic approach is to be used: visualization and manipulation of two-dimensional diagrams is based on certain patterns. A framework will be created, which allows creating complete editors with drag-and-drop functionality, by specifying by specifying the structure of the diagram elements (abstract syntax) and the visual notation elements (concrete syntax). This enables to easily create various UML editors and to align the notation elements with those of the course. In the course of this thesis, three UML editors are to be created with the aid of the framework and integrated into the learning platform.

The thesis combines three big topics: Metamodeling, e-Learning, and Rich Internet Applications (RIA): With Rich Internet technologies a metamodeling framework is build that allows an easy creation of diagram editors, which can be used for teaching modeling.

This approach exhibits several advantages: First, it allows the interactive creation of diagrams with a graphical sophisticated user interface, without the need of installing a proprietary modeling software. Then, solution is completely integrated in the e-Learning platform and the notation elements can be aligned with the lecture. Furthermore, the software is operating system independent and available anytime and anywhere.

### 1.4 Structure

This thesis consists of eight chapters:

- Chapter 1 (“Introduction”) introduces the contents of this thesis
- Chapter 2 (“Basic Concepts”) introduces metamodeling, e-Learning and Rich Internet Applications.
- Chapter 3 (“Developing the Web Modeling Framework”) deals with the main challenges of developing a web-based, generic framework for the creation of 2D diagram editors.
- Chapter 4 (“Sample Application”) demonstrates how to use the framework to build a specific diagram editor.
- Chapter 5 (“Developing eLearning Support with WebMF”) shows the creation of three UML diagram editors (class diagram, state diagram, sequence diagram) using the WebMF framework and the integration into the e-Learning platform Moodle.

- Chapter 6 (“Related Work”) compares this approach to other approaches in the field of web based 2D modeling.
- Chapter 7 (“Evaluation”) points out strengths and weaknesses of the approach.
- Chapter 8 (“Conclusion”) summarizes the thesis and suggests topics for future work.

## *1 Introduction*

## 2 Basic Concepts

This chapter explains three topics that are relevant for the creation of the web-based modeling software. Section 2.1 describes modeling and metamodeling. Section 2.2 introduces e-Learning basics and heuristics for creating effective e-Learning tools. In Section 2.3, the most popular Rich Internet Application technologies are introduced and compared.

### 2.1 Metamodeling

This section introduces metamodeling techniques. First, the core concepts are explained (Section 2.1.1). Then, the constitution of modeling languages is analyzed (Section 2.1.2). Finally, the Eclipse Modeling Project (EMP), which is the major metamodeling framework in the Java environment, is presented (Section 2.1.3).

#### 2.1.1 Core Concepts

Model Driven Engineering (MDE) [39] is a software development methodology which focuses on creating models rather than computing concepts. It should increase productivity by simplifying the design processes and communication between team members. The most important MDE initiative is Model Driven Architecture (MDA) proposed by the Object Management Group (OMG)<sup>1</sup>, which is based upon several OMG standards.

Jean Bezivin [6] stated a paradigm shift in the field of software engineering: Whereas in classical OOM the basic principle was “Everything is an object”, MDE led to a new principle: “Everything is a model”. The basic relations of OOM are *instanceOf* and *inheritsFrom* (Figure 2.1): An object can be an instance of a class and a class can inherit behavior from other classes. In MDE, other relations are focused: A particular view of the system is *representedBy* a model, and each model *conformsTo* its metamodel.

Metamodeling promises a set of advantages: It provides concise and precise definition of language concepts. It also provides a uniform data exchange format and allows for validating of the correctness of models. Finally, the administration of models can be simplified by the use of model repositories.

A metamodel defines a language for models that conform to this metamodel. While each metamodel itself conforms to its metamodel, this leads to a model hierarchy, which consists of four (3+1) layers (Figure 2.2) [6]:

The bottom layer (M0) is the real system. This system is *representedBy* a model (M1). That model *conformsTo* its metamodel (M2). In turn, that metamodel *conformsTo* its metamodel (M3).

To avoid confusion, it is important to do not mix OOM and MDE notions. Figure 2.3 illustrates the different focuses of metamodeling and metaprogramming. The horizontal axis shows metaprogramming aspects: “Felix” is an instance of the Java class “Cat”. The vertical axis shows metamodeling aspects: “Felix” is a Java instance. The relations

---

<sup>1</sup>OMG: <http://www.omg.org/>

## 2 Basic Concepts

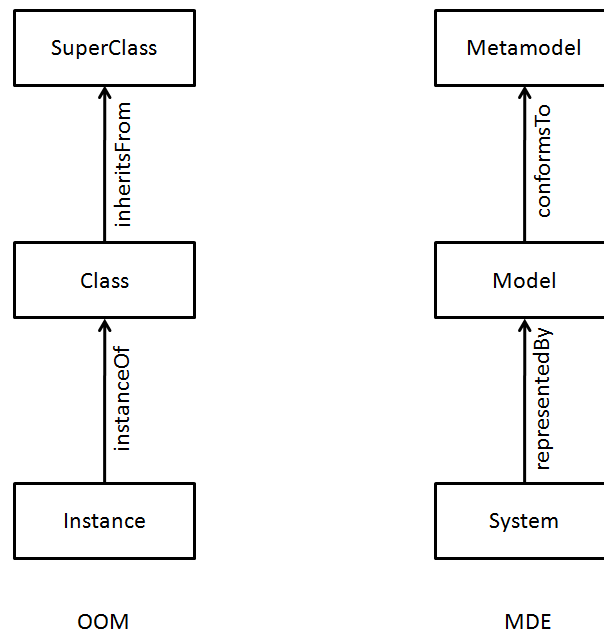


Figure 2.1: Basic notions in OOM and MDE [6]

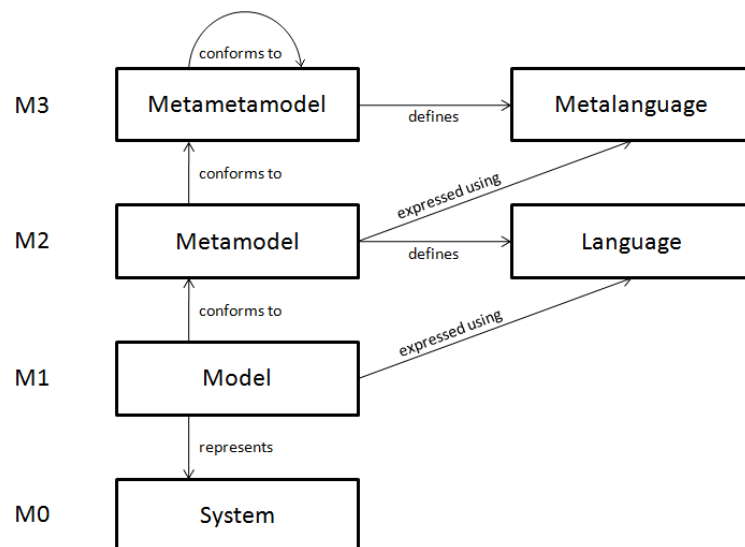


Figure 2.2: Metamodeling: Levels of abstraction [6]

are also referred as “ontological metamodel” (metamodeling) and “linguistic meta-model” (metaprogramming) [3].

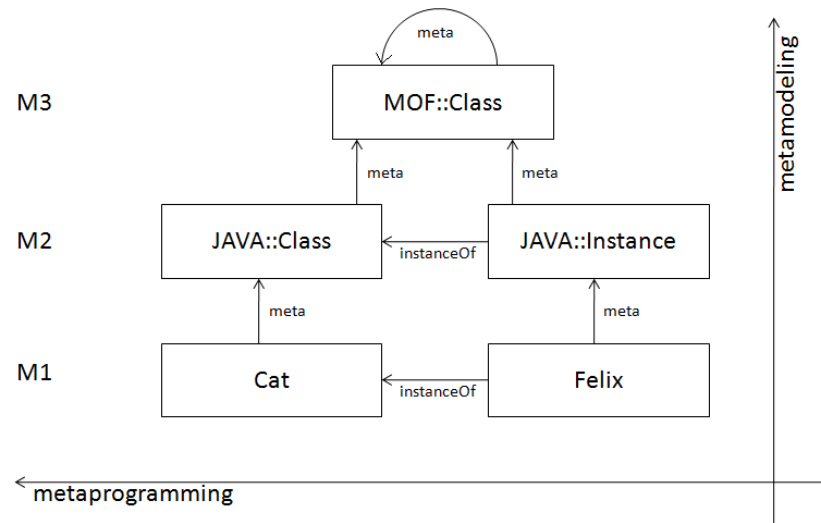


Figure 2.3: Metamodeling vs. metaprogramming [6]

## Meta Object Facility (MOF)

The Meta Object Facility (MOF) [30] is the metamodel standard for MDA that was established by the OMG. It defines language concepts for modeling object oriented structures. These language concepts are reflexive, i.e. MOF is itself described in MOF. It is cut into two parts: EMOF (essential MOF) and CMOF (complete MOF).

The core constructs (classes, attributes, operations, and parameters) in EMOF are depicted in Figure 2.4: Objects are defined as generalizable classes (**superclass**). Classes have intrinsic properties (**Property**). Relationships between classes are defined as typed properties. The related **Class** is defined as type of the property of the source class. If the relationship is bidirectional, both classes have properties referring each other with the opposite association.

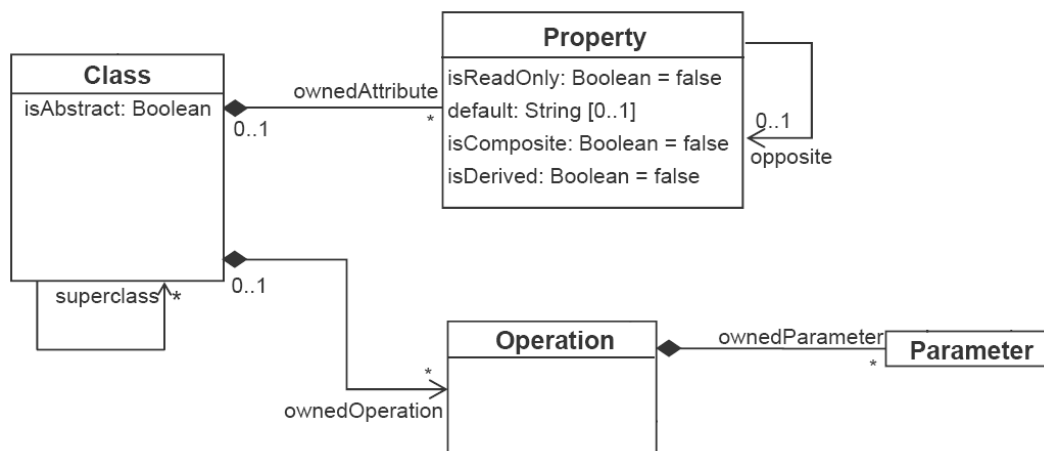


Figure 2.4: EMOF language core

## XML Metadata Interchange Format (XMI)

The OMG Standard XMI [31] allows XML based storage and exchange of models. For this purpose it defines a set of rules for the transformation of MOF based metamodels to XML Schema or XML DTD and transformation of MOF based metamodels to XML documents (Standard file extension: \*.xmi).

This allows simple exchange of models between different (UML) modeling tools. Complementary to XMI, the XMI Diagram Interchange (XMI[DI]) [32] standard, describes layout information of models.

### 2.1.2 Modeling Languages

Modeling languages allow for expressing models of a target meta model. Due to the specific purpose they are also called Domain Specific Languages (DSLs). Modeling languages can be either textual or visual (diagrammatic), but the theoretical principles behind them are almost the same [19]. Visual languages are also referred as Domain Specific Visual Language (DSVL).

Textual languages consist of linear character strings and symbols (words, sentences, etc.). They reach from natural languages like English to highly formalized languages like XML. Visual languages, on the other side, basically consist of graphical elements, but can also contain textual elements (think of a class in a class diagram which shows the class name on top).

In common, one has to distinguish between the syntax (the notation) and the semantic (the meaning) of a language, since they have different purposes, styles, and usage (Figure 2.5).

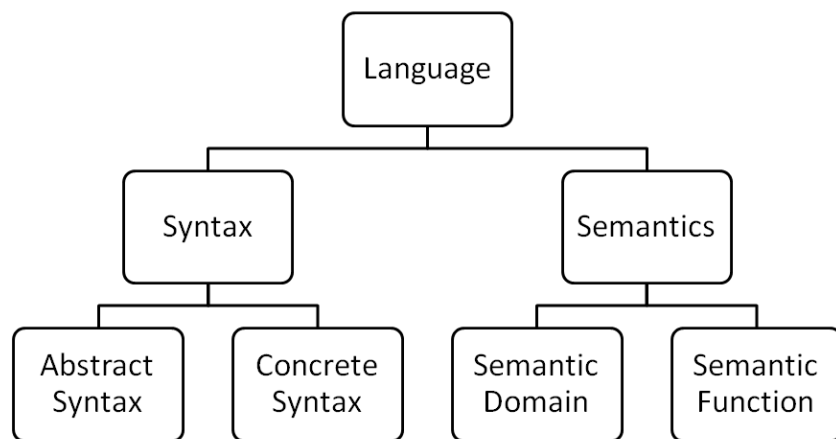


Figure 2.5: Parts of modeling languages

The *syntax* only describes notational aspects of a language, independently of the meaning. A syntax description consist of a set of elements (called “expressions”) that are used in the communication. A textual language can define, e.g., words and how words can be combined to a sentence. In the same manner diagrammatic languages define graphical elements and how they can be related together.

The *semantic* describes the meaning of a notation and consists of two parts. The *semantic domain* is a description of the targeted application domain. Its elements describe the properties of this system. The *semantic mapping*, however, relates each syntactic construct to a construct of the semantic domain. This is typically done just

with a textual description (like in the UML) where new constructs are defined using other constructs that are already known.

Everything that is seen on a screen or on paper is a syntactic representation, but also the internal representation in the software system, the so-called *abstract syntax* or metamodel is a syntactic representation. Correspondingly, the visual or textual representation is called *concrete syntax*. One semantic can be described by multiple abstract syntaxes, and vice versa. The same holds for the relationship between abstract syntaxes and concrete syntaxes, resulting that a concrete syntax can describe the same semantics with different abstract syntaxes (Figure 2.6). For example, the semantics of UML class diagrams is known, but there are many different implementations that use different internal data structures.

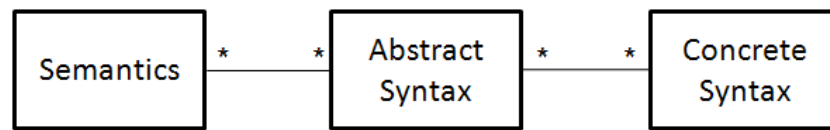


Figure 2.6: Multiplicity in language definitions

However, there is still disunity about the notions in this field. This thesis keeps to those that are explained above.

### 2.1.3 Eclipse Modeling Project (EMP)

The Eclipse Modeling Project [18] is known as *the* modeling technology for the Java programming language. It features code-generation for building Java applications based on simple model definitions. Its main part, the Eclipse Modeling Framework (EMF), combines Java, XML, and UML. The programmer can specify the model with annotated Java Classes, XML schema definitions, or an UML tool. On the basis of one of these definitions, EMF can generate all others (Figure 2.7).

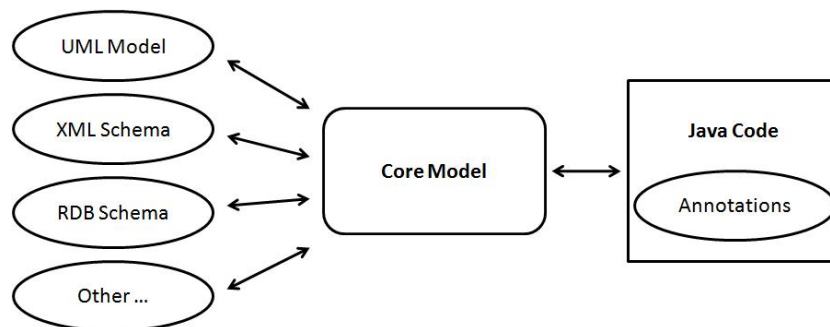


Figure 2.7: EMF Model Creation

While EMF itself is a Eclipse plug-in, it also features simple interfaces for building EMF viewers and editors for the specified models. The plug-in architecture of Eclipse uses this concept to share data between various plug-ins.

### Abstract Syntax

Ecore, the metamodel of EMF, is based on the EMOF standard. It provides straightforward transformation to the elements of the Java language and to XML. Figure 2.8

shows the Ecore class hierarchy: All elements of Ecore inherit from `EObject` which is an equivalent to Java's `java.lang.Object`. Primarily, there are two branches: `EClassifier` and `ETypedElement`. `EClassifier` sums up all types (classes, data types and enumerations). `ETypedElement` covers all elements that have a type (`EClassifier`), e.g. attributes (`EAttribute`), references (`EReference`), and operations (`EOperations`).

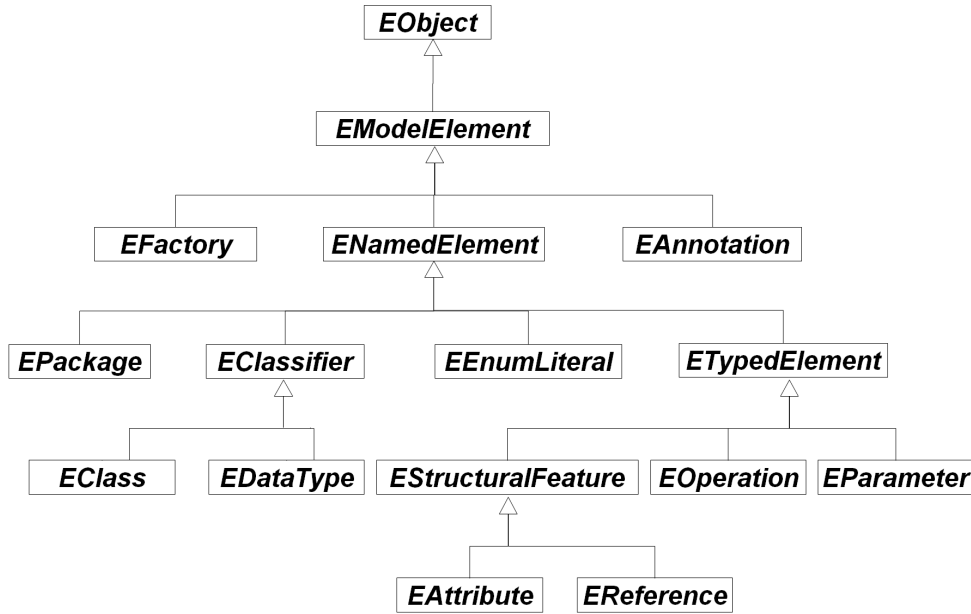


Figure 2.8: Ecore Class Hierarchy [8]

Figure 2.9 shows the Ecore Kernel. The elements refer to common object oriented concepts (classes, attributes, relationships, inheritance): Inheritance of classes (`EClass`) is defined via an association (`eSuperTypes`). Attributes (`EAttribute`) and references (`EReference`) are aggregated by classes (`EClass`). References (`EReference`) describe cardinality (`lowerBound`, `upperBound`) and containment type (`containment`, composition if true, association otherwise). Binary associations are described with two references (`EReference`) which refer to each other (`eOpposite`). Finally, data types (`EDataType`) base upon Java data types. The structure of Ecore and MOF is similar and they influence each other in their evolution.

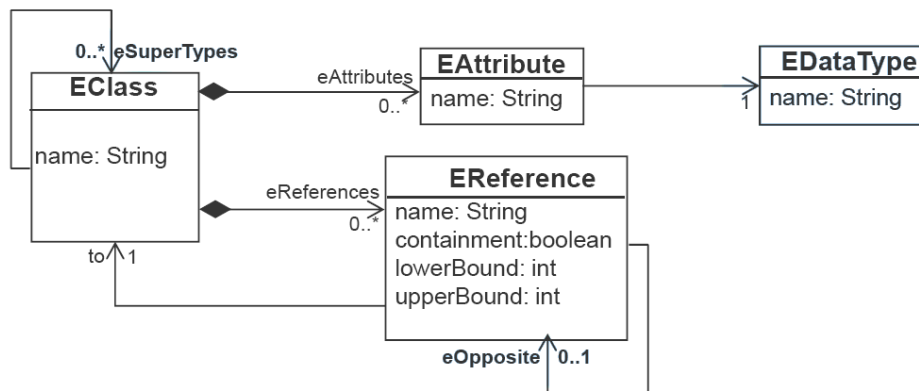


Figure 2.9: Ecore Core Classes

## Concrete Syntax

Building on the Ecore / EMF based abstract syntax one can define a textual or visual concrete syntax. The Graphical Modeling Framework (GMF) allows for defining a visual concrete syntax and therefore uses the graphics / UI library Graphical Editing Framework (GEF). Moreover, it provides an infrastructure and runtime environment for diagram manipulation. The Textual Modeling Framework (TMF) analogously allows for defining a textual concrete syntax.

## 2.2 E-Learning

The term e-Learning covers all forms of learning, where digital media are used for the presentation and distribution of learning materials and/or for the support of interpersonal communication [25]. The wide range of e-Learning forms includes:

- Computer-based training (CBT): The use of computers and special learning software (e.g. CD-ROM/DVD software)
- Web-based training (WBT): Use of the Internet for providing learning materials and, additionally, communication facilities
- Blended Learning: The combination of e-Learning and attendance in training courses. [21]

The main reason to establish e-Learning are time independence and location independence (“Learn anytime, anywhere”). These advantages are enabled by Learning Management Systems.

### 2.2.1 Learning Management Systems

Learning Management Systems (LMS) are complex software systems which allow to supply learning material and to organize learning processes. Generally, there are five functionality areas in LMS (Figure 2.10) [5]:

- Administration: All involved students, contents, and courses are administrated.
- Presentation: Taught contents can be provided as text, graphics, images, audio, video, etc.
- Communication: Teachers and students can communicate asynchronously (e.g. e-mail, forums) or synchronously (e.g. chats, application sharing).
- Production: Tools allow for the creation of exercises and assignments.
- Evaluation and rating: The students’ performance can be evaluated and rated.

## Moodle

There is a high number of LMS available, both open source and commercial products. One of the most popular LMS is Moodle<sup>2</sup>, which is available under open source license.

---

<sup>2</sup>Moodle Learning Management System: <http://www.moodle.org>

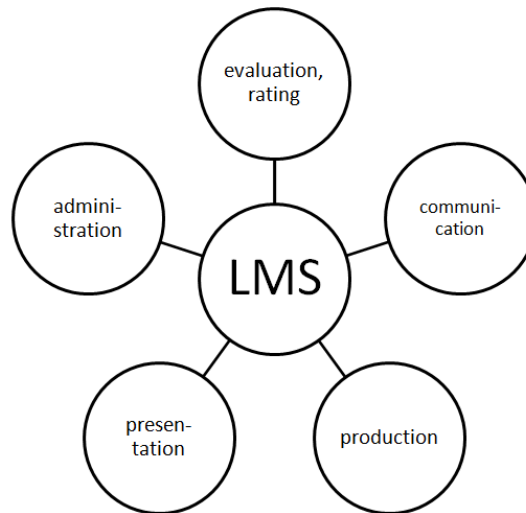


Figure 2.10: Functions of Learning Management Systems [28]

Moodle is also an acronym and stands for “Modular Object Oriented Dynamic Learning Environment”. It is a classical web application, based on the technologies PHP, MySQL, HTML, and JavaScript. For functional extension, it provides a comfortable plug-in mechanism.

Moodle is designed for the typical stakeholders and elements in a learning environment: courses, modules, resources, teachers, and students. Teachers can create courses for specific topics and add contents. The course contents are twofold: *Resources* represent learning materials (e.g. web pages, text pages or files of any type) that are provided for the students. *Activities* encourage students to exercise themselves in the treated topic (e.g. assignments, forums, chats, wikis, quizzes).

### TUWEL

The corporate LMS of the Vienna University of Technology (VUT) is TUWEL (“TU Wien ELearning”, “VUT e-Learning”) <sup>3</sup>. TUWEL builds upon Moodle and was slightly adapted. It was integrated with the central authentication service of the VUT and the course administration system TUWIS (“TU Wien Informationssystem”, “VUT information system”). Further, the corporate design of the VUT was adopted. Today, almost 700 courses are available on the TUWEL platform.

### 2.2.2 Object-Oriented Modeling Course at the VUT

The course “Object-Oriented Modeling (OOM)” at the VUT gives students an understanding of UML. The contents of teaching are required for advanced courses like Software Engineering or Model Engineering [7].

Due to the restructuring of course curricula at the VUT in 2006, the number of attending students jumped to about 1000 each year. The preceding courses had been much smaller and the practical exercises were supervised by student tutors, who corrected and discussed the solutions with groups of two students. The course administration saw that this mode did not scale for so many students. First, they could not find

---

<sup>3</sup><http://tuwel.tuwien.ac.at/>

enough appropriate tutors who were able and willing to hold the tutorials. Second, the management of the groups led to a large administration overhead.

It was decided to reorganize the course and to divide it into three parts: the lecture, a practical part and e-Learning.

file:///localhost/C:/Users/Work/Desktop/tuwel-oom/KD4\_%20Supermarkt.mht

Eine Applikation für einen Supermarkt verwaltet Daten über Abteilungen und Mitarbeiter, wobei es Lehrlinge und Angestellte gibt. Von jedem Mitarbeiter werden seine eindeutige Sozialversicherungsnummer, Personalnummer, Name, Adresse und die Abteilung, in der er arbeitet, gespeichert. Lehrlinge haben zusätzlich noch ein Lehrjahr. Von jeder Abteilung werden die eindeutige Bezeichnung, die Mitarbeiter, der Abteilungsleiter (ein Angestellter) und die angebotenen Artikel gespeichert. Jede Abteilung bietet mehrere Artikel an, wobei ein Artikel nur von einer Abteilung angeboten werden kann. Ein Artikel wird durch seine Artikelnummer identifiziert, darüber hinaus werden die Bezeichnung und der Verkaufspreis pro Einheit gespeichert. Ein Artikel wird von mindestens einem Lieferanten angeboten.

Lieferanten werden durch ihre Bezeichnung und Adresse identifiziert. Jeder Artikel kann von jedem Lieferanten zu jeweils unterschiedlichen Konditionen angeboten werden. Konditionen werden durch den Einkaufspreis pro Einheit, die Lieferanteninterne Artikelnummer (unterschiedlich von der Artikelnummer des Supermarkts) und maximale Rabatt-Konditionen näher beschrieben. Nicht alle potentiellen Lieferanten liefern tatsächlich.

Eine Lieferung wird von einem Lieferanten durchgeführt. Sie findet an einem Datum statt und ist mit einer eindeutigen laufenden Nummer versehen. Eine Lieferung umfasst mehrere Artikel. Pro Artikel wird gespeichert, in welcher Menge und zu welchem tatsächlichen Preis pro Einheit die Artikel geliefert wurden.

**Erstellen Sie ein UML-Klassendiagramm.**

1 Wollen Sie die Musterlösung sehen?

Antwort wählen:

☒ a. Ja

```

classDiagram
    class Mitarbeiter {
        +pn: int
        +svnr: int
        +name: String
        +adresse: String
    }
    class Lehrling {
        +lehrjahr: int
    }
    class Angestellter {
    }
    class Abteilung {
        +bezeichnung: String
    }
    class Artikel {
        +bezeichnung: String
        +id: int
        +preis: int
    }
    class Lieferung {
        +datum: Date
        +id: int
    }
    class Lieferant {
        +bezeichnung: String
        +adresse: String
    }
    class Info {
        +menge: int
        +preis: int
    }
    class Konditionen {
        +id: int
        +preis: int
        +rabatt: int
    }

    Mitarbeiter <|-- Lehrling
    Mitarbeiter <|-- Angestellter
    Mitarbeiter "1" -- "*" Abteilung : arbeitet_in
    Angestellter "1" -- "0..1" Abteilung : leitet
    Abteilung "1" -- "*" Artikel
    Lieferung "*" -- "1" Lieferant : führt_durch
    Lieferung "*" -- "*" Artikel
    Lieferant "1..*" -- "*" Artikel
    Artikel "*" -- "*" Info
    Artikel "*" -- "*" Konditionen
  
```

Figure 2.11: Screenshot of a modeling exercise in the TUWEL OOM course

- The lecture is held in a traditional manner. The five most relevant UML diagrams (class, sequence, state, activity, use case) are presented in the auditorium. Attendance is recommended but not mandatory.
- In the practical part the students are divided into groups of 50 persons. The students have to solve a number of modeling exercises for several meetings. For each exercise a student is picked by the professor to present his solution to the others. Then the group should discuss different approaches for the exercise. In practice, the students draw their diagrams on a sheet of paper and take it along to the meeting. For the presentation, they draw their solution on the blackboard.
- Broad e-Learning support is provided for all diagram types. The contents are twofold. First, there are multiple choice questions that cover the theoretical background. For example, a state diagram is presented to the student and he has to

figure out, in which state the system is after a certain sequence of events (Figure 2.11). Second, there are open modeling exercises with a standard solution. A textual description of a problem domain is given. The student has to model his solution. He can do this by hand or with the aid of a modeling tool. Then he can compare his solution with the standard solution, which is provided by the course staff on the platform.

All learning materials are provided on the TUWEL course. The administration (course registration, grading, etc.) is also handled via the e-Learning platform.

An evaluation [7] showed that this new organization performs much better than the preceding one and is honored by both teachers and students. Nevertheless, more advanced e-Learning solutions were suggested by the evaluation authors.

### 2.2.3 Perspectives of a Course Centered Around a Modeling Tool

While OOM is a short introductory course (3 ECTS <sup>4</sup>), the introduction of a complicated commercial modeling tool does not make sense. Moreover, the notation of the specific diagrams would not conform to the ones used in the course.

The use of a simple tool for drawing diagrams (according to the notation used in the lecture) throughout the course promises better learning success. As an online tool, integrated into the e-Learning platform, this tool could be used in a large variety of forms.

- In the lecture, the lecturer can show the modeling procedure with a beamer in front of the students.
- For the practical part, the students provide their solutions online. In the meetings, the students' solution can be displayed in the TUWEL environment on a beamer. Corrections of the solutions also can be done live.
- The practical modeling exercises in the e-Learning part can be solved directly on the platform. Optionally, the solutions can be viewed and rated by the course staff (online assignments).
- Students can post their diagrams in a TUWEL forum to discuss them with the other students.

Once the framework is established, persons that are involved in the course can utilize the framework for new e-Learning applications with new ideas and miscellaneous didactic approaches. Independent from the form of integration, the use of such a tool throughout the course promises many benefits:

- Quick and comfortable creation and manipulation of models with a graphical sophisticated user interface.
- Available anywhere, anytime.
- No installation of a proprietary modeling software.
- A consistent notation that can be aligned with the lecture.

---

<sup>4</sup>ECTS is the European Credit Transfer System which helps to compare courses of universities across Europe (<http://www.ects.at/inhalt1.php>)

## 2.3 Rich Internet Applications

The use of Rich Internet Application (RIA) technology promises several benefits for providing e-Learning support in the field of UML diagrams. First, this chapter introduces basic notions (Section 2.3.1) and technologies (Section 2.3.2). Then these technologies are compared, finally leading to the technology choice for the implementation (2.3.3).

### 2.3.1 Basics

Over the past years there were essential innovations in the field of web applications. The use of web applications (social networks, wikis, user generated content, etc.) becomes more user centered and interactive. Complementary, new technological concepts (Ajax, thin clients, etc.) support these changes.

Feature	Desktop Applications	Web Applications	Rich Internet Applications
Rich user experience	+	-	+
Interactive, responsive	+	-	+
Low maintenance	-	+	+
High reach	-	+	+

Table 2.1: RIA settlement

A RIA is something between a web application and a desktop application (see Table 2.1). The strength of traditional desktop applications are their highly responsive and interactive user interfaces and multimedia abilities. Contrary, the major benefits of web applications are their high reach <sup>5</sup>, low maintenance and platform independence. RIAs attempt to combine the benefits of both approaches and to avoid their drawbacks.

In a nutshell, the key benefits of RIAs are [34]:

- A rich user interface which is not downloaded repeatedly as in traditional web applications.
- Interactive UI controls make the application more feel like a desktop application.
- The application reacts immediately to user actions without reloading the entire application.
- Contrary to traditional web applications some actions can be entirely handled by the client without asking the server.
- A reach as traditional web applications as they are accessible via the web.
- Maintenance costs are as low as those of conventional web applications. The application is not installed locally and therefore no updates have to be distributed.

---

<sup>5</sup>

### 2.3.2 RIA Technologies

The most popular technologies for the creation of RIAs — as stated by Ostermaier [34] — are briefly described in the following. Later in this section they are evaluated, and the best fitting technology is used for implementation (Chapter 3).

#### Flash / Flex

Flash <sup>6</sup> has its origin in multimedia applications. It is primarily used for animations which are displayed via a Flash Plug-in in the browser window. But with its own scripting language ActionScript (AS), Flash also allows for building highly interactive applications.

Flash applications can be developed with the aid of the authoring tool Adobe Flash [45]. Developers define graphical objects and their transformations for specific points in time (time line) and specify their interactive behavior (AS).

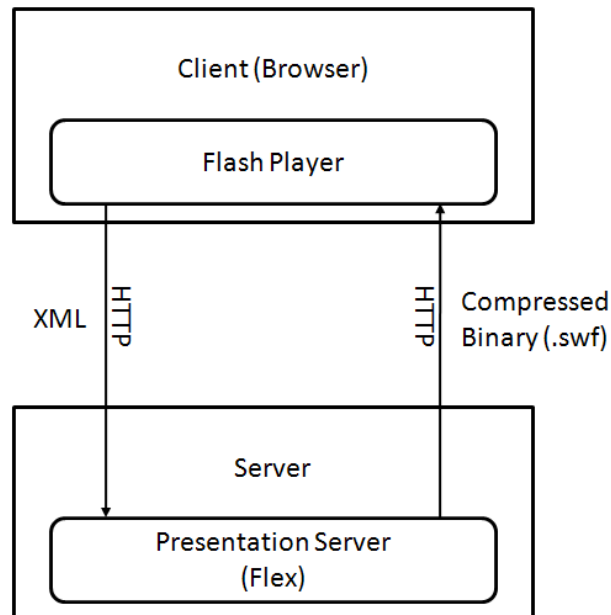


Figure 2.12: Architecture of Flash/Flex applications

While Flash’s strengths are located in multimedia, it has weaknesses when building data intensive form based GUIs. Here, Flex [46] comes into play: The Flex API and SDK allow for building component based applications that compile to Flash binaries (\*.swf). For complicated issues, it follows the approach of Code-Behind [1]: The layout of components is defined with an XML file (the XML derivative MXML, \*.mxml), whereas the logic of the specific component is encapsulated in an ActionScript class (\*.as).

The standard Flash / Flex deployment architecture is shown in Figure 2.12. The binary code (UI definition and UI logic) is compressed to a \*.swf file and is placed on the server. The client’s browser loads an HTML page, which defines an embedded object (the \*.swf file) to be executed with the Flash plugin. Further communication between the application and the server (for example database usage) at runtime (embedded in the Flash plug-in) is realized via XML requests over HTTP(S).

<sup>6</sup>Adobe Flash: <http://www.adobe.com/flash/>

Adobe AIR (Adobe Integrated Runtime) <sup>7</sup>, which is built on the same core as Flash and Flex, is a cross-platform runtime environment for RIAs. Unlike most other RIA technologies it does not use the web-browser. AIR applications run directly on the desktop. Like Flex, AIR applications are built with the Adobe Flex Builder around MXML components and ActionScript classes.

Flash was released by Macromedia in 1999 and the Flash plug-in has actually (December 2008) a penetration of 99% (i.e. 99% of locally installed browsers can display this type of content) [2].

## Silverlight

Silverlight <sup>8</sup> is a light-weight version of Microsoft's Windows Presentation Foundation (WPF) [40]. With Silverlight, developers can build RIAs including videos and animations using the eXtensible Application Markup Language (XAML [26], XML derivate for graphical definitions) and JavaScript (logic behind). Silverlight can be seen as Microsoft's counterpart to market leader Flash in the field of multimedia integration into web pages.

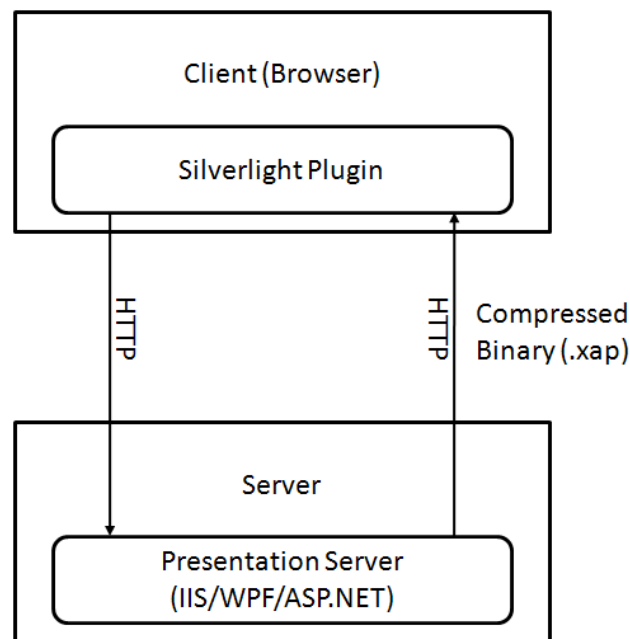


Figure 2.13: Architecture of Silverlight applications

The Silverlight deployment architecture is similar to Flash or Flex (Figure 2.13): The Silverlight application is loaded from the presentation server and running inside the Silverlight browser plug-in. Further Communication can be realized via HTTP(S) requests.

It is very advisable to use Silverlight in projects that set up on the .NET framework. Silverlight plug-ins are available for Internet Explorer, Firefox, Chrome and Safari web browsers, but not for Opera. In December 2008, the penetration of the Silverlight plug-in was at only 25 % of PCs [23].

<sup>7</sup>AdobeAIR: [http://www.adobe.com/devnet/air/flex/articles/airf15\\_training.html](http://www.adobe.com/devnet/air/flex/articles/airf15_training.html)

<sup>8</sup>Silverlight: <http://www.microsoft.com/silverlight/default.aspx>

### JAVA applets

Java Applets [43] were very early attempts for providing RIA. Applets can run in the browser on any platform which has installed an adequate Java Runtime (JRE). The compiled Java Applet bytecode (\*.class / \*.jar files) is located on the server and referenced in an HTML object-tag (Figure 2.14).

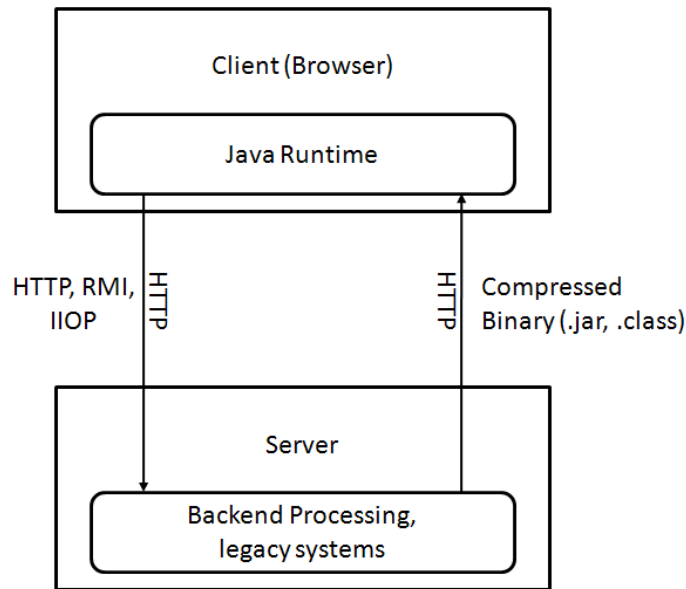


Figure 2.14: Architecture of applet based applications

An advantage of applets is that they can use the entire J2SE API and have therefore a broad application range. The communication is not limited to the HTTP(s) and therefore not bound to the request - response paradigm. High time consumption of loading the JVM and initializing the applet are major drawbacks of this technology. For security reasons, applets run in a restricted environment called sandbox. In December 2008 the JVM had a penetration of approximately 88% [2].

### AJAX

Ajax (Asynchronous JavaScript and XML) [16] is a concept for building RIAs based on the established technologies JavaScript and XML. Contrary to other RIA technologies, Ajax applications don't need a browser plug-in. With JavaScript, which is provided by nearly every modern web browser, the Document Object Model (DOM) [10] of the underlying HTML page can be accessed and manipulated in order to react on UI events. Further, it provides asynchronous HTTP requests to the server (Figure 2.15). When the data is received, the registered callback function is called. The underlying data format of the communication is not limited to XML, as the name presumes, it can also be formatted in JavaScript Object Notation (JSON) [12] or just plain text. In the case of Ajax, the UI definition is declared via standard HTML, the logic is implemented in JavaScript.

There is a huge number of AJAX frameworks available for most popular web technologies. They reach from low level JavaScript libraries (e.g. [script.aculo.us](http://script.aculo.us/)<sup>9</sup>, proto-

<sup>9</sup>scriptaculous JavaScript library: <http://script.aculo.us/>

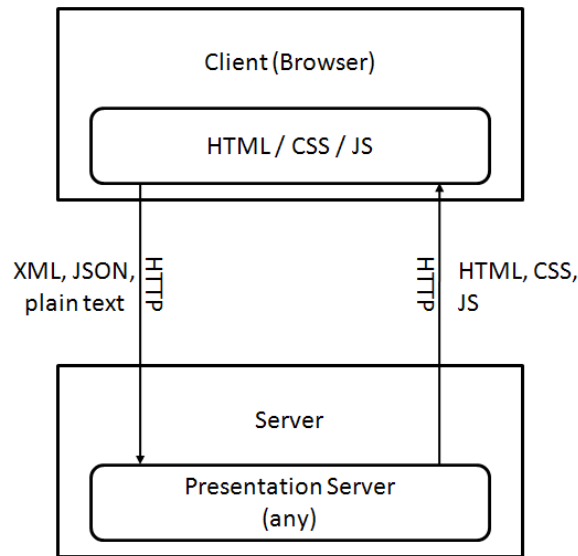


Figure 2.15: Architecture of AJAX applications

type <sup>10</sup>) to high level, widget based, frameworks (Adobe Spry <sup>11</sup>, Google Web Toolkit <sup>12</sup>, ASP.NET Ajax <sup>13</sup>) which relieve developers from JavaScript coding.

Ajax has a very high reach, as it only requires a JavaScript capable web browser that provides asynchronous HTTP requests. The major disadvantage of AJAX are incompatibilities between the web browsers' different JavaScript implementations.

## XUL

XUL (XML User Interface Language) is an XML based language for the description of RIA user interfaces. The language has been developed in association with the Mozilla web browser and is used for the browser's UI. Developers can implement XUL applications by specifying the UI structure with XUL and implementing application logic with JavaScript. XUL applications depend on Gecko <sup>14</sup>, the Mozilla layout engine.

The standard deployment architecture for XUL applications is illustrated in Figure 2.16. The UI specification is loaded by the Gecko based web browser. Further asynchronous communication is realized via simple HTTP requests and XML.

XUL applications run in the rendering engine which is included in the web browser (e.g. Mozilla Firefox). The major drawback is that ZUL is provided by only less than a half of all used browsers [44]. Since Microsoft pushes its own declarative UI language (XAML, see Section 2.3.2), it is improbable that they will support ZUL in Internet Explorer.

<sup>10</sup>prototype JavaScript library: [www.prototypejs.org/](http://www.prototypejs.org/)

<sup>11</sup>Adobe Spry: <http://labs.adobe.com/technologies/spry/>

<sup>12</sup>GWT: <http://code.google.com/webtoolkit/>

<sup>13</sup>ASP.NET-Ajax: <http://www.asp.net/ajax/>

<sup>14</sup>Gecko Layout Engine: <http://www.mozilla.org/newlayout/>

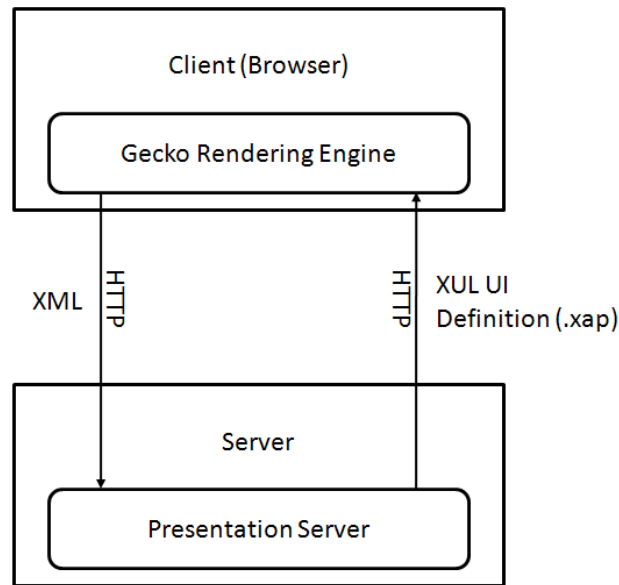


Figure 2.16: Architecture of XUL applications

### 2.3.3 Evaluation and Choice

There are two prerequisites for this project to allow the integration into the current, Moodle<sup>15</sup> based, e-Learning platform (TUWEL):

1. The RIA can be embedded in HTML (e.g. HTML `object` tag, JS).
2. The RIA technology is supported by the majority of web browsers (Internet Explorer, Mozilla Firefox, Opera, Google Chrome, Safari).

From the most popular RIA technologies described in the previous section Java, Ajax, and Flash/Flex fulfill these prerequisites. There is a large variety among criteria for RIA technologies. The remaining RIA technologies are compared according to 10 criteria groups, that were identified by Noda and Helwig in [29].

#### Graphical Richness

Graphical Richness is essential for the creation of a rich, high responsive, interactive diagram editor. The technology of choice has to provide support for:

- Custom UI components (notation elements, tool bars, property editors), which ideally can be specified with a declarative language
- UI components that can be arranged freely (absolute layout) and are dynamically adjustable
- Adequate drawing API (lines, circles, etc.)
- Drag-and-drop support

Flex performs best in this category, it provides declarative UI component specification, a rich drawing API, etc.

<sup>15</sup>Moodle Learning Management System: <http://www.moodle.org>

With Ajax, there are more possible ways for the graphical layout, but all have some weaknesses. Standard HTML containers would suffice for rendering boxes, but HTML fails does not support drawing lines etc. The Canvas HTML-Tag provides 2D graphics with JS functions. But beside challenging scripting code it is not available across all main browsers (no support for Internet Explorer). Similarly, the more popular SVG graphics is not available on all major web browser.

Java Applets match the requirements, but are not as comfortable in the implementation as Flex, which is purposed for these concerns.

### **Container / Engine Footprint**

Container / engine footprint means a measure of influences to the environment of the local client PC, e.g. the need of installed software, use of local disk space or memory, etc. Java Applets need the relatively heavyweight JRE to be downloaded and installed, which still is acceptable. Flash/Flex (lightweight plug-in) and AJAX (no additional SW) are much lighter.

### **Application Download**

AJAX also has the shortest download time. There is no need to load the whole application at the beginning. The downloaded UI, data and logic is all text-based and requires only little traffic. UI and logic of Java Applets and Flash/Flex is transferred via byte-code, which creates more traffic. For the case of large applications, both Java Applets and Flash/Flex provide mechanisms where parts of the application can be downloaded on demand.

### **Audio / Video Support**

Audio and video support is not very significant for this project. Nonetheless, future versions could include e.g. learning videos. For these aspects, the multimedia oriented Flash/Flex technologies have good audio and video support, followed by Java Applets and Ajax, which only support browser built-in audio and video functionality.

### **Consistency on Different Computing Environments**

Most AJAX applications have serious problems with different browsers and different browser versions, which often increases production and maintenance costs. JAVA, and—especially—Flash/Flex applications are more consistent on different client system environments. It is a major issue for selecting RIA technology.

### **Server Requirements**

All three considered technologies provide standard XML HTTP requests, with which they can communicate with most web servers. Specific web server technologies can optimize the communication, but are not necessary. Particularly, the use of JAVA applets would unify language and communication when using EMF and/or Tomcat web server.

### **Plug-in / Runtime Requirement on Client**

High reach is guaranteed by all 3 technologies: AJAX does not require any plug-in or runtime. Flash/Flex depends on an appropriate Flash plug-in, and JAVA applets on

the JAVA JRE, but both are available on most clients.

### Development Challenge

AJAX (HTML/JS) coding can be very complex, especially with the browser problematics mentioned above. Flash/Flex's ActionScript 3 is significantly better as it conforms to most principles of established object oriented languages like JAVA.

### Security Concerns

Flash/Flex and JAVA applications are provided as compressed binaries, the Flash plugin (Flash/Flex)—respectively JRE (JAVA Applets)—act as a sandbox. Contrary, the JS logic of AJAX applications is open to public. Authentication, authorization, and encryption can be realized with HTTP(s) based communication to a specific web server.

### Cost

Almost all parts of RIAs can be realized using cost free technologies: Established web servers are provided by the Apache Software Foundation <sup>16</sup> (Apache web server, Apache Tomcat). Client code (HTML, JS, or Java) can be implemented with the help of Eclipse plug-ins. Only the Flex Builder API for building client UI causes some costs when using Flex based RIA (actually about \$700,-).

### Technology Choice

The criteria are weighted upon the relevance for this project (Table 2.2). The graphical richness is critical for displaying a wide variety of notation elements. Also the consistency on different computing environments is essential for eliminating unsuspected behavior for certain students, which would cause inequity in their rating.

Importance	Criteria
A (high)	Graphical Richness Consistency on Different Computing Environments
B (middle)	Security Concerns Cost
C (low)	Container / Engine Footprint Application Download Audio / Video Support Server Requirements Plugin / Runtime Requirements on Client Development Challenge

Table 2.2: Weighting of RIA technology criteria

Table 2.3 shows the summarized rates for the proved technologies, weighted with the factors above. With an overall score of 8.0, Flex seems to be best suiting, followed by Java Applets (6.6) and Ajax (4.7).

---

<sup>16</sup>Apache Software Foundation: <http://www.apache.org>

Criteria	Weight	Ajax	Flash/Flex	Java
Graphical Richness	25%	3	10	5
Container/ Engine Footprint	5%	10	8	4
Application Download	5%	10	6	6
Audio / Video Support	5%	4	10	8
Consistency on Different Computing Environments	25%	3	9	8
Server Requirements	5%	5	5	5
Plugin / Runtime Requirements on Client	5%	10	8	7
Development Challenge	5%	3	8	10
Security Concerns	10%	5	5	5
Cost	10%	8	5	8
	100%	<b>4.7</b>	<b>8.0</b>	<b>6.6</b>

Table 2.3: Comparison of RIA technologies



## 3 Developing the Web Modeling Framework

A Framework is to be developed that allows creating complete diagram editors with the specification of diagram types comprising the diagram elements. This chapter deals with the main challenges when creating a generic diagram visualization and manipulation framework. First, a requirements analysis (Section 3.1) defines features that have to be covered by the framework. Then, basic usage of the chosen implementation technology is explained (Section 3.2).

Based on that, the framework is developed upon two functionality parts (Figure 3.1): The *Diagram Type Definition Mechanism* (Section 3.3) allows users of the framework to define diagram types by specifying the diagram elements, properties, and rules. And the *Generic 2D Diagram Editing Functionality* (Section 3.4) provides a common base structure for diagram elements and user interface components for diagram editing. Finally, the utilization of the created framework is explained (Section 3.5)

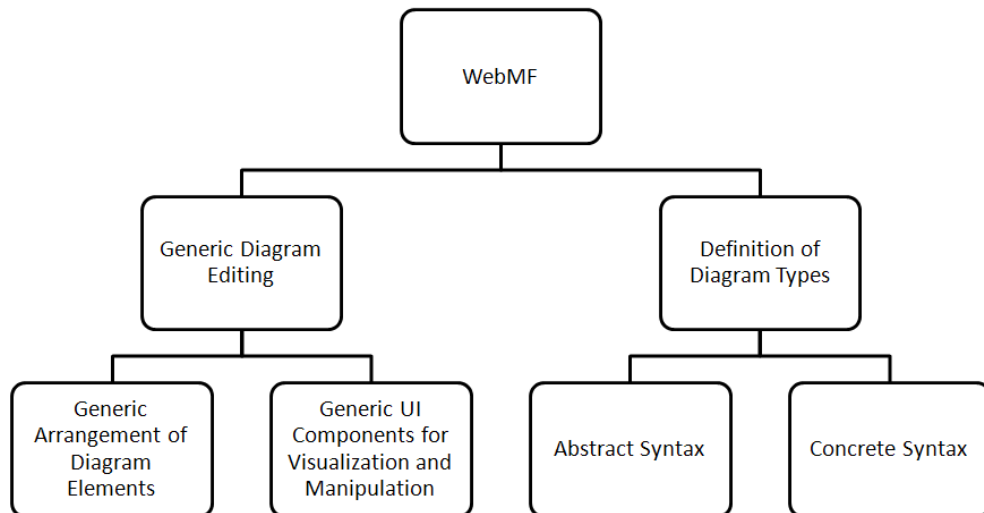


Figure 3.1: Framework Functionality

### 3.1 Requirements

The framework has to provide reusable components for the visualization and manipulation of two-dimensional diagrams. It has to handle the UI actions for editing the diagram elements (add, delete, connect, move, select, and property editing). Furthermore, it has to persist a model of the currently displayed diagram and allow to import and export this persistent model (Figure 3.2).

The functionality is summarized in requirements catalog (Table 3.1): “Must Have”-functions are seen essential to provide basic editing, “Nice to have”-functions are com-

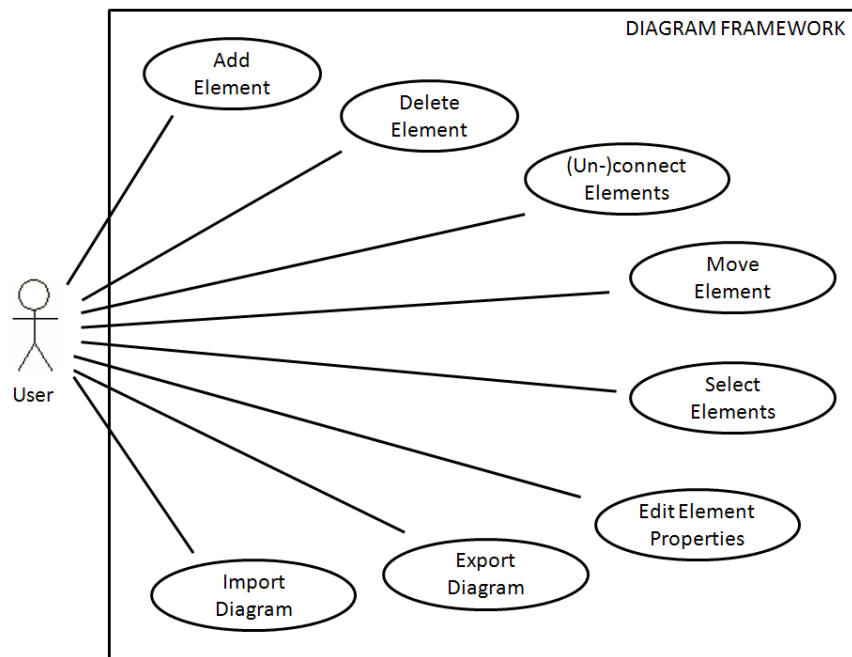


Figure 3.2: Use cases of the WebMF

plement functionality and usability improvements. The user interface functions are equivalent to basic functions of common modeling tools [4]. Details are described in the following.

#### 3.1.1 Visualization and Manipulation of Diagrams

The use of the editor should be as simple as possible. The user should be able to start modeling without any introduction by just using a few core actions. Diagram creation can be allowed in a simple but effective way by providing a diagram area with drag-and-drop, a tool bar, a properties view, and a selection mechanism (described in the following). Further, the framework should be extensible and should allow to add further functions or components for diagram specific demands.

##### Diagram Area With Drag-and-Drop

The visual elements are shown on a diagram area. The framework has to provide drag-and-drop mechanisms for different user interactions:

- For creating an element, the user has to drag a notation element from the tool bar into the diagram area.
- For moving an existing node, the user has to drag-and-drop a node inside the diagram.
- For (dis-)connecting an edge from/to a node, the user has to drag the edge endpoint onto/out-of the node.

##### Node and Edge Elements

A diagram can be modeled as graph that consists of a set of nodes and edges. The following constructs are sufficient to visualize most diagram types:

Category	Description	Must Have	Nice To Have
Diagram	Display nodes, subnodes, and edges	x	
Diagram	Move nodes (drag-and-drop)	x	
Diagram	Connect edges to nodes (drag-and-drop)	x	
Diagram	Delete nodes, subnodes and edges (DEL key)	x	
Diagram	Edges that connect two nodes	x	
Diagram	Edges that connect more than two nodes		x
Diagram	Connection of edges on different points of nodes	x	
Diagram	Drag and Drop usability: different mouse cursors		x
Diagram	Select element (highlight)	x	
Diagram	Hover highlighting		x
Diagram	Different arrow types for edges		x
Diagram	Displaying property labels in nodes	x	
Diagram	Displaying images in nodes		x
Tool Bar	Add new nodes/edges from toolbar (drag-and-drop)	x	
Tool Bar	Show icons of elements in toolbar		x
Properties Editor	Show properties of selected element	x	
Properties Editor	Edit properties of selected element	x	
Import/Export	Import/export interface for browser sandbox	x	
Import/Export	Import/export to file system		x

Table 3.1: Requirements catalog for WebMF

### 3 Developing the Web Modeling Framework

- *Nodes and Subnodes*: Normal (Top-level) nodes are located with absolute dimensions in the diagram area. Subnodes are nodes that are included in a (parent) node.
- *Edges*: Edges connect exactly two nodes with a straight line, one source node and one target node. Optionally, arrows can be specified.

The user must be able to select each element in the diagram area.

#### Tool Bar

A tool bar has to allow to add new nodes and edges by dragging an element from the tool bar to the diagram area. It has to show all node and edge types that can be added to the diagram.

#### Selection Mechanism

A selection mechanism has to make sure that at most one diagram element (node or edge) selected at a time: When the user just clicks (no drag-and-drop operation) on an element it has to be selected. Optionally, elements have to be highlighted in an other way, when the user moves the mouse over it (hover).

#### Properties View

If there is an element selected in the diagram area, a properties view has to allow for viewing and editing the properties of this selected element.

### 3.1.2 Specification of Diagram Types

The framework has to provide a mechanism which allows the simple specification of a specific diagram type. This mechanism has to cover following aspects:

- Specification of all supported element types with their properties, connectivity of these elements, and possible subnodes of node elements.
- Specification of the visual appearance of each diagram element: This view specifications should allow to display the elements' properties in various forms including graphical shapes, boxes, images, etc.

A declarative form in an XML or similar format of these specifications is preferable.

### 3.1.3 Import and Export Interfaces

The framework has to provide importation and exportation of diagram models. Therefore, the model data has to be serialized to an XML format.

For the integration of a framework based editor into TUWEL or another web application, the framework has to provide an interface for exchanging data within the browser's sandbox. This could be handled via JavaScript and / or HTTP requests.

## 3.2 Implementation Technology: Adobe Flex

In Section 2.3.3 Adobe Flex was chosen for implementing the Modeling Tool. Basic concepts of this technology are described in the following.

### 3.2.1 Visual Components

All visual components (Figure 3.3) of WebMF presented in the following are based on Flex’s `UIComponent` component. This component supports adding child components and usage of the drawing API. The `UIComponentNodeView` and `UIComponentEdgeView` view base classes directly extend this component. **Container** components further control the layout characteristics of the child components, three types are used. The **Canvas** container allows absolute positioning of child elements and is used in the diagram MVC view `DiagramView`. The **Box** container lays out its children vertically or horizontally. A vertical **Box** is used for the `BoxNodeView` view base class. The **Panel** container acts like a **Box**, but additionally shows a title bar.

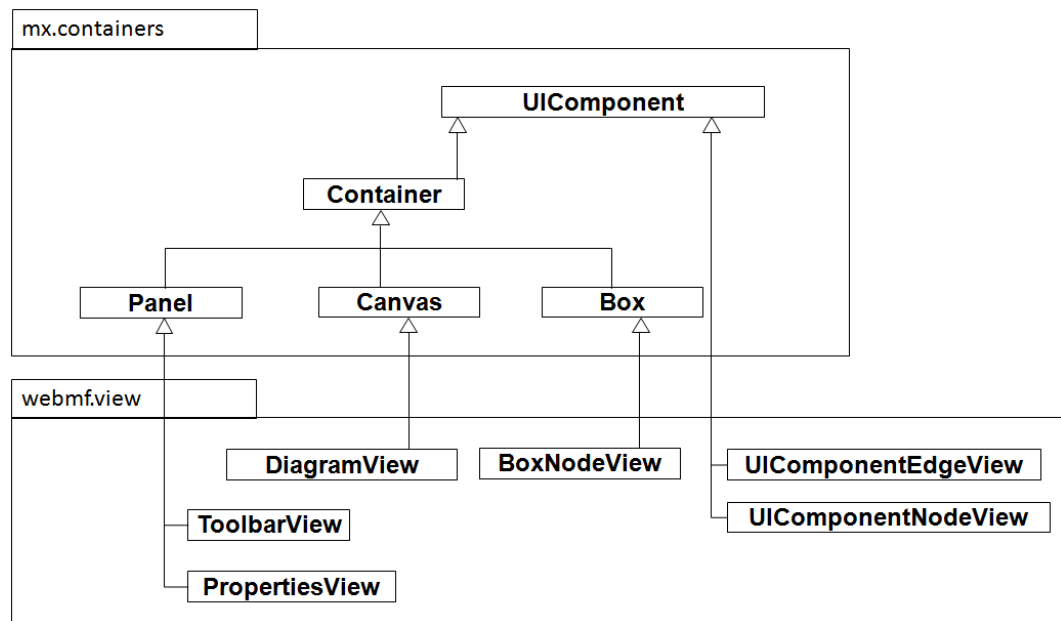


Figure 3.3: Container components of the WebMF

### 3.2.2 Model-View-Controller Pattern

Model-View-Controller (MVC) [9] is a software pattern that is applied to separate data-, presentation- and control-logic. When each part acts independently from each other, one part can easily be replaced or changed:

- The model contains state and/or data, optionally some application logic.
- The view renders the model’s data and/or state, and requests for updates.
- The controller interprets the user’s input and maps it to the model.

However, it is not always necessary to separate the controller from the view. In Flex applications the controller usually is included in the view <sup>1</sup>.

In this chapter, it is very important to distinguish between the MVC and meta-modeling notions of “models”:

<sup>1</sup>The Adobe Flex technology was chosen in Chapter 2.3. Books (e.g. [46]) and the API reference (<http://livedocs.adobe.com/flex/2/langref/package-summary.html>) help to understand the specific constructs and source code of implementation chapters

- “Model” in terms of metamodeling stands for “Metamodeling layer M1”.
- “Model” in terms of MVC stands for “Model part of Model-View-Controller”. For clearness, the MVC layers are referred to as “MVC model”, “MVC view”, and “MVC controller”. The package structure and the class names of the created framework always relate to the MVC meanings.

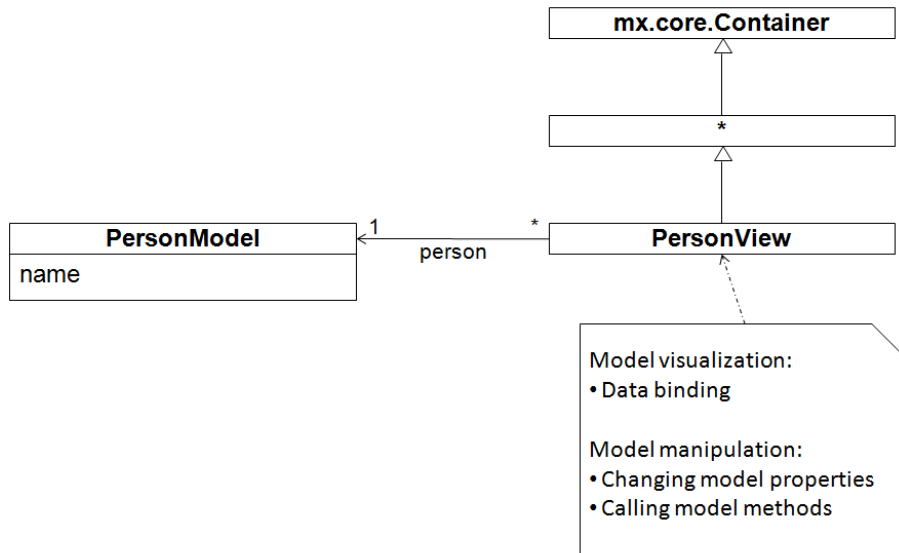


Figure 3.4: MVC Pattern in Flex

An example will demonstrate MVC implementation in Flex (Figure 3.4). The example allows to view and edit instances of a “Person” entity, which has only one “name”-property. A model class (**PersonModel**) holds the data. The view class **PersonView** is derived from a UI container component (a visual element that lays out visual child elements) and is used for visualization and manipulation. Details of the implementation are explained in the subsections.

#### Model

A MVC model can be specified with an ActionScript 3 class using common object oriented programming techniques (Listing 3.1). All properties of a person are declared as instance variables in a class named **PersonModel**. The `[Bindable]` metadata tag (similar to JAVA annotations) causes that changes to this variable are observed at runtime. When this variable is used in a binding (a mechanism for setting variables dynamically to the current value of “bound” variables), this binding is updated.

Listing 3.1: MVC Model in Flex, `mvc/model/PersonModel.as`

```

1 package mvc.model
2 {
3     /**
4      * Person MVC Model
5      *
6      */
7     public class PersonModel
8     {
9         [Bindable]
10        /**
11         * The person's name

```

```

12         */
13         public var name:String;
14
15         public function PersonModel()
16         {
17         }
18
19     }
20 }

```

## View & Controller

A MVC view can be specified using a Flex MXML Component. MXML components allow to specify a user interface in a declarative way using an XML dialect. An MXML component is equivalent to a class defined with ActionScript. Like ActionScript classes, the MXML code is compiled to bytecode. Following rules are applied:

- Each MXML component (\*.mxml file) declares a class. This class extends the class specified in the XML root tag.
- The name of the \*.mxml file specifies the class name, the location in the folder structure specifies the package (A file named MyComponent.mxml in src/pack/-subpack is treated as class MyComponent in package pack.subpack).
- Classes in other packages can be accessed via namespace declarations.
- Commonly, an XML tag is treated as instantiation of a class (class name = tag name, package = namespace) as a member variable. When an “id” attribute is specified, the attribute value is used as variable name.
- If this class is visual and its parent element is a container, than it is added as a child in the layout of the parent element.
- Attributes of an XML tag can specify properties (public instance variables or getter/setter), or event handlers of this instance.

Listing 3.2: MVC View in Flex, mvc/view/PersonView.mxml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <mx:Panel
3      xmlns:mx="http://www.adobe.com/2006/mxml"
4      xmlns:model="mvc.model.*"
5      title="A person">
6
7      <!-- A reference to the MVC model that is visualized and manipulated -->
8      <model:PersonModel id="person" />
9
10     <!-- Data Binding of the name property of the MVC model (curly brackets) -->
11     <mx:TextInput
12         id="nameInput"
13         text="{person.name}" />
14
15     <!-- Manipulation of the model in order to UI events (MouseEvent.CLICK event
16         handler) -->
17     <mx:Button
18         label="Change Name"
19         click="person.name = nameInput.text" />
20 </mx:Panel>

```

Listing 3.2 shows the view definition of the MVC example:

- It creates a class `PersonView` in `mvc.view` that extends the `mx.Panel` component. The component will be rendered as a container with a title bar (line 2).
- Namespace `mx` references standard MXML components (line 3) and namespace `model` references all classes in the `mvc.model` package (line 4).
- The property `title` of the panel is set. The text value is shown in the panel title bar (line 5).
- An instance variable `person` references the MVC model (line 8).
- A text input component (`nameInput`) is added to the panel. The text always shows the actual value of the name property of the person instance (property binding, lines 11-13).
- A button control is added to the panel. When the button is clicked, the `name` of the `person` (the MVC model) is set to the actual value of the name input control (lines 16-18).

## 3.3 Definition of Diagram Types

The framework has to provide a language with which framework users can define diagram types. This language consists of the specification of the abstract syntax (AS) and the concrete syntax (CS) <sup>2</sup>: The *abstract syntax* defines the concepts of the language and their relationships. The *concrete syntax* defines the physical appearance of the language. In the case of a graphical language like WebMF that means the graphical appearance of the language concepts.

The presented approach consists of two languages (Figure 3.5): the Stencil Set DSL and the Stencil View DSL.

The *Stencil Set DSL* covers the abstract syntax. It allows for specifying a set of notation elements, properties that can be set for these elements, and rules that define how elements can be combined.

Complementary, the *Stencil View DSL* covers the concrete syntax. It provides a universal and powerful language for the definition of the visual representation of the elements by providing the Flex UI components to build custom MVC views.

With this architecture, the framework provides support for a variety of diagram types. It promises simplicity in the definition of diagram types and avoids complicated mapping of the notation elements to the abstract syntax, since the models created by the editors simply contain the spacial information and the property values of the diagram elements (defined in Section 3.4).

### 3.3.1 Stencil Set DSL

The Stencil Set DSL covers the abstract syntax of the WebMF DSL. The language is inspired by the “Stencil set” concept of the Oryx Editor (Related Work, Chapter 6.1). It allows for specifying a set of notation elements, properties that can be set for these elements, and rules that define how elements can be combined.

Concretely, a stencil set specifies structure and properties of the notation elements of a specific diagram type. It defines the different nodes and edge types, and their property

---

<sup>2</sup>The notions are introduced in Chapter 2.1

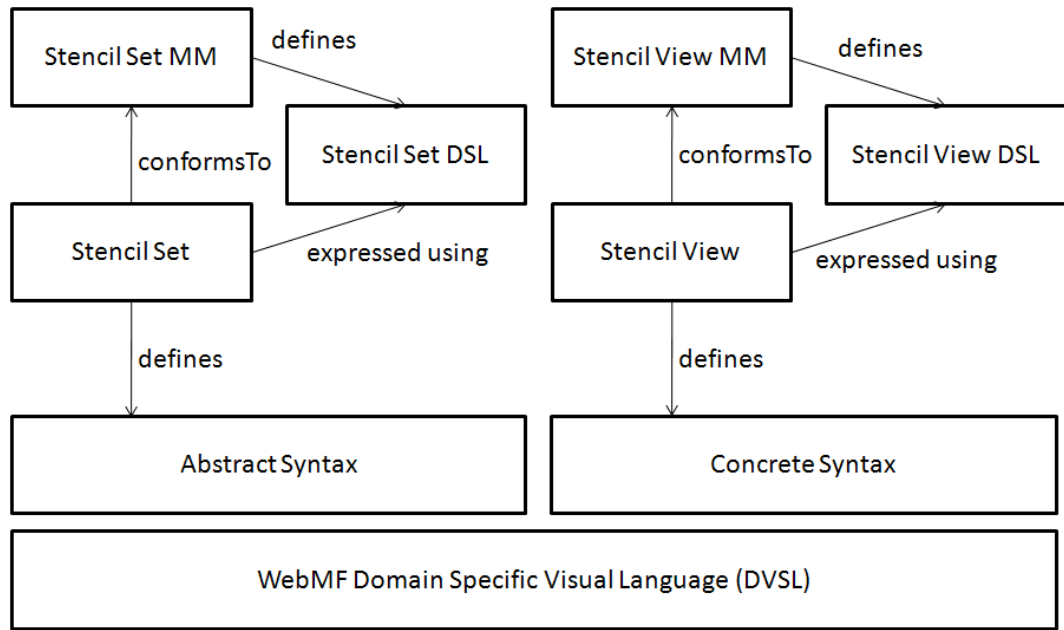


Figure 3.5: Language specification in WebMF

fields (which can be edited in the properties view). It also defines where elements can be created (or nested) and to which nodes or edges they can be connected. And it defines for each notation element, which Stencil View definition is to be used for the rendering.

For defining the metamodel and the DSL, WebMF makes use of the Flex MXML component mechanism (see Section 3.2.2): All elements of the stencil set metamodel are defined as Flex classes. Since every defined class can be instantiated with a tag in an MXML component, this provides the concrete syntax for defining stencil set models.

The modeled classes are shown in Figure 3.6:

- The **StencilSet** class includes a collection of stencils. Stencils can be retrieved by calling `getStencilByName()` or using the `stencils` collection.
- The **Stencil** class defines the characteristics of an element.
  - The `stencilName` is used as unique identifier inside a stencil set.
  - The `type` indicates if this stencil is a “node” or an “edge”.
  - The `stencilViewClass` references the view class that is used to render elements of this element type. The referenced class is defined via the Stencil View DSL described later.
  - The `toolbarIcon` references an embedded image which is used in the tool bar.
  - The `properties` collection comprises the property definitions.
  - The `canCreateIn` and `canConnect` collections allow to specify rules for the creation and connection of elements.
  - The `canCreateInNode()` operation uses the defined creation rules and evaluates if the specified create operation is allowed.
  - The `canCreateInNode()` operation uses the defined connection rules and evaluates if the specified connect operation is allowed.

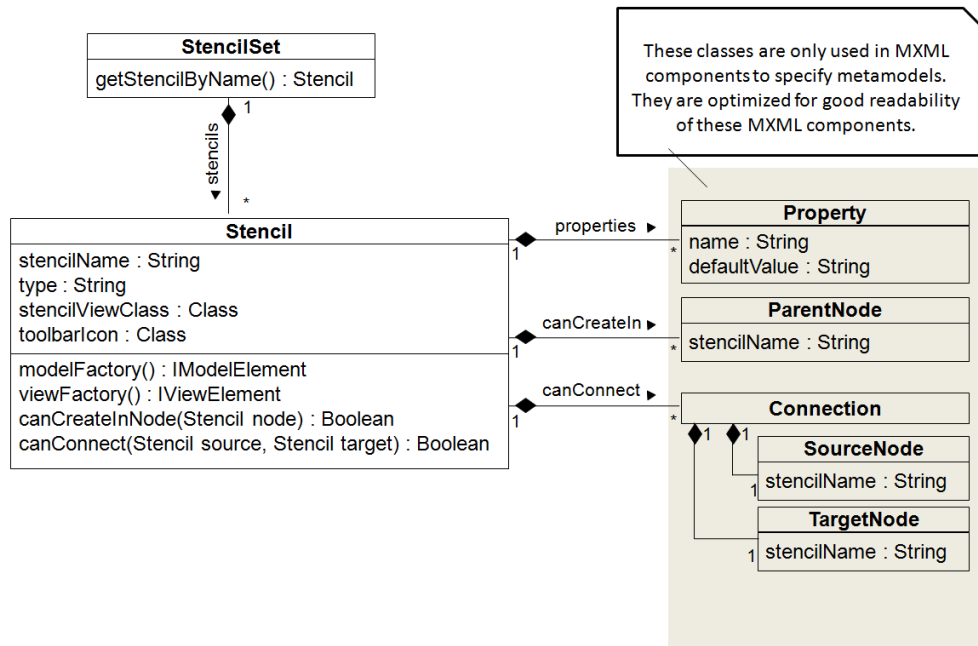


Figure 3.6: Classes for Stencil Set Specification in WebMF

- A property definition (**Property**) consists of the property name and a default value. Possible parent nodes (**ParentNode**) are referred by their stencil name. Connection rules (**Connection**) consist of a source and a target node type, each referred by their stencil name.

These classes are designed in such a way that they can be easily used inside MXML components for the descriptive definition. This allows quick and clear specification of stencil sets. The concrete definition of a stencil set is described in the following. In Section 4.3 provides detailed information about how to create stencil sets.

A template for the definition of the core construction of a stencil set is shown in Listing 3.3: The **StencilSet** class is created (line 2) and the **stencils** array is filled with a number of node and / or edge stencils.

A template for the definition of a node stencil is shown in Listing 3.4: The created stencil is identified by the stencil name “MyNode” and is of type “node” (lines 2-3). The ActionScript class “MyNodeViewClass” is used for visualization (line 4) and the file “myNodeIcon.gif” is used in the tool bar (line 5)<sup>3</sup>. Nodes of this type can be created as top level node (“Diagram” parent) or inside the node “MyParentNode” (lines 7-11), created nodes contain two properties “myProperty1” and “myProperty2” (lines 14-15).

A template for the definition of an edge stencil is shown in Listing 3.5: The created stencil is identified by the stencil name “MyEdge” and is of type “edge”. The ActionScript class “MyEdgeViewClass” is used for visualization and the file “myEdgeIcon.gif” is used in the tool bar. Edges of this type can connect nodes of type “NodeA” (source) to nodes of type “NodeB” (target), created edges contain two properties (“myProperty1” and “myProperty2”).

<sup>3</sup>The file has to be embedded in the project properties (Project → Properties; Category “Build Path”; Tab assets)

Listing 3.3: Core construction of a stencil set component

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <StencilSet
3      xmlns="webmf.model.*"
4      xmlns:mx="http://www.adobe.com/2006/mxml">
5      <stencils>
6
7          <!-- add stencils here -->
8
9      </stencils>
10 </StencilSet>

```

Listing 3.4: Template for the definition of a node stencil in a stencil set component

```

1      <Stencil
2          stencilName="MyNode"
3          type="node"
4          stencilViewClass="{MyNodeViewClass}"
5          toolbarIcon="@Embed(source='assets/myNodeIcon.gif')">
6
7          <canCreateIn>
8              <ParentNode stencilName="Diagram" />
9              <ParentNode stencilName="MyParentNode" />
10             <!-- ... -->
11          </canCreateIn>
12
13          <properties>
14              <Property name="myProperty1" defaultValue="defaultValue" />
15              <Property name="myProperty2" defaultValue="defaultValue" />
16              <!-- ... -->
17          </properties>
18      </Stencil>

```

Listing 3.5: Template for the definition of an edge stencil in a stencil set component

```

1      <Stencil
2          stencilName="MyEdge"
3          type="edge"
4          stencilViewClass="{MyEdgeViewClass}"
5          toolbarIcon="@Embed(source='assets/myEdgeIcon.gif')">
6
7          <canConnect>
8              <Connection>
9                  <source>
10                     <SourceNode stencilName="NodeA" />
11                 </source>
12                 <target>
13                     <TargetNode stencilName="NodeB" />
14                 </target>
15             </Connection>
16             <!-- ... -->
17          </canConnect>
18
19          <properties>
20              <Property name="myProperty1" defaultValue="defaultValue" />
21              <Property name="myProperty2" defaultValue="defaultValue" />
22              <!-- ... -->
23          </properties>
24      </Stencil>

```

### 3.3.2 Stencil View DSL

By use of the Stencil View DSL, the concrete syntax of the single graphical elements can be specified. It is a universal and powerful language for the definition of the visual representation of the elements by providing the Flex UI components to build custom MVC views.

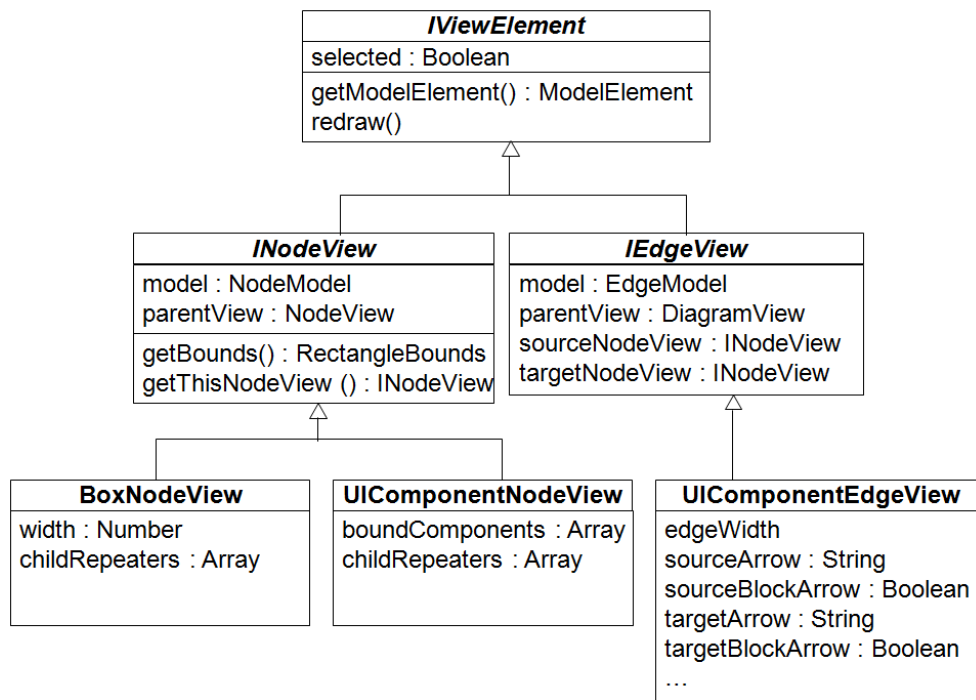


Figure 3.7: Interfaces and view base classes for stencil views

The notation elements can be build upon standard Flex UI components, they only have to implement the **INodeView** respectively **IEdgeView** interface (Figure 3.7). Hence, the metamodel is represented by the Flex Visual Component API classes, and the DSL is represented by the Flex MXML component language. The interfaces are described in the following:

- The common characteristics of node views and edge views are abstracted in the **IViewElement** interface. Each view element contains a reference to the underlying model (`getModelElement()`). It also can be selected and unselected (`selected`). The `redraw()` operation updates the view when the model is changed.
- Node views visualize a node model (`model`). The `model` reference can be used for retrieving properties (`model.getProperty("propertyName")`) and children (`model.getChildren("childStencilName")`) of a node. The `thisNodeView()` method just returns a reference to `this` object (used for binding `this` object; binding `this` directly would result in runtime errors).
- Each node view is visualized inside a parent node view (`parentView`). Further, custom node views specify their bounds (position and size) by implementing the `getBounds()` method.
- Edge views visualize an edge model (`model`). The `model` reference can be used for retrieving properties. Each edge view is visualized inside a parent diagram view `parentView`. They connect a `sourceNodeView` to a `targetNodeView` according to the edge model. If one of these properties is null, this means that this side is not connected to a node.

## View Base Classes

Node views and edge views can be created by implementing the interfaces `INodeView` or `IEdgeView` (Figure 3.7). To ease the development, the WebMF provides three view base components which have common purposes and can be derived for creating views:

The `BoxNodeView` component is based on the Flex `Box` layout container and lays out its children vertically. It allows adjustment of the size to the size of its contents. The boundary is rectangular and can have rounded corners. This type of node is widely spread across many diagram types. The width of the container can be specified, the height is adjusted dynamically to the contents. All bindings of the component and its children are recognized automatically.

The `UIComponentNodeView` component is based on the Flex `UIComponent` container. The layout can be specified using absolute positioning of child components, and adding them manually (`UIComponent.addChild()`) to the container. Components including bindings have to be added to the `boundComponents` array so that they are recognized. It is very flexible and allows for specifying the appearance of the node with the 2D graphics engine, but also requires more implementation effort.

Both node view base components provide an array for registering repeater elements (`childRepeaters`)<sup>4</sup>. All bindings included in these repeaters are recognized automatically.

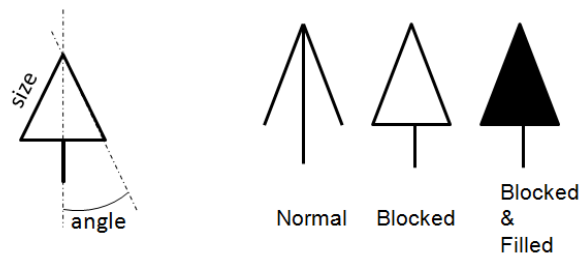


Figure 3.8: Arrow dimensions and styles

The `UIComponentEdgeView` component is based on the Flex `UIComponent` container. It displays a straight line from source to target and, optionally, various arrows. The arrow size, angle, and style can be set for both ends of the edge separately (Figure 3.8).

Because more than one edge can connect the same pair of nodes, it is necessary that edges can be connected at different anchors of the node's boundary. Since the size of nodes can change, absolute positions of endpoints are not practicable. The `UIComponentEdgeView` endpoint anchor positions are defined with percentages of the nodes' size. The example (Figure 3.9) shows two nodes *A* and *B* which are connected by edge *e*. The relative endpoint anchor *a* is located on 70% of the node's width and 15% of the node's height. The real edge endpoints are located at the intersection points  $i_a$  and  $i_b$  of the nodes' boundaries and the connection of the two endpoint anchors. The anchor position percentages are calculated when the user drops the edge endpoint onto a node (see Section 3.4.2).

The reader may refer to the API documentation<sup>5</sup> for a comprehensive description of the WebMF view base components.

<sup>4</sup>Repeaters are used for repeating a set of UI components. The number of repetitions corresponds to the size of the data provider.

<sup>5</sup>API documentation: <http://web.student.tuwien.ac.at/~e0427416/webmf/docs/1.0/>

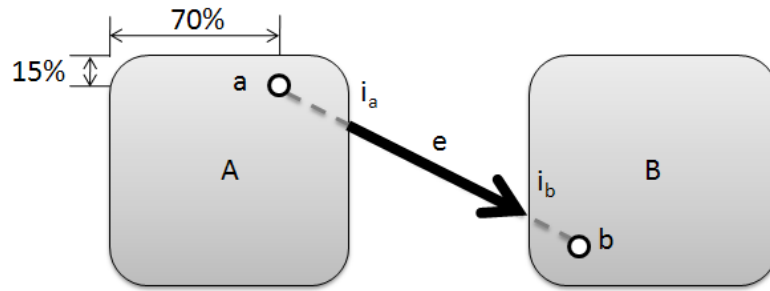


Figure 3.9: Calculation of edge endpoint positions

## 3.4 Generic Diagram Editing

Generic diagram editing requires a common data structure for different diagram types and visual UI components that can manipulate this common data structure.

### 3.4.1 Common Data Structure for Arranging Diagram Elements

For the specified requirements, three elements are needed to describe the actual state of a displayed diagram:

- A *diagram* contains a number of elements (nodes and edges). Elements can be added to or removed from the diagram. The diagram conforms to a stencil set that defines structural information for the distinct types of these elements.
- A *node* is positioned with absolute (horizontal and vertical) coordinates. It conforms to a stencil of the stencil set of the diagram in which it is included. It also holds the values of a number of properties (which are defined in the stencil set). A node can have a number of subnodes which can be added or removed.
- An *edge* can connect a source node to a target node. If an endpoint of the edge is connected to a node, the connection position (anchor) is stored as percentage value (see Section 3.3.2). Otherwise the endpoint is positioned with absolute coordinates in the diagram. Like a node, an edge conforms to a stencil of the stencil set of the diagram and holds a number of properties.

These aspects are modeled in Figure 3.10.

- The common characteristics of edges and nodes are abstracted through the `ModelElement` class. Each element references the stencil type (`stencil`) and has a collection <sup>6</sup> for the properties (`properties`).
- Class `NodeModel` specifies numeric values for the horizontal and vertical position in the diagram. Parent/child relationships are represented by the `parent` and `children` associations. The `createChild()` operation creates a child node of the specified type (`stencil`). The `removeChild()` operation removes the child node specified by the `childNode` parameter. The `remove()` operation deletes all child nodes and removes this node from its parent. All children of a specific type can be retrieved with `getChildren()`.

<sup>6</sup>`flash.utils.Dictionary`: A Flex collection data type with name-value pairs, similar to JAVA Map

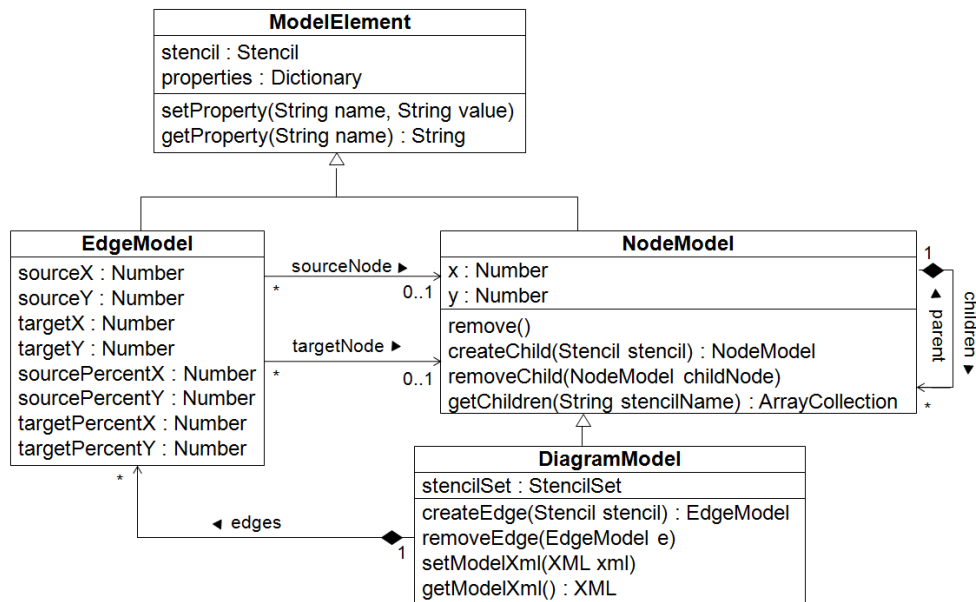


Figure 3.10: Metamodel of diagram elements in WebMF

- Class **EdgeModel** has two node references: `sourceNode` and `targetNode`. The properties with percentages are used when the according node is set to an existing node, the other (absolute) properties are used otherwise.
- Class **DiagramModel** inherits from class **NodeModel**. The horizontal and vertical coordinates are ignored. Additionally the diagram has a reference to — and conforms to — the related `stencilSet` and provides the import (`setModelXml()`) and export (`getModelXml()`) of the serialized XML model, according to the referenced stencil set.

### 3.4.2 Visual UI Components for Visualization and Manipulation

The framework has to provide three UI Components for generic diagram visualization and manipulation: A *diagram* that contains the visual diagram elements, a *tool bar* that allows adding new elements to this diagram, and a *properties editor* that allows for editing the properties of the currently selected diagram element (Figure 3.11).

The *diagram* is an area that acts as a container for all element views (nodes and edges). The layout of this container is absolute, which means that the position of the children can be specified explicitly. Within this area element views can be moved, connected, or selected. Only zero or one element can be selected in the diagram at a time. The **DiagramView** extends the **Canvas** component, which realizes diagram functionality. It also handles drag-and-drop events for the manipulation of the diagrams.

The *tool bar* shows controls for all stencils of the current stencil set. These controls are enriched with drag-and-drop support, so that they can be added to the diagram.

The *properties editor* builds upon the **DataGrid** component and displays two columns, one for the property names and one for the property values. If an element is selected in the diagram, all its properties are shown. The values can be edited in the control. The data in the model is then updated accordingly.

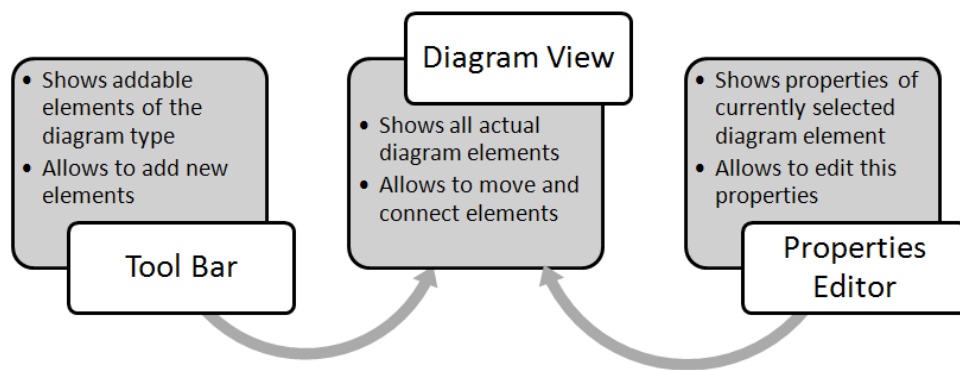


Figure 3.11: UI components of WebMF

### Manipulating the Generic Structure

Dragged item	Target	Drag Over	Drop
Node element from toolbar	Diagram area	Show symbol „create“ / „not allowed“	Add node (top level)
Edge element from toolbar	Diagram area	Show symbol „create“ / „not allowed“	Add edge (top level)
Node element from toolbar	Node	Show symbol „create“ / „not allowed“	Add node as subnode
Edge element from toolbar	Node	Show symbol „create“ / „not allowed“	Add edge (top level)
Existing Node	Diagram area	Show symbol „move“	Move node
Edge endpoint	Diagram area	Show symbol „link“	Move edge endpoint
Edge endpoint	Node	Show symbol „connect“	Connect edge endpoint to node

Figure 3.12: Overview over drag-and-drop functionality

All user interactions except the deletion of elements (keyboard DELETE key) are done with drag-and-drop. Table 3.12 shows, when a certain operations are invoked. A drag-and-drop operation starts when the user moves the mouse cursor over a user interface element that supports dragging (the dragged item), and presses the left mouse button. When the user drags the dragged item over another user interface element that function as a drop target (the target), the mouse cursor indicates whether this drag-and-drop operation is allowed (drag over). This depends on the combination of dragged item, target, and on the certain involved notation elements. If the operation is allowed and the user releases the mouse button, the operation is executed (drop).

In the following the steps of manipulating the MVC model and the MVC view — invoked by certain user interactions — is described. The MVC views handle the events

that are dispatched by the MVC model elements, and update themselves accordingly.

- *Creating a new node:* When the user drags a node stencil from the tool bar into the diagram area or an other node that allows this node type as subnode, a new node will be created. First, a new `NodeModel` object is instantiated, and its coordinates are set according to the mouse drop position. Then, it is added to the `children` of the `parent` node model. This triggers a `CreateChildModelEvent` event. The parent view (diagram or node) reacts to the creation of the child, creates a node view, binds it to the node model and displays it.
- *Creating a new edge:* When the user drags an edge stencil from the tool bar into the diagram area, a new edge will be created. First, a new `EdgeModel` object is instantiated, and its coordinates are set according to the mouse drop position. Then, it is added to the `parent` diagram. This triggers a `CreateEdgeModelEvent` event. The parent diagram view reacts on the creation of the edge, creates an edge view, binds it to the edge model and displays it.
- *Connecting/Disconnecting Edge Endpoints:* When the user drags an edge endpoint into a node of the right edge endpoint type, the edge will be connected to this node. First, the source/target node of the `EdgeModel` is set to to the new node. The `EdgeModel` is registered to react on `DeleteNodeModelEvent.DELETE` events of the newly connected model (for cascading delete of nodes). Then the relative endpoint position (width/height percentage of the drop mouse position to the newly connected model) is set. An `ConnectModelEvent` is triggered and the related edge view is updated.

In almost the same manner, when the user drags an edge endpoint to an empty location of the diagram area, the edge endpoint will be disconnected: The source/-target node of the `EdgeModel` is unset and it is unregistered from delete events (`DeleteNodeModelEvent.DELETE`) of the previously connected node model. And the absolute endpoint position (according to the drop mouse position) is set for the edge.

- *Deleting a node:* When a node is selected and the user presses the delete key on the keyboard, the currently selected node will be deleted. First, all `children` of the `NodeModel` are recursively deleted. The connected edges of the nodes are informed via the `DeleteNodeModelEvent.DELETE` event, and are unconnected (see above). Then the node model is removed from the parent node (or diagram). This triggers a `DeleteChildModelEvent.DELETE` event. The parent node view removes the child from its layout (the unused object can be garbage collected), and the selection of the parent diagram is set to none.
- *Deleting an edge:* When an edge is selected and the user presses the delete key on the keyboard, the currently selected edge will be deleted. First, the `EdgeModel` is removed from the parent diagram model. This triggers a `DeleteChildModelEvent.DELETE` event. The parent diagram view reacts and removes the child from its layout (the unused object can be garbage collected), and the selection of the parent diagram is set to none.

## 3.5 Framework Utilization

This section demonstrates, how the components of WebMF are utilized to build a specific diagram editor, and how the different functionality parts work together.

### 3.5.1 Design of a Typical WebMF Application

For building a typical WebMF editor, five components are assembled in an application (Figure 3.13). The first two are non-visual. The WebMF Stencil Set DSL and the WebMF Stencil View DSL are used for building a DSVL (called Stencil Set), which then is instantiated in the application. Then a Diagram Model is instantiated and connected with the Stencil Set. Through this connection, the behavior of the Diagram Model in response to user interaction is adapted.

For the user interaction, the three visual components are utilized. A Diagram View is connected to the Diagram Model. Thus, the view shows the graphical representation of this model and initiates manipulation of this model on user interface actions. A tool bar is connected to the Stencil Set and so shows all notation elements of this Stencil Set. These notation elements can be dragged into the Diagram View to create new diagram elements. Finally, a Properties View is connected to the Diagram View and shows the properties of the currently selected element of the diagram (if any) and permits to edit them.

Chapter 4 demonstrates the creation of a sample application step by step.

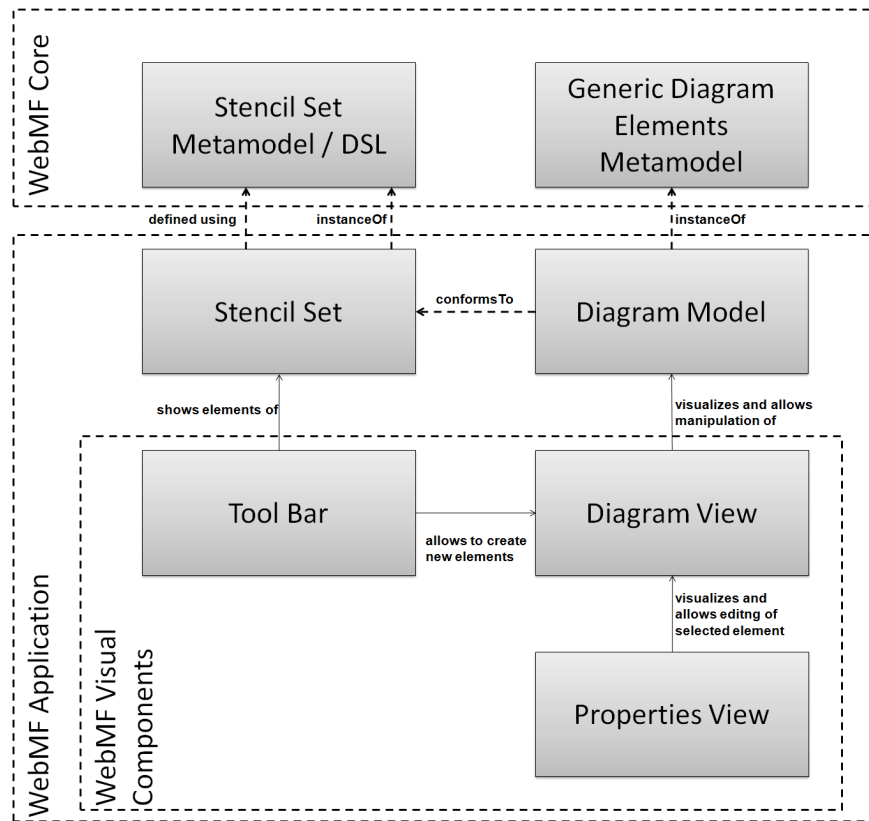


Figure 3.13: Design of a typical WebMF application

### 3.5.2 Interplay of the Functionality Parts

The relationships between diagram types and the concrete diagram elements in a running applications are illustrated in Figure 3.14. It shows the relationships on a class layer as well as on an object layer. A diagram type is described through a stencil set together with its stencils (specified using the DSLs). Concretely, `myStencilSet` includes

the stencils `nodeStencil` and `edgeStencil`.

A diagram model to be visualized and manipulated is described through objects of the `ModelElement` (`DiagramModel`, `NodeModel`, and `EdgeModel`) classes. In this case, the diagram consists of the node `node1` and the edge `edge1`. This diagram is of diagram type `myStencilSet`. This implicates that all children of this diagram relate to notation elements of this diagram type. In this case those are the notation elements `nodeStencil` and `edgeStencil`.

The manipulation of the elements according to the diagram types is implemented in the generic data structures and the generic UI components.

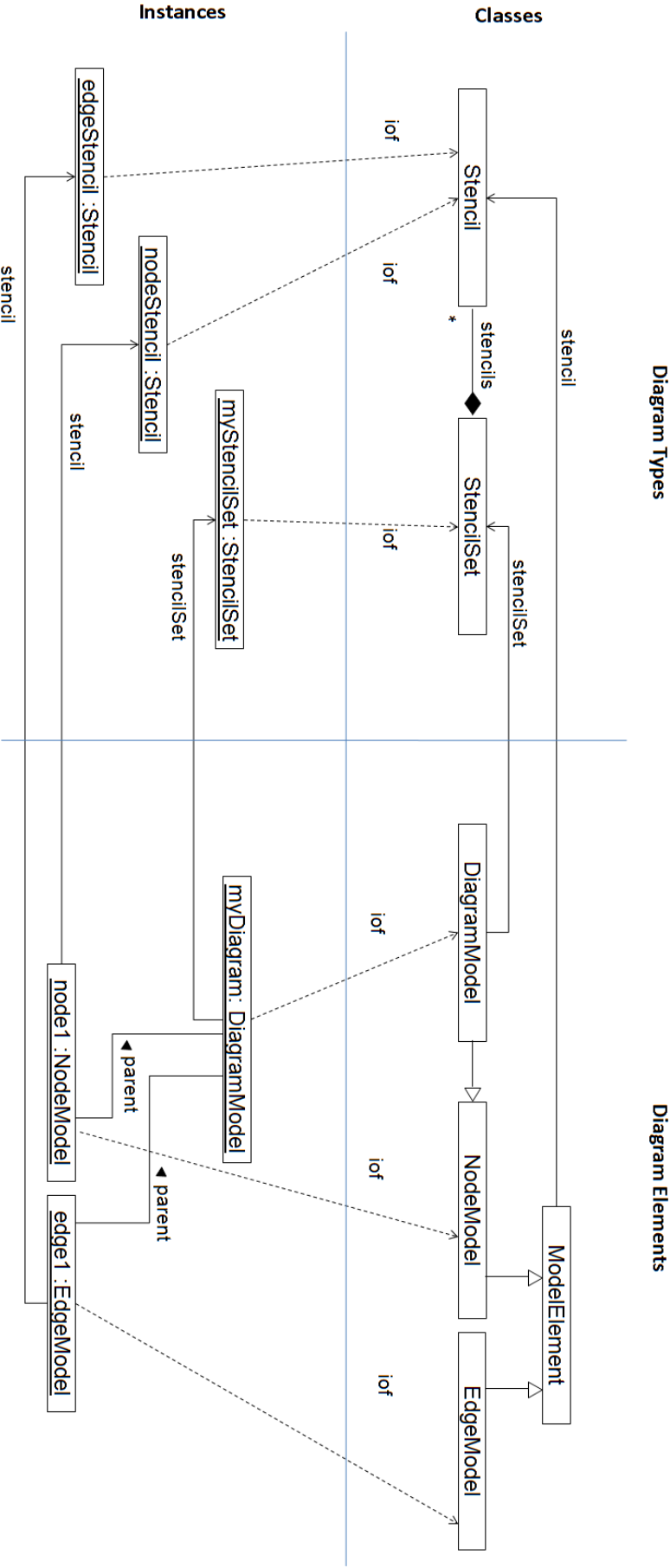


Figure 3.14: Interplay of the functionality parts

## 4 Sample Application

This chapter demonstrates how to use the WebMF to build a specific diagram visualization and manipulation application step by step. First, the Abstract Syntax (Section 4.1) and the Concrete Syntax (Section 4.2) are defined for the editor. Then, a WebMF stencil set is created (Section 4.3) upon these specifications. Finally, an application is created that uses the created stencil set and the generic visual UI components of WebMF (Section 4.4).

There are only a few prerequisites to build this sample stencil set and application. FlexBuilder 3.0 (or higher) with Flex SDK 3.3. (or higher) have to be installed. Also a web browser with the Flash debug plug-in (comes along with the installation of Flex Builder) is needed. Finally, the WebMF library SWC (`webmf.swc`) has to be downloaded.

### 4.1 Abstract Syntax

The sample application domain is called “TaiPan”, and is based on an demonstration example of a GMF Tutorial [14]. The TaiPan application allows for coordinating ports, ships, and items to transport<sup>1</sup>. Ports are identified by their name. From one port a number of other ports can be navigated (routes) and the distances to these other ports are known.

Ships are also identified by their name. They transport a number of item packages with a specific item type and amount. It can be defined, to which ports a ship has to navigate (destinations). All aspects are modeled in the Abstract Syntax (Figure 4.1).

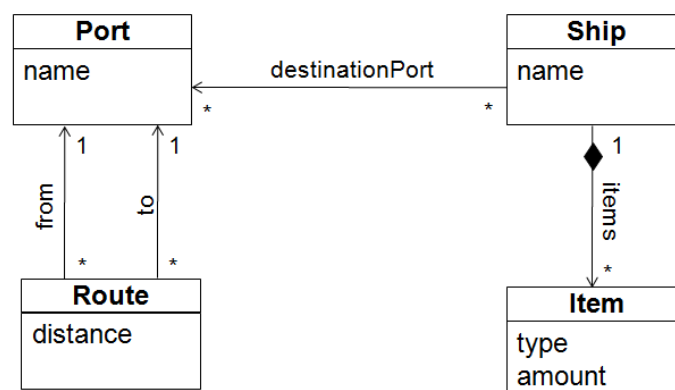


Figure 4.1: Abstract Syntax of the WebMF TaiPan editor

<sup>1</sup>The example varies a bit from the one in the GMF Tutorial

## 4.2 Concrete Syntax

Ships are visualized as rectangles that show their name on the top. Then the different items are listed in horizontal lines. Thereby, one item is shown as a square graphics including the amount of the item. Ports are visualized as ragged shapes. Above this shape the name of the port is shown.

The routes between the different ports are displayed as simple connection lines. And, finally, the destination ports of ships are displayed as blocked arrows from the ship to the port of call.





Name	Concrete Syntax
Ship	
Port	
Route	
Destination Port	

Table 4.1: Concrete syntax of WebMF TaiPan diagrams

## 4.3 Building the Stencil Set Library

Although it's possible to define both stencil set and application logic in a single Flex application, we suggest to create a stencil set library so that it can be used in several applications.

To create the stencil set, we first have to create a Flex library project and add the WebMF library. Then we build the structure of the stencil set component. Finally, we create stencils (view definitions) for each of the required elements.

### 4.3.1 Creating a Flex Library Project

We start with the creation of a flex library project “TaiPanStencilSet” (File → New → Flex Library Project, Figure 4.2) using the Flex Builder.

Inside the library project, we create the following folder (package) structure (Flex Navigator view on the left side):

- src (main source folder)
  - assets
  - taipan

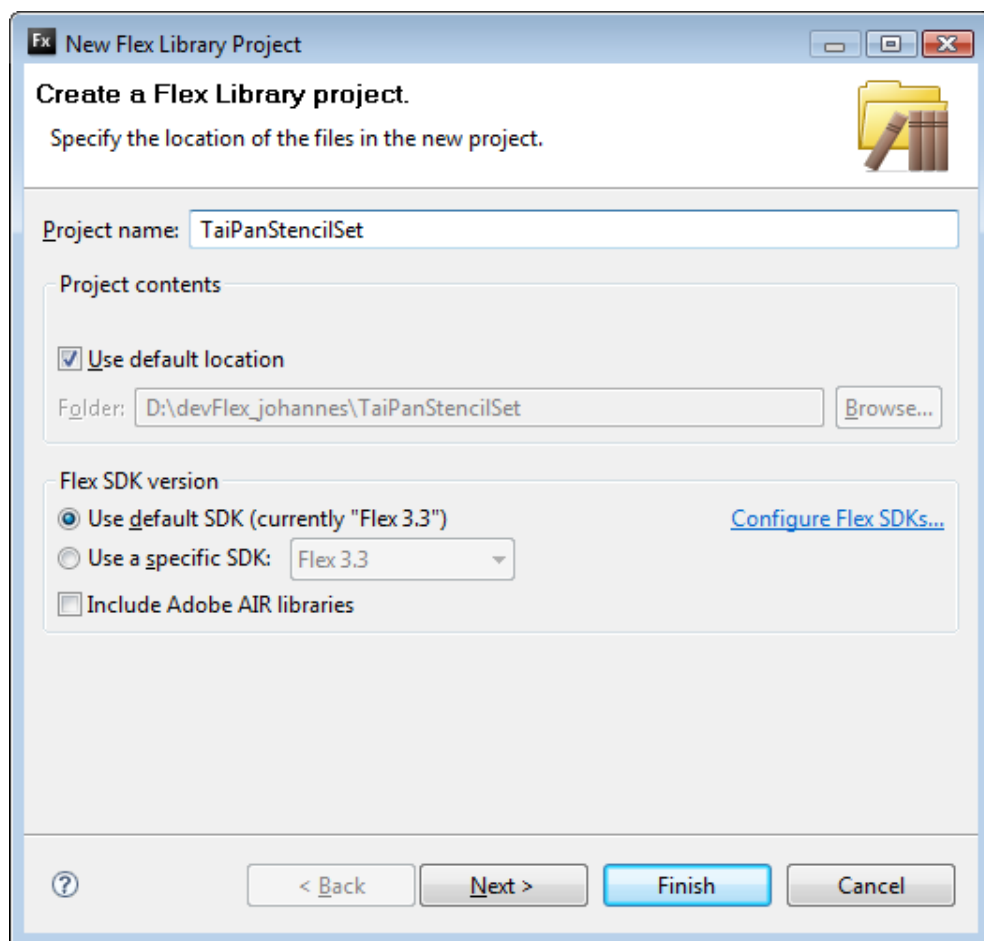


Figure 4.2: Create Flex library project

## 4 Sample Application

- \* stencil
- \* stencilset

Package `taipan.stencilset` will include the TaiPan stencil set and package `taipan.stencil` will include the view definitions. The assets folder will be used for all embedded images in the views.

To make use of the WebMF we have to include it in the Flex library build path of your project: In the project properties (Project → Properties), we choose tab Library path in Flex Library Build Path (Figure 4.3) and add the delivered SWC (`webmf.swc`).

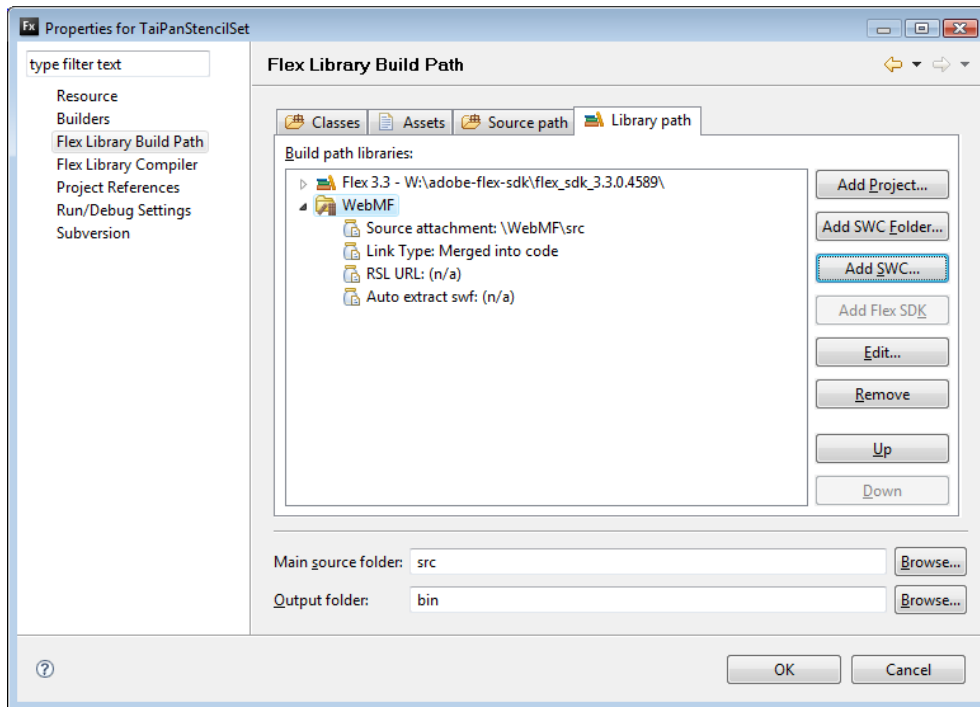


Figure 4.3: Add WebMF library

### 4.3.2 Creating a Stencil Set

To create the stencil set, we create a new MXML component (File → New → MXML Component) in package `taipan.stencilset` with name “TaiPanStencilSet” and specify “StencilSet” as superclass (field “Based on:”, Figure 4.4) <sup>2</sup>.

Listing 4.1: Core construction of a stencil set component

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <StencilSet
3     xmlns="webmf.model.*"
4     xmlns:mx="http://www.adobe.com/2006/mxml">
5     <stencils>
6
7         <!-- add stencils here -->
8
9     </stencils>
10 </StencilSet>
```

<sup>2</sup>We cannot choose the Stencil Set base Component from the framework because Flex Builder only shows visual components in the drag-and-drop list

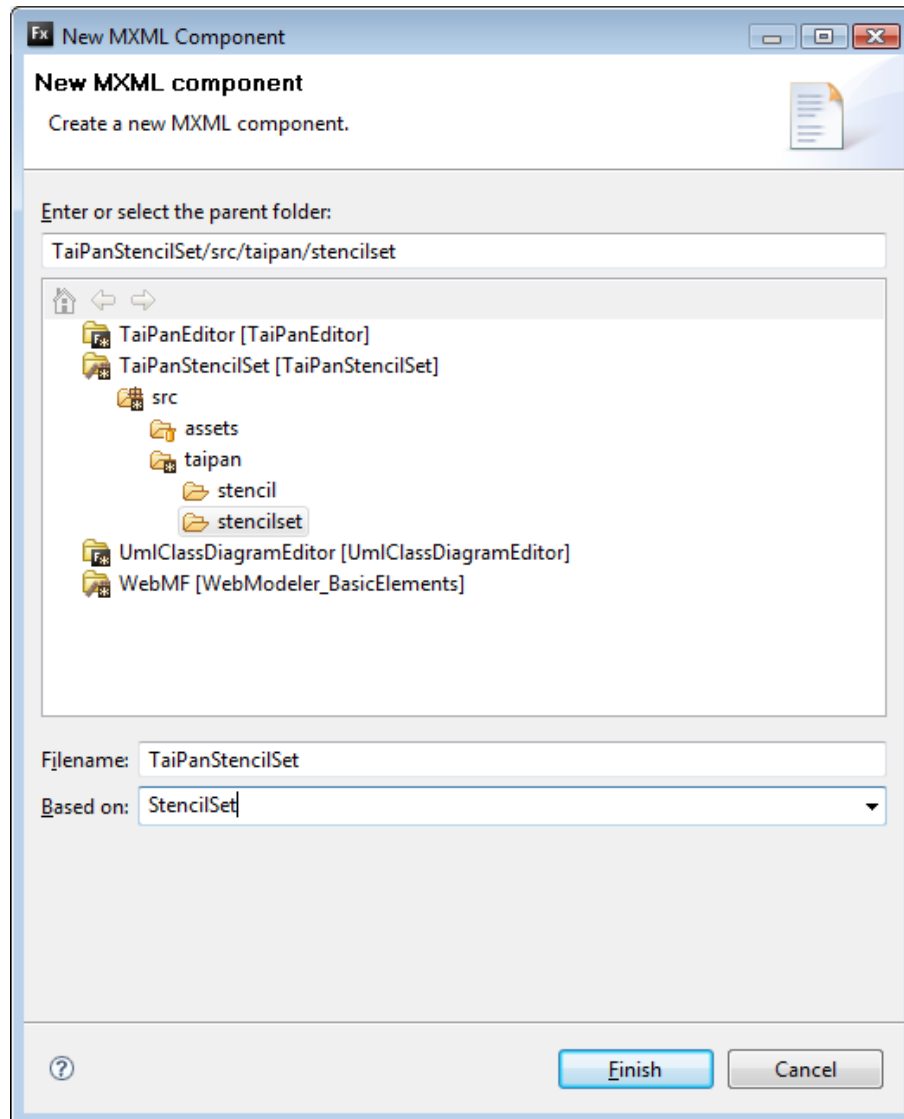


Figure 4.4: Creating stencil set component

## 4 Sample Application

After finishing the wizard, Flex Builder shows an automatically generated MXML file. The XML root tag `StencilSet` specifies the base component for this non-visual component. Listing 4.1 shows the core of the stencil set definition after some adaptations: We change the default namespace to “webmf.model.\*” (line 3)<sup>3</sup> and create a `stencils` tag inside the root tag. Stencil definitions have to be put inside this tag.

Listing 4.2: taipan/stencilset/TaiPanStencilSet.mxml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <StencilSet xmlns="webmf.model.*" xmlns:mx="http://www.adobe.com/2006/mxml">
3   ...
4   <stencils>
5
6     <Stencil stencilName="Ship" type="node" stencilViewClass="{ShipView}"
7       toolbarIcon="@Embed(source='assets/defaultIcon.gif')">
8       <canCreateIn> <ParentNode stencilName="Diagram" /> </canCreateIn>
9       <properties> <Property name="name" defaultValue="Shipname" /> </
10         properties>
11     </Stencil>
12
13     <Stencil stencilName="Port" type="node" ...>
14       <canCreateIn> <ParentNode stencilName="Diagram" /> </canCreateIn>
15       <properties> <Property name="name" defaultValue="A Port" /> </
16         properties>
17     </Stencil>
18
19     <Stencil stencilName="Item" type="node" ...>
20       <canCreateIn> <ParentNode stencilName="Ship" /> </canCreateIn>
21       <properties> <Property name="type" defaultValue="Rum" /> <Property
22         name="amount" defaultValue="1" /> </properties>
23     </Stencil>
24
25     <Stencil stencilName="Route" type="edge" ...>
26       <canConnect>
27         <Connection>
28           <source> <SourceNode stencilName="Port" /> </source>
29           <target> <TargetNode stencilName="Port" /> </target>
30         </Connection>
31       </canConnect>
32       <properties> <Property name="distance" defaultValue="0" /> </
33         properties>
34     </Stencil>
35
36     <Stencil stencilName="ShipDestinationPort" type="edge" ...>
37       <canConnect>
38         <Connection>
39           <source> <SourceNode stencilName="Ship" /> </source>
40           <target> <TargetNode stencilName="Port" /> </target>
41         </Connection>
42       </canConnect>
43     </Stencil>
44   </stencils>
45 </StencilSet>
```

In the TaiPan project we create five stencil definitions (Listing 4.2):

- We specify a unique `stencilName` for every node and edge. Elements are referred by this name, e.g. in creation and connection rules. Additionally, we specify the element type ( “node” / “edge”), the view class and the tool bar icon (lines 6/11/16/21/31).
- In the `canCreateIn` tag we specify, where nodes can be created: `Ship` and `Port` are top-level nodes (can be created in `Diagram`, lines 7/12). `Item` nodes can be created in `Ship` nodes (line 17).

<sup>3</sup>Only necessary for non-visual components while base component is not processed automatically

- In the `canConnect` tag we specify connection rules: `Route` edges can only connect two `Port` nodes (lines 24-25), `ShipDestinationPort` edges can only connect a `Ship` node to a `Port` node.
- In the `properties` tag we specify the property types for the elements (lines 8/13/18/28).

### 4.3.3 Creating Stencil Views

We have to specify only one file per stencil to describe how a specific stencil is displayed, a so-called stencil view. We create a new MXML Component (File → New → MXML Component) “MyNodeView” in package `taipan.stencil` based on the `BoxNodeView`, `UIComponentView`, or `UIComponentEdgeView` view base component. Then we can specify the layout using standard Flex components and the features of the WebMF base components (see elements of the TaiPan project below).

We choose the base component for the stencil views as follows: Simple rectangular node types with automatic child layout can be displayed using the `BoxNodeView` view base class. Other custom node types can be displayed using the `UIComponentNodeView` view base class. All edges can be displayed using the `UIComponentEdgeView` view base class.

All major *aspects* of stencil views are covered by this example. Most *nodes* can be rendered as rectangular container nodes, based on the `BoxNodeView` view base class (ship view and item view). Other nodes can be rendered as custom nodes, based on the `UIComponentNodeView` (port view). *Edges* can be rendered as connecting lines, optionally with specific arrow types, using the `UIComponentEdge` view base component (route view, ship destination port view).

The *views* can automatically show the current values of model properties (ship view, item view, port view). Also, images can be displayed using standard Flex components (item view, port view). Subnodes can be displayed one after the other by using `Repeater` components inside the parent node (item views in ship view).

### Ships and Items

A `Ship` node shall be displayed as rectangular container, which shows the name of the ship and its transported items in one or more vertical rows (Figure 4.1). To show a rectangular container, we use the `BoxNodeView` view base component (Listing 4.3, line 2). Into this component we put a label which shows the name property (line 3) and a `mx:Tile` container for the items (line 4)<sup>4</sup>.

To dynamically show all included items in this container, we use a `mx:Repeater` component which creates an `stencil:ItemView` for each `Item` child (line 5), and register the repeater for automatic update (line 2). We set the `model` that the child view has to render to the current element of the iteration, and set the `parentElement` to the current displayed `Ship` (line 6).

TaiPan `Item` nodes shall be displayed as little images which are overlaid with a label indicating the item amount (Figure 4.1). We use the `BoxNodeView` view base component (Listing 4.4, line 2) and insert a `mx:Canvas` container to put children one upon the other (line 3). To this container, we add the image (line 4) and the text showing the amount (line 5). The result of the two views is shown in Figure 4.5.

<sup>4</sup>The `mx:Tile` container lays out its children in one ore more rows or columns.

Listing 4.3: taipan/stencil/ShipView.mxml

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <BoxNodeView childRepeaters="{[itemRepeater]}" ...>
3   <mx:Label text="{model.getProperty('name')}}" ... />
4   <mx:Tile width="100%">
5     <mx:Repeater id="itemRepeater" dataProvider="{model.getChildren('Item')}}"
6       ">
7       <stencil:ItemView parentElement="{thisNodeView}" model="{
8         itemRepeater.currentItem}" />
9     </mx:Repeater>
10  </mx:Tile>
11 </BoxNodeView>

```

Listing 4.4: taipan/stencil/ItemView.mxml

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <BoxNodeView ... >
3   <mx:Canvas>
4     <mx:Image ... source="@Embed('assets/item.gif')"/>
5     <mx:Label ... text="{model.getProperty('amount')}}" />
6   </mx:Canvas>
7 </BoxNodeView>

```

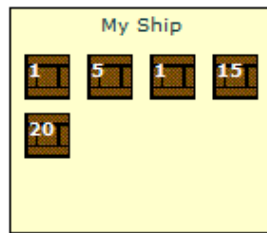


Figure 4.5: Look of a ship with five items and different amounts

### Ports

TaiPan Port nodes shall be displayed as ragged yellow Shape with a name label (Figure 4.1). For the custom form, we use the `UIComponentView` view base component. We insert a label that shows the name property (line 11) and an image that shows the shape (line 12). For both we specify absolute positions and we add them manually as children of the view (line 3). We specify the bounds of view via the `bounds` getter function. The result is presented in Figure 4.6.

Listing 4.5: taipan/stencil/PortView.mxml

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <UIComponentNodeView ...
3   initialize="addChild(image); addChild(label);"
4   boundComponents="{[label]}">
5   <mx:Script>
6     <![CDATA[
7       ...
8       override public function get bounds():RectangleBounds { return new
9         RectangleBounds(x, y, 70, 90); }
10    ]]>
11   </mx:Script>
12   <mx:Label id="label" text="{model.getProperty('name')}}" x="0" y="0" width="
13     70" height="20" />
14   <mx:Image id="image" source="@Embed('assets/port.gif')" x="0" y="20"
15     width="70" height="70" />
16 </UIComponentNodeView>

```



Figure 4.6: Look of a port

### Routes and Ship Destination Ports

TaiPan `Route` edges shall be displayed as a simple line without arrows between two ports that has no arrows (Figure 4.1). We just use the `UIComponentEdgeView` view base component (Listing 4.6, line 2) without further customization effort.

TaiPan `ShipDestinationPort` edges shall be displayed as a line with a block arrow at the target (port) side and no arrow on the source (ship) side (Figure 4.1). We use the `UIComponentEdgeView` view base component (Listing 4.6, line 2) and specify that an arrow is displayed on the target side and that this arrow shall have “block” style (line 3). The result of the tow edge views is presented in Figure 4.7.

Listing 4.6: taipan/stencil/RouteView.mxml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <UIComponentEdgeView ... >
3 </UIComponentEdgeView>
```

Listing 4.7: taipan/stencil/ShipDestinationPortView

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <UIComponentEdgeView ...
3     targetArrow="arrow" targetArrowBlock="true">
4 </UIComponentEdgeView>
```

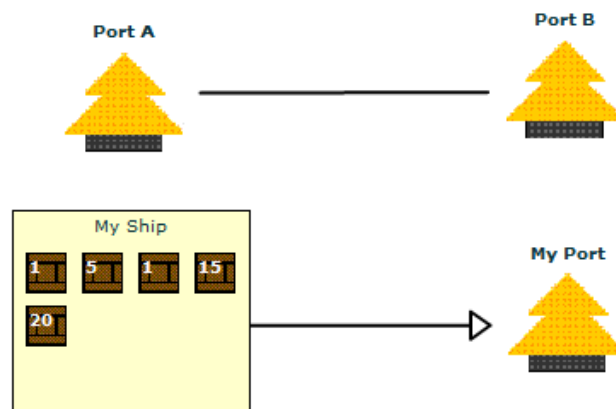


Figure 4.7: Look of the route and ship destination port edges

## 4.4 Building an Editor Application

The component oriented approach of WebMF allows for building very customizable diagram visualization and manipulation applications. In this sample, we build a web application that uses the three view components of the WebMF (diagram, tool bar, and properties editor).

### 4.4.1 Creating a Flex Application

Once we have created the stencil set library, we can use it in various web applications (Flex) or desktop applications (AIR). In this sample we will create a Flex web application (File → New → Flex Project, project name “TaiPanEditor”). After committing the wizard, the MXML source code of the created main application (TaiPanEditor.mxml) is shown.

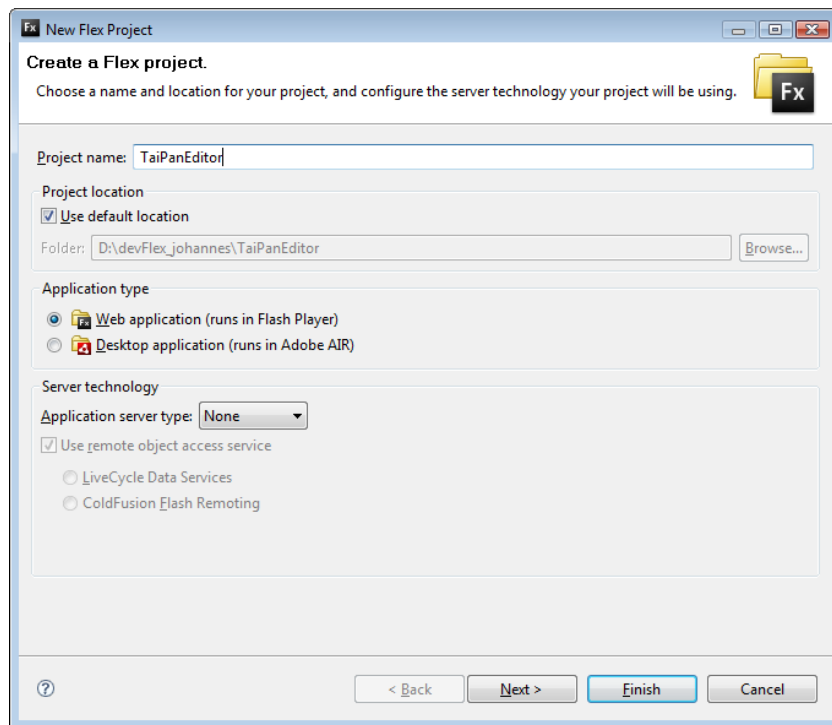


Figure 4.8: Creating sample application project

To make use of the WebMF and the TaiPan stencil set library we have created, we have to include both in the Flex library build path of the project: In the project properties (Project → Properties) we select tab Library path in Flex Library Build Path. There we add the delivered SWC library `webmf.swf` (“Add SWC...”) and the stencil set project (“TaiPanStencilSet”) (“Add Project...”) to the build path.

### 4.4.2 Creating and Connecting Components

In the application, we create and connect the required components (Listing 4.8): First, we instantiate the previously created stencil set (line 3) and a diagram model that conforms to this stencil set (line 4). Then we create a tool bar that shows all notation elements of the stencil set (line 6), a diagram view that visualizes the diagram model

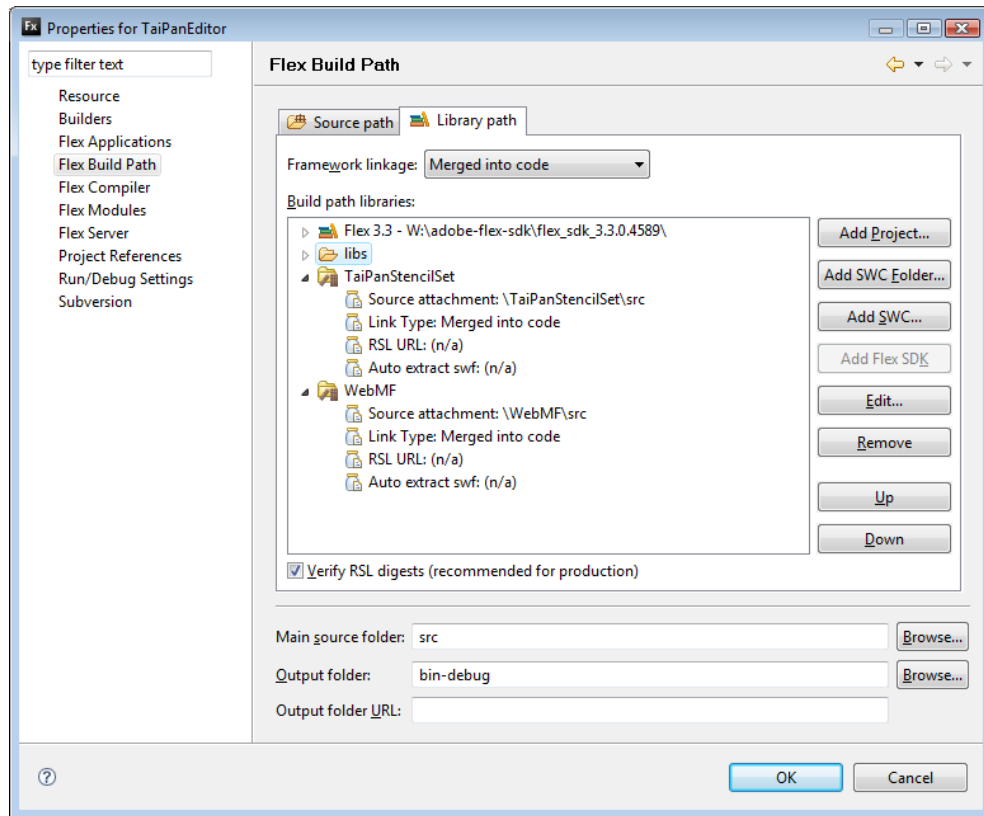


Figure 4.9: Add libraries to application build path

(line 7) and a properties view that shows the properties of the selected element of the diagram view (line 8).

Listing 4.8: Create and connect components (TaiPanEditor.mxml)

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <mx:Application ... layout="vertical">
3   <stencilset:TaiPanStencilSet id="stencilSet" />
4   <model:DiagramModel id="diagramModel" stencilSet="{stencilSet}" />
5   ...
6   <view:ToolBarView ... stencilSet="{stencilSet}" />
7   <view:DiagramView id="diagram" model="{diagramModel}" ... />
8   <view:PropertiesView ... element="{diagram.selectedElement}" />
9   ...
10 </mx:Application>

```

### 4.4.3 Appearance of the Created Application

Figures 4.10 and 4.11 show the created sample editor application in the initial state and after a few user operations.

## 4 Sample Application

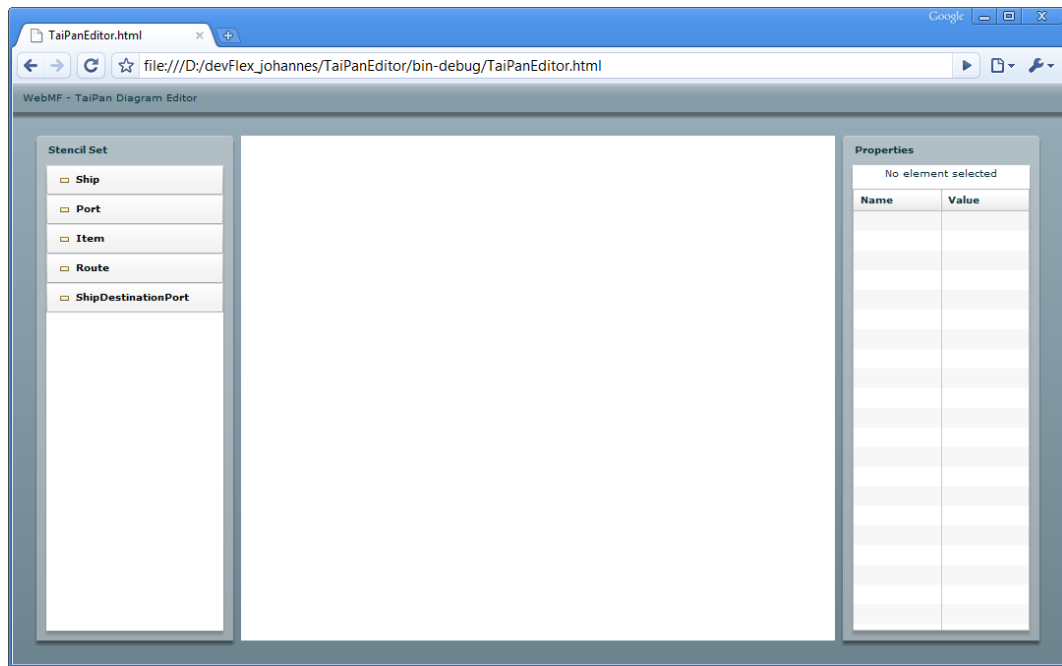


Figure 4.10: Screenshot of the created sample editor application

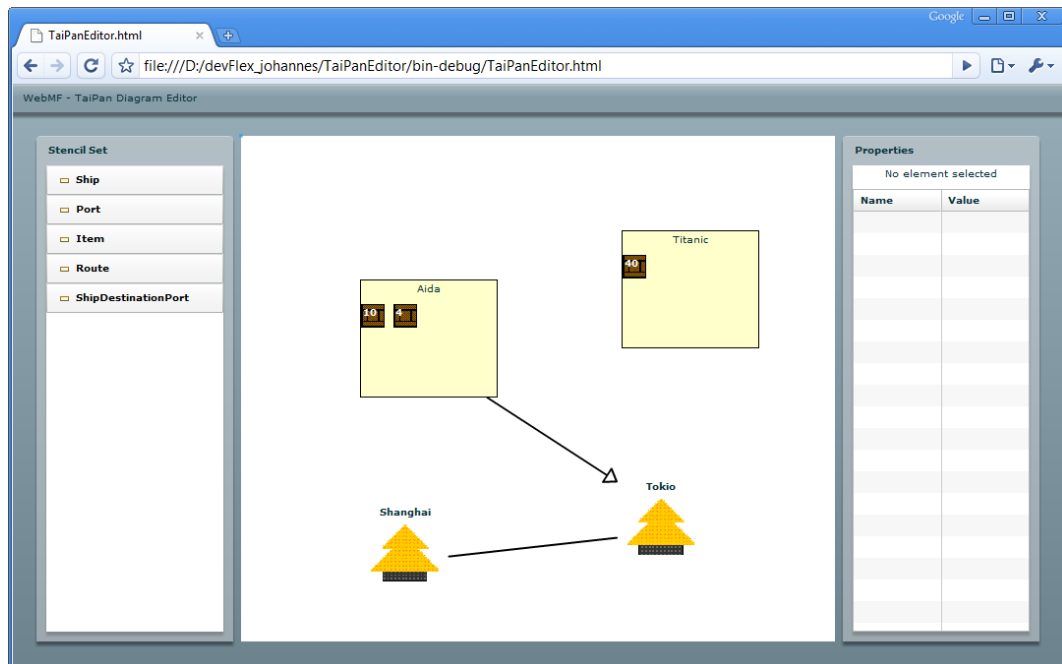


Figure 4.11: Screenshot of the created sample editor application (2)

# 5 Developing e-Learning Support with WebMF

The created framework is used for the prototypical creation of three UML editors. With class diagram (Section 5.1), state diagram (Section 5.2) and sequence diagram (Section 5.3) both structural and behavioral modeling are covered. Finally, the integration of an editor into the eLearning platform Moodle is demonstrated (Section 5.4).

The implementation will also show strengths and weaknesses of the framework, which are presented in Section 7.3 in the evaluation chapter.

## 5.1 UML Class Diagram Editor

With UML class diagrams, structural aspects of systems can be described through object classes and their relationships.

### 5.1.1 Abstract Syntax

The editor prototype shall allow for modeling classes, relationships between classes and inheritance (Figure 5.1). Centrally, classes are modeled (**Class**). Each class can inherit from any number of other classes (**inheritsFrom**). Classes can be abstract (**isAbstract**) and can include attributes (**Attribute**) and operations (**Operation**). Attributes as well as operations have a data type (**dataType**) and a specific visibility setting (**visibility**). Besides, attributes and operations cannot exist outside a class.

Associations (**Association**) connect the comprising class (**from**) to the target class (**to**, the data type). They also have a visibility setting (**visibility**). Additionally, the multiplicity can be described through an interval (**lowerBound** and **upperBound**). All classes, associations, attributes, and operations are identified by their name (**name**).

### 5.1.2 Concrete Syntax

The concrete syntax conforms to familiar UML standards, which are taught in the lecture (Table 5.1). Classes are visualized as rectangles with rounded corners and three sections that are divided by rules. In the first section the name of the class is shown. In the case of an abstract class, this is indicated through the stereotype label (“«abstract»”). The second section shows the attributes of the class in a vertical list. A symbol on the left side indicates that it is an attribute. Then the name and the data type are displayed. Accordingly, the third section shows the operations of the class. Function braces are added to the operation name.

Associations are displayed as arrow from the comprising class to the target class. The name and the multiplicity is shown nearby the arrow. Generalization (**inheritsFrom** in the abstract syntax) is shown as arrow (blocked style) from the class to its superclass.

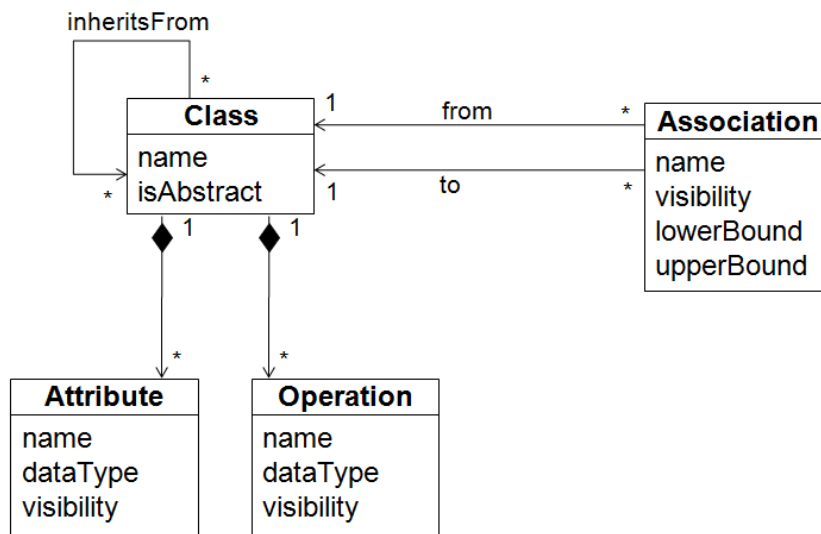


Figure 5.1: Abstract syntax of UML class diagrams

Name	Concrete Syntax
Class	<div> <b>ClassName</b> <div> <div>att1: Type</div> <div>att2: Type2</div> <div>⌘ op1() : Type</div> <div>⌘ op2() : Type</div> </div> </div>
Abstract Class	<div> <div>«abstract»</div> <b>ClassName</b> <div></div> </div>
Association	<div> <div></div> <div>→</div> </div>
Generalization	<div> <div></div> <div>▷</div> </div>

Table 5.1: Concrete syntax of UML class diagrams

### 5.1.3 Implementation

#### Stencil Set

We create a stencil set that implements the mentioned aspects (Listing 5.1): **Class** nodes are the only top-level nodes (line 6) and can include **Attribute** and **Operation** subnodes (line 11 and 16). Properties define the class name and if the class is abstract (line 8). **Association** edges connect two **Class** nodes (line 23-26). The supertypes of classes can be modeled with **Generalization** edges that connect a **Class** node (subclass) to an other **Class** (superclass) (lines 33-36). **Attribute** and **Operation** nodes have properties for specifying the name and the data type (lines 13 and 28). The elements **Association**, **Attribute** and **Operation** have a property for specifying visibility (lines 13, 18, 28).

Listing 5.1: Class diagram stencil set

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <StencilSet ...>
3      ...
4      <stencils>
5
6          <Stencil stencilName="Class" type="node" ...>
7              <canCreateIn> <ParentNode stencilName="Diagram" /> </canCreateIn>
8              <properties> <Property name="name" defaultValue="ClassName" /> <
                Property name="abstract" defaultValue="false" /> </properties>
9          </Stencil>
10
11         <Stencil stencilName="Attribute" type="node" ...>
12             <canCreateIn> <ParentNode stencilName="Class" /> </canCreateIn>
13             <properties> <Property name="name" defaultValue="name" /> <Property
                name="type" defaultValue="Type" /> <Property name="visibility"
                defaultValue="public" /> </properties>
14         </Stencil>
15
16         <Stencil stencilName="Operation" type="node" ...>
17             <canCreateIn> <ParentNode stencilName="Class" /> </canCreateIn>
18             <properties> <Property name="name" defaultValue="name" /> <Property
                name="type" defaultValue="Type" /> <Property name="visibility"
                defaultValue="public" /> </properties>
19         </Stencil>
20
21         <Stencil stencilName="Association" type="edge" ...>
22             <canConnect>
23                 <Connection>
24                     <source> <SourceNode stencilName="Class" /> </source>
25                     <target> <TargetNode stencilName="Class" /> </target>
26                 </Connection>
27             </canConnect>
28             <properties> <Property name="name" defaultValue="name" /> <Property
                name="lowerBound" defaultValue="0" /> <Property name="upperBound"
                defaultValue="1" /> <Property name="visibility" defaultValue="
                public" /> </properties>
29         </Stencil>
30
31         <Stencil stencilName="Generalization" type="edge" ...>
32             <canConnect>
33                 <Connection>
34                     <source> <SourceNode stencilName="Class" /> </source>
35                     <target> <TargetNode stencilName="Class" /> </target>
36                 </Connection>
37             </canConnect>
38         </Stencil>
39     </stencils>
40 </StencilSet>

```

## Classes, Attributes and Operations

For displaying **Class** nodes, we use the **BoxNodeView** view base class to show a rectangular node (Listing 5.2, line 2) and insert several elements which are grouped with rules (lines 6/10). To indicate, if the class is abstract, the label at the top with the text “abstract” is only shown when the appropriate property is set to “true” (line 4). The class name is displayed with a second label (line 5). Then all **Attribute** children and, finally, all **Operation** children are added at the bottom.

For displaying the **Attribute** child nodes, we use the **BoxNodeView** view base component and specify “horizontal” direction to show an “attribute” image on the left side and a label with the attribute name and type on the right side (Listing 5.3, lines 3 and 4). The operation view can be implemented similarly. The result of the three views definitions at runtime is presented in Figure 5.2.

Listing 5.2: Class view

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <view:BoxNodeView childRepeaters="{[attributesRepeater, operationsRepeater]}"
  ...>
3   ...
4   <mx:Label visible="{model.getProperty('abstract') == 'true'}" text="#171;
      abstract &#187;" ... />
5   <mx:Label htmlText="{model.getProperty('name')}" ... />
6   <mx:HRule ... />
7   <mx:Repeater id="attributesRepeater" dataProvider="{model.getChildren('
      Attribute')}" ...>
8     <stencil:AttributeView parentView="{thisNodeView}" model="{
      attributesRepeater.currentItem}" ... />
9   </mx:Repeater>
10  <mx:HRule ... />
11  <mx:Repeater id="operationsRepeater" dataProvider="{model.getChildren('
      Operation')}" ...>
12    <stencil:OperationView parentView="{thisNodeView}" model="{
      operationsRepeater.currentItem}" ... />
13  </mx:Repeater>
14 </view:BoxNodeView>

```

Listing 5.3: Attribute view

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <view:BoxNodeView direction="horizontal" ...>
3   <mx:Image source="assets/ecore/EAttribute.gif" />
4   <mx:Label text="{_model.getProperty('name') + ' : ' + _model.getProperty('
      type')}" ... />
5 </view:BoxNodeView>

```

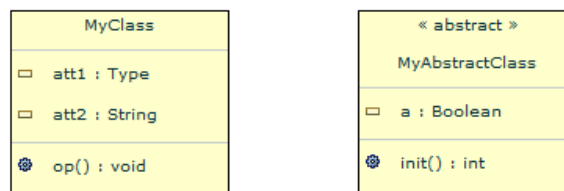


Figure 5.2: Look of a class and an abstract class with some attributes and operations

## Associations and Generalizations

For displaying **Association** edges, we use the **UIComponentEdgeView** view base component to show a line connector and specify that it additionally shows an arrow on the

target side (Listing 5.4). The generalization view can be implemented similarly. The results of both view definitions at runtime are presented in Figure 5.3.

Listing 5.4: Association view

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <view:UIComponentEdgeView ...
3     targetArrow="arrow"
4     >
5 </view:UIComponentEdgeView>

```

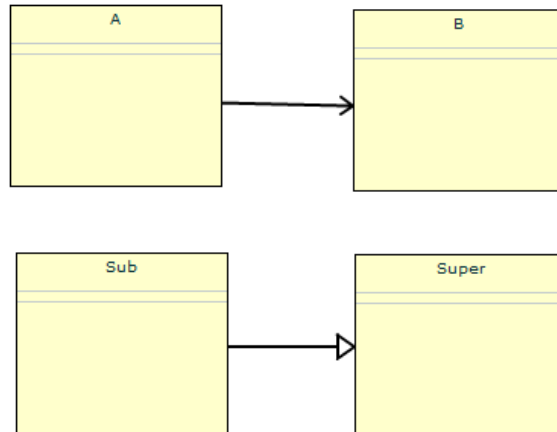


Figure 5.3: Look of an association edge and a generalization edge

## Editor Application

For creating the application, we create and connect the required components (Listing 5.5): First, we instantiate the previously created stencil set (line 3) and a diagram model that conforms to this stencil set (line 4). Then we create a tool bar that shows all notation elements of the stencil set (line 6), a diagram view that visualizes the diagram model (line 7) and a properties view that shows the properties of the selected element of the diagram view (line 8). The resulting editor application is presented in Figure 5.4.

Listing 5.5: Create and connect components

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <mx:Application ... layout="vertical">
3     <stencilset:ClassDiagramStencilSet id="stencilSet" />
4     <model:DiagramModel id="diagramModel" stencilSet="{stencilSet}" />
5     ...
6     <view:ToolBarView ... stencilSet="{stencilSet}" />
7     <view:DiagramView id="diagram" model="{diagramModel}" ... />
8     <view:PropertiesView ... element="{diagram.selectedElement}" />
9     ...
10 </mx:Application>

```

### 5.1.4 Issues

The editor could be implemented almost completely, only a few issues appeared during the implementation of the stencil set.

First, in UML class diagrams the name of associations is shown as label at the center of the association edge and the multiplicity intervals are shown at the edge endpoints

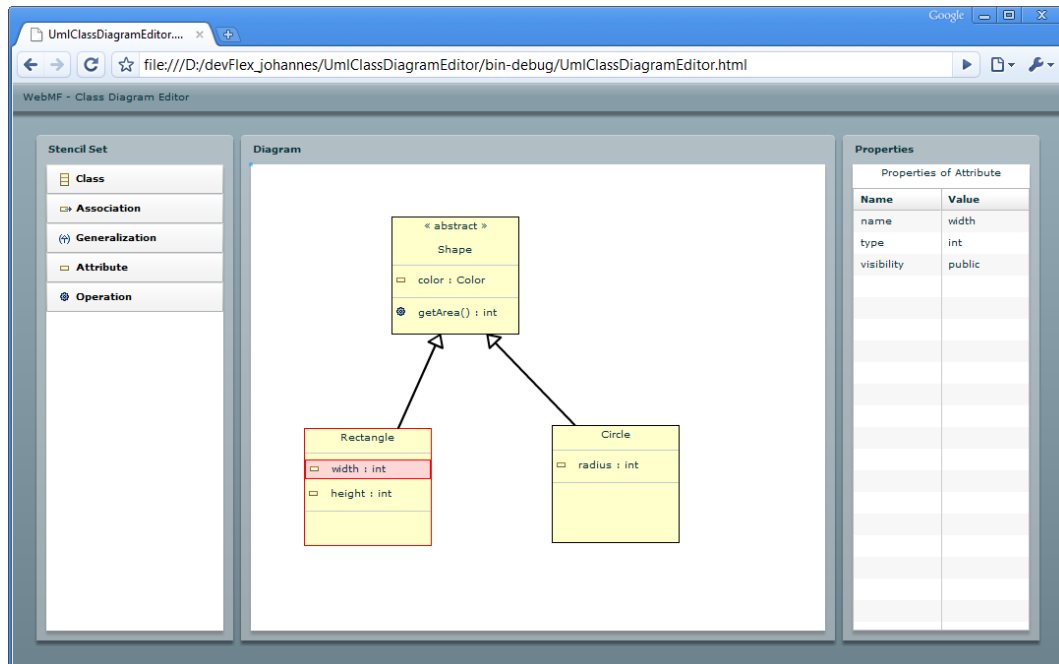


Figure 5.4: Screenshot of the WebMF UML class diagram editor

(Figure 5.5). This functionality, which can be referred to as “edge labels”, is not yet supported by the WebMF. A mechanism should be provided for easy creation of such labels for the start point, center and the end point of an edge.

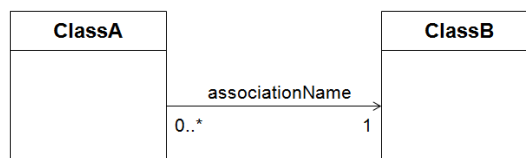


Figure 5.5: Edge labels in UML class diagrams

Second, the data types of attributes or operations can be primitive data types, but also one of the other classes (Example in Figure 5.6). With the current implementation, the elements cannot be related. So if the class name of the ClassB is changed, the attribute b with this class as data type is not updated. Currently, there are no mapping facilities provided by the framework since a basic idea was to avoid complicated mapping and to focus on the visualization features.



Figure 5.6: Referencing another class as data type in a UML class diagram

And third, the storage of all property values is problematically. E.g. the `isAbstract` property of the class stencil represents a Boolean value. The values that are set in the properties editor cannot be validated, also the evaluation is error-prone (e.g. `isAbstract == "true"`).

## 5.2 UML State Diagram Editor

### 5.2.1 Abstract Syntax

The editor prototype shall allow for modeling states (including initial state and end state), activities inside these states, and transitions between states. States (**State**) are the central constructs (Figure 5.7). They are identified by their name, and can comprise activities (**Activity**). The latter define what to execute (description) and when this activity is executed (execution, e.g. “entry” / “exit”). Further, there are two pseudo-states (**InitialState** / **EndState**), which both can exist only once.

Transitions (**Transition**) lead from one state to the next state. Thereby, the outgoing state (**source**) can be the initial state or a normal state (i.e. not the end state). Vice versa, this applies to the target state (**target**). It is also described, what happens in the course of a transition (description).

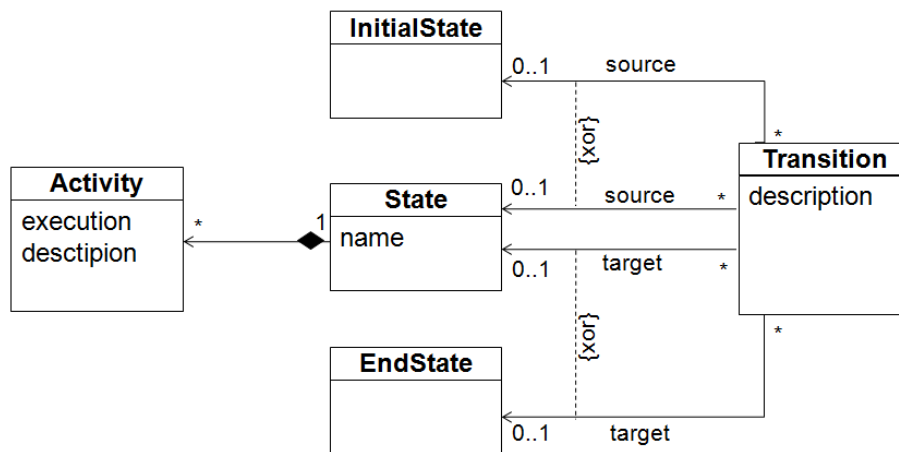


Figure 5.7: Abstract syntax of UML state diagrams

### 5.2.2 Concrete Syntax

The concrete syntax conforms to familiar UML standards, which are taught in the lecture (Table 5.2). States are visualized as rectangles with two sections that are divided by rules. The first section shows the name of the state. The second section shows the details of the state. Concretely, the activities (entry or exit activities, etc.) are shown in a vertical list, consisting of the execution type and the description of the executed activity.

The initial state and the end state are shown as circular shapes with no additional information. Transitions are displayed as simple arrows from the outgoing state to the ingoing state.

### 5.2.3 Implementation

#### Stencil Set

We create a stencil set that implements the mentioned aspects (Listing 5.6): **State**, **InitialState** and **EndState** nodes are top-level nodes (line 6) and can include subnodes of type **Activity** (line 20). **State** nodes can include **Activity** subnodes (line 20). **Transition** edges usually connect two **State** nodes (lines 27-39). Alternatively there

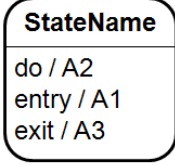



Name	Concrete Syntax
State	
Transition	
Initial State	
End State	

Table 5.2: Concrete syntax of WebMF UML state diagrams

can be an `InitialState` node on the source side or an `EndState` node on the target side. Properties are added for all attributes.

Listing 5.6: State diagram stencil set

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <StencilSet ...>
3      ...
4      <stencils>
5
6          <Stencil stencilName="State" type="node" ...>
7              <canCreateIn> <ParentNode stencilName="Diagram" /> </canCreateIn>
8              <properties> <Property name="name" defaultValue="MyState" /> </
              properties>
9          </Stencil>
10
11         <Stencil stencilName="InitialState" type="node" ...>
12             <canCreateIn> <ParentNode stencilName="Diagram" /> </canCreateIn>
13         </Stencil>
14
15         <Stencil stencilName="EndState" type="node" ...>
16             <canCreateIn> <ParentNode stencilName="Diagram" /> </canCreateIn>
17         </Stencil>
18
19         <Stencil stencilName="Activity" type="node" ...>
20             <canCreateIn> <ParentNode stencilName="State" /> </canCreateIn>
21             <properties> <Property name="execution" defaultValue="do" /> <
                Property name="description" defaultValue="doSomething" /> </
                properties>
22         </Stencil>
23
24         <Stencil stencilName="Transition" type="edge" ...>
25             <properties> <Property name="description" defaultValue="Event" /> </
                properties>
26             <canConnect>
27                 <Connection>
28                     <source> <SourceNode stencilName="State" /> </source>
29                     <target> <TargetNode stencilName="State" /> </target>
30                 </Connection>
31                 <Connection>
32                     <source> <SourceNode stencilName="InitialState" /> </source>

```

```

33         <target> <TargetNode stencilName="State" /> </target>
34     </Connection>
35 </Connection>
36     <source> <SourceNode stencilName="State" /> </source>
37     <target> <TargetNode stencilName="EndState" /> </target>
38 </Connection>
39 </canConnect>
40 </Stencil>
41
42 </stencils>
43 </StencilSet>

```

## States and Activities

For displaying **State** nodes, we use the **BoxNodeView** view base class to show a rectangular node (Listing 5.7, line 2). At the top we add a label that displays the class name (line 4), followed by a horizontal rule (line 5) and finally all **Activity** children (line 6-8).

For displaying the **Activity** child nodes, we use the **BoxNodeView** view base component (Listing 5.8). Inside the node we compose the text of a label with the execution and description properties.

The result of the two views definitions at runtime is presented in Figure 5.8.

Listing 5.7: State view

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <view:BoxNodeView cornerRadius="16" childRepeaters="{[activityRepeater]}" ...>
3     ...
4     <mx:Label text="{model.getProperty('name')}}" ... />
5     <mx:HRule ... />
6     <mx:Repeater id="activityRepeater" dataProvider="{model.getChildren('
    Activity')}}">
7         <ActivityView parentView="{thisNodeView}" model="{activityRepeater.
            currentItem}" ... />
8     </mx:Repeater>
9     ...
10 </view:BoxNodeView>

```

Listing 5.8: Activity view

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <view:BoxNodeView ...>
3     <mx:Label htmlText="&lt;b&gt;{model.getProperty('execution')}&#47;&lt;/b&gt;
        {model.getProperty('description')}}" ... />
4 </view:BoxNodeView>

```



Figure 5.8: Look of a state including two activities

## Initial State and End State

For displaying an `InitialState` node, we use the `UIComponentNodeView` view base component (Listing 5.9) and draw a filled circle (line 5). The `EndState` node view can be implemented similarly by drawing two filled circles. The results of both view definitions at runtime are presented in Figure 5.9.

Listing 5.9: Initial state view

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <UIComponentNodeView width="31" height="31" ...>
3   <mx:Script>
4     <![CDATA[
5       override protected function draw():void {
6         this.graphics.lineStyle(1);
7         this.graphics.beginFill(0x000000, 1);
8         this.graphics.drawCircle(15, 15, 15);
9         this.graphics.endFill();
10      }
11    ]]>
12   </mx:Script>
13 </UIComponentNodeView>

```



Figure 5.9: Look of the initial and the end state

## Transitions

For displaying `Transition` edges, we use the `UIComponentEdgeView` view base component to show a line connector and specify that it additionally shows an arrow on the target side (Listing 5.10, result in Figure 5.10).

Listing 5.10: Transition view

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <view:UIComponentEdgeView
3   xmlns="stateDiagram.stencil.*"
4   xmlns:mx="http://www.adobe.com/2006/mxml"
5   xmlns:view="webmf.view.*"
6
7   targetArrow="arrow"
8   targetArrowAngle="20"
9   >
10 </view:UIComponentEdgeView>

```

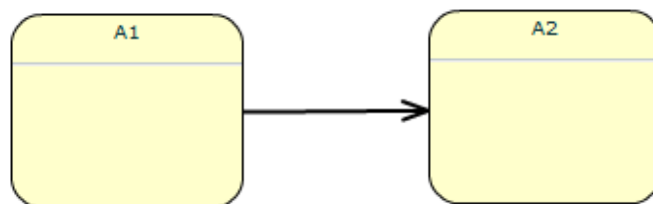


Figure 5.10: Look of a transition edge

## Editor Application

For creating the application, we create and connect the required components (Listing 5.11): First, we instantiate the previously created stencil set (line 3) and a diagram model that conforms to this stencil set (line 4). Then we create a tool bar that shows all notation elements of the stencil set (line 6), a diagram view that visualizes the diagram model (line 7) and a properties view that shows the properties of the selected element of the diagram view (line 8). The resulting editor application is presented in Figure 5.11.

Listing 5.11: Create and connect components

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <mx:Application ... layout="vertical">
3      <stencilset:StateDiagramStencilSet id="stencilSet" />
4      <model:DiagramModel id="diagramModel" stencilSet="{stencilSet}" />
5      ...
6      <view:ToolBarView ... stencilSet="{stencilSet}" />
7      <view:DiagramView id="diagram" model="{diagramModel}" ... />
8      <view:PropertiesView ... element="{diagram.selectedElement}" />
9      ...
10 </mx:Application>

```

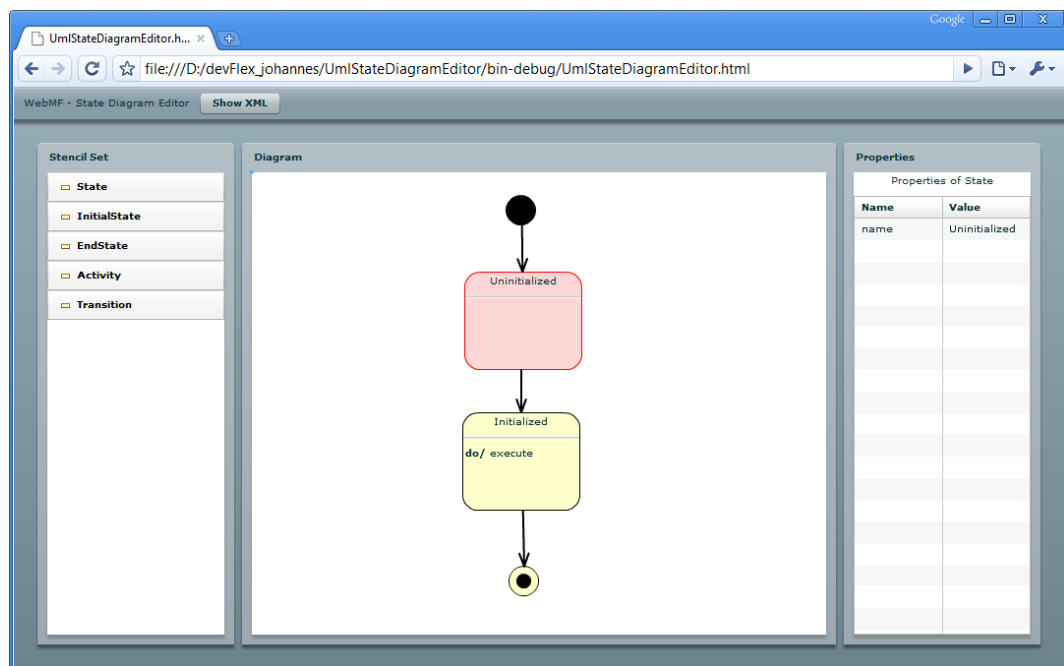


Figure 5.11: Screenshot of the WebMF UML state diagram editor

### 5.2.4 Issues

The editor could be implemented almost completely, only a few issues appeared during the implementation of the stencil set.

First, in a valid UML state diagram there has to be exactly one initial state. This type is not yet supported but desirable for future enhancements.

Second, states in the UML state diagram are nested sometimes, i.e. there are states including substates (Figure 5.12). This cannot be implemented with the current functionality. The Stencil View DSL would allow positioning the subelements with absolute

layout. Concretely, a **Canvas** container could be filled with a subnode **Repeater**, similar to the activity subnodes.

Currently, when dropping a potential subnode onto an existing node (when this parent-child relationship is allowed), this causes *that* the node is added as a child, but there is no possibility to designate *where* the child is placed inside its parent, i.e. the position in the list of children or the relative coordinates inside the parent node. Such an enhancement of the drag-and-drop mechanism would enable many more diagram constructions. Of course, these inner node positions have also to be stored in the data structure.

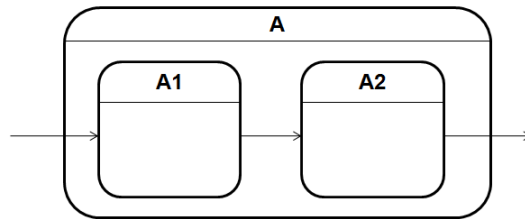


Figure 5.12: Nesting of states in UML state diagrams

## 5.3 UML Sequence Diagram Editor

UML Sequence Diagrams are used to show the chronological order of messages between different objects of an application. The editor prototype shall support the following aspects (Figure 5.13): Lifelines and messages can be modeled. Lifelines have a role and a data type. Messages have a description and can be synchronous or asynchronous.

### 5.3.1 Abstract Syntax

The editor prototype shall allow for modeling the lifelines of actors and the sent messages (Figure 5.13). Lifelines (**Lifeline**) show the behavior of a role (**role**) of a specific type (**type**). Every message (**Message**) has one lifeline as sender (**sender**) and one as receiver (**receiver**). Further they can be synchronous or asynchronous (**isSynchronous**).

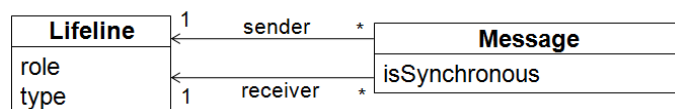


Figure 5.13: Abstract syntax of WebMF UML sequence diagrams

### 5.3.2 Concrete Syntax

The concrete syntax conforms to familiar UML standards, which are taught in the lecture (Table 5.3). Lifelines are visualized as dashed vertical lines with a lifeline head at the top. This head shows the role and the type of the actor. Messages connect the sender to the receiver lifeline with an arrow. Different arrow types are used for different message types (synchronous / asynchronous).

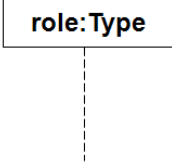
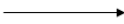

Name	Concrete Syntax
Lifeline	
Synchronous Message	
Asynchronous Message	

Table 5.3: Concrete syntax of WebMF UML sequence diagrams

### 5.3.3 Implementation

#### Stencil Set

We create a stencil set that implements the mentioned aspects (Listing 5.12): **Lifeline** nodes are top-level nodes and define properties for role and type (lines 6-9). The edges of type **SynchronousMessage** and **AsynchronousMessage** connect two **Lifeline** nodes and define a property for their description.

Listing 5.12: Sequence diagram stencil set

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <StencilSet ...>
3      ...
4      <stencils>
5
6          <Stencil stencilName="Lifeline" type="node" ...>
7              <canCreateIn> <ParentNode stencilName="Diagram" /> </canCreateIn>
8              <properties> <Property name="role" defaultValue="role" /> <Property
9                  name="type" defaultValue="Type" /> </properties>
10          </Stencil>
11
12          <Stencil stencilName="SynchronousMessage" type="edge" ...>
13              <canConnect>
14                  <Connection>
15                      <source> <SourceNode stencilName="Lifeline" /> </source>
16                      <target> <TargetNode stencilName="Lifeline" /> </target>
17                  </Connection>
18              </canConnect>
19              <properties> <Property name="description" defaultValue="label" /> </
20                  properties>
21          </Stencil>
22
23          <Stencil
24              stencilName="AsynchronousMessage" type="edge" ...>
25              <canConnect>
26                  <Connection>
27                      <source> <SourceNode stencilName="Lifeline" /> </source>
28                      <target> <TargetNode stencilName="Lifeline" /> </target>
29                  </Connection>
30              </canConnect>
31              <properties> <Property name="description" defaultValue="label" /> </
32                  properties>
33          </Stencil>
34      </stencils>
35  </StencilSet>

```

## Lifelines

For displaying a **Lifeline** node, we use the **UIComponentNodeView** view base component (Listing 5.13). We define (line 10-12) and add (line 3) a rectangle with an label to show the lifeline head with role and type. In the **draw** method we draw the dashes of the lifeline. The results of the view definitions at runtime is presented in Figure 5.14.

Listing 5.13: Lifeline view

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <view:UIComponentNodeView boundComponents="{[head]}"
3      initialize="this.addChild(headBox); headBox.addChild(head);" ...>
4      <mx:Script>
5          ...
6          override protected function draw():void {
7              // draw dashes
8          }
9          ...
10     </mx:Script>
11     <mx:Box id="headBox" ...>
12         <mx:Label id="head" text="{_model.getProperty('role')} : {_model.
13             getProperty('type'))}" ... />
14     </mx:Box>
15 </view:UIComponentNodeView>

```



Figure 5.14: Look of a lifeline

## Messages

For displaying **SynchronousMessage** edges, we use the **UIComponentEdgeView** view base component to show a line connector and specify that it shows a filled arrow with block style on the target side (Listing 5.14). The **AsynchronousMessage** edges can be implemented similarly, with a normal arrow on the target side (**targetArrowBlock="false"**). The results of both view definitions at runtime are presented in Figure 5.15.

Listing 5.14: Synchronous message view

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <view:UIComponentEdgeView
3      xmlns="sequenceDiagram.stencil.*"
4      xmlns:mx="http://www.adobe.com/2006/mxml"
5      xmlns:basic="basic.*"
6      xmlns:view="webmf.view.*"
7
8      targetArrow="arrow"
9      targetArrowBlock="true"
10     targetArrowFill="true"
11     targetArrowSize="10"
12 >
13 </view:UIComponentEdgeView>

```

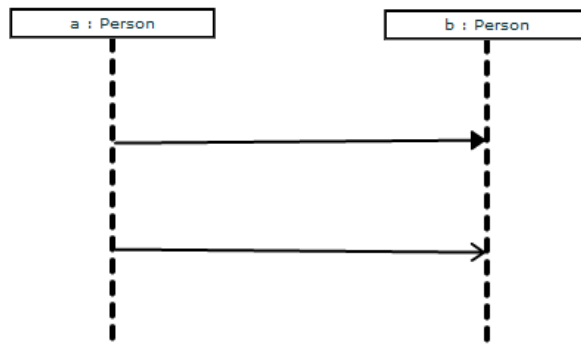


Figure 5.15: Look of a synchronous and asynchronous messages

### Editor Application

For creating the application, we create and connect the required components (Listing 5.15): First, we instantiate the previously created stencil set (line 3) and a diagram model that conforms to this stencil set (line 4). Then we create a tool bar that shows all notation elements of the stencil set (line 6), a diagram view that visualizes the diagram model (line 7) and a properties view that shows the properties of the selected element of the diagram view (line 8). The resulting editor application is presented in Figure 5.4.

Listing 5.15: Create and connect components

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <mx:Application ... layout="vertical">
3      <stencilset:SequenceDiagramStencilSet id="stencilSet" />
4      <model:DiagramModel id="diagramModel" stencilSet="{stencilSet}" />
5      ...
6      <view:ToolBarView ... stencilSet="{stencilSet}" />
7      <view:DiagramView id="diagram" model="{diagramModel}" ... />
8      <view:PropertiesView ... element="{diagram.selectedElement}" />
9      ...
10 </mx:Application>

```

### 5.3.4 Issues

Overall the UML sequence diagram does not fit well into the schema of WebMF. Many issues appeared and the result is a fragmentary implementation.

In contrast to other UML diagrams, the position of elements matters in sequence diagrams. The vertical axis of the diagram represents the timeline (Figure 5.17). So it makes a difference if a message is sent before or after (in the visualization above or below) another message. Also, lifelines are placed in a horizontal row (equal distance  $d$ ), sometimes a lifeline is below others, when the related actor appears at a later point in time. This general discrepancy becomes manifest in the following issues.

First, lifelines cannot be realized adequately. Beside the mentioned placement of the lifelines, they also have to be vertically resizable to show their lifetime. Manual resizing of nodes — and the storage of the sizes in the data structures — is not yet supported by the framework. This is also the reason, why execution specification cannot be implemented for lifelines. The length of these bars is important and has to be set by the user.

Second, execution specifications and combined fragments, also essential constructs in UML sequence diagrams, cannot be implemented. Figure 5.18 shows an example with both constructs. The execution specifications are displayed as bars with variable

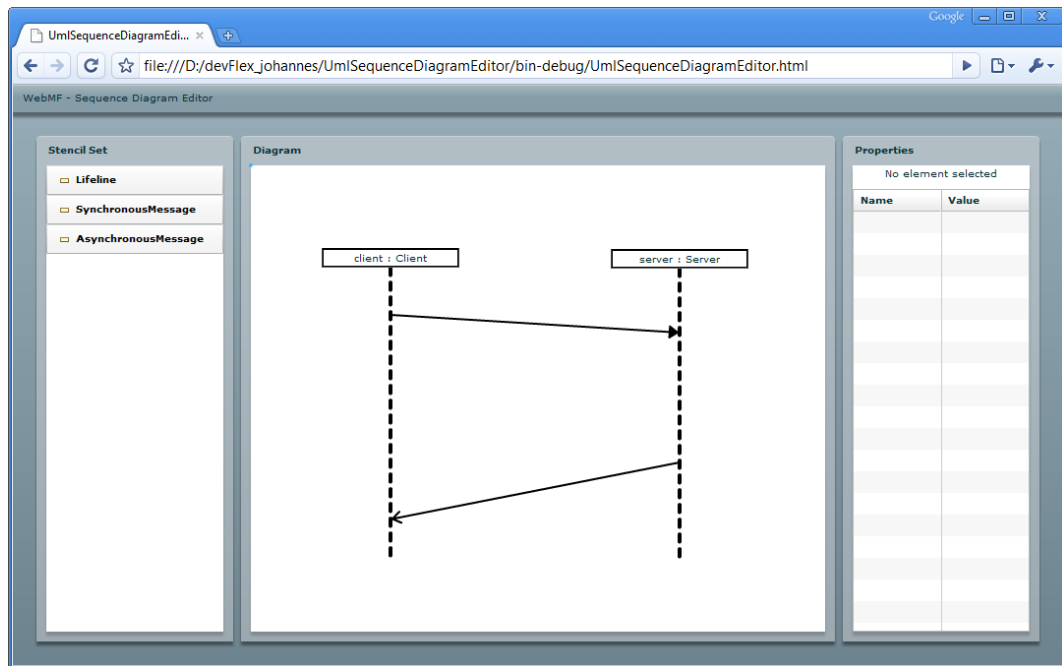


Figure 5.16: Screenshot of the WebMF UML sequence diagram editor

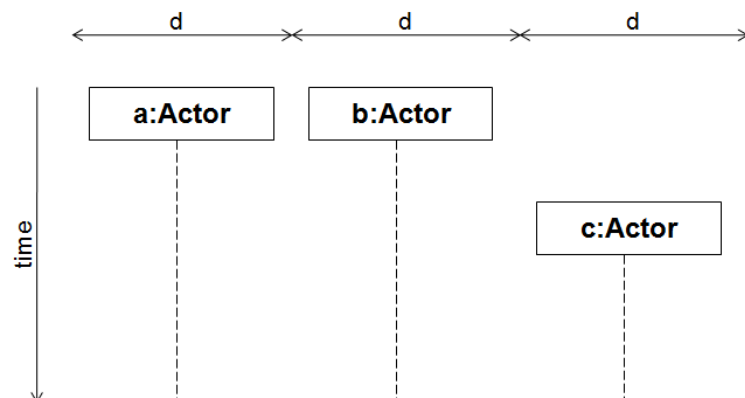


Figure 5.17: Placement of lifelines in UML sequence diagrams

length on the lifeline. As already mentioned, there is no resize functionality in WebMF. Also, the placement of the execution specification on the lifeline is problematic since they have to be placed central on it in the horizontal dimension and with an absolute coordinate according to the time in the vertical dimension. This mismatches with the alignment system of the framework.

Combined fragments, displayed as one or more fragments according to the number of operands, cannot be implemented due to the need of special size determination. This size determination is quite complex as it has to account for several related elements in different manner. In the horizontal dimension it has to cover all lifelines that are involved in one of the operands of the combined fragment. In the vertical dimension each operand has to cover all involved messages and also all execution specifications that are part of the operand. The different operands have to follow up one to another without gap. These multiple alignment rules cannot be represented in WebMF. Following the node-subnode schema, the execution specifications would be subnodes of lifelines. But they have also to be aligned to the comprising operand. WebMF only supports parent-child relationships where a child has exactly one parent.

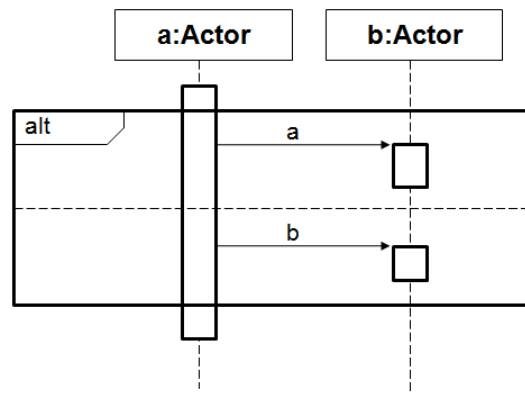


Figure 5.18: Execution specifications and combined fragments in UML sequence diagrams

Third, a message edge has to be horizontal as long the message is not indicated as time consuming. This contradicts the automatic adjustment to related nodes (lifelines or execution specifications) of WebMF. In a correct implementation the begin of an execution specification has to follow up the message edge's end point.

Some of these issues could be realized with workarounds, but this seems to be a hardly feasible solution. UML sequence diagrams do not fit in the concept of WebMF.

Besides, the synchronous message stencil and the asynchronous message stencil are almost the same. It would be good to encapsulate functionality with an inheritance mechanism to avoid redundancies and inflated stencil definition files.

## 5.4 Integration of Editors Into Moodle

This section demonstrates, how WebMF applications are integrated into the Moodle platform. Concretely, a practical modeling exercise is created for an example Moodle course, where students have to create an UML class diagram for an assignment.

Section 5.4.1 explains how to enable import- and export functionality in WebMF applications. Section 5.4.2 describes the functionality of the created Moodle WebMF

plug-in. Finally, Section 5.4.3 demonstrates how to use the plugin to create a sample assignment.

### 5.4.1 Enabling Import and Export Functions

WebMF features an `ImportExportManager` class that provides import and export functionality. Listing 5.16) shows the application MXML file of the UML class editor created in Section 5.1, extended with JavaScript import and export functionality. An instance of the `ImportExportManager` is added (line 13). After the application is initialized, JavaScript import and export are enabled for the edited `diagramModel` (line 8). These functions activate communication between the editor application SWF file (which runs in the Flash Player Plug-in) with the surrounding HTML page.

Listing 5.16: Use Import and Export functions

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <mx:Application
3      xmlns:mx="http://www.adobe.com/2006/mxml"
4      xmlns:view="webmf.view.*"
5      xmlns:model="webmf.model.*"
6      xmlns:stencilset="classDiagram.stencilset.*"
7      layout="vertical"
8      initialize="importExportManager.enableJavascriptExport(diagramModel);
        importExportManager.enableJavascriptImport(diagramModel);"
9  >
10
11  <stencilset:ClassDiagramStencilSet id="stencilSet" />
12  <model:DiagramModel id="diagramModel" stencilSet="{stencilSet}" />
13  <util:ImportExportManager id="importExportManager" />
14
15  <mx:ApplicationControlBar dock="true">
16      <mx:Label text="WebMF - Class Diagram Editor" />
17  </mx:ApplicationControlBar>
18
19  <mx:HBox height="100%" width="100%">
20      <view:ToolBarView id="toolbar" stencilSet="{stencilSet}" height="100%"
        width="200" />
21      <mx:Panel width="100%" height="100%" title="Diagram">
22          <view:DiagramView id="diagram" model="{diagramModel}" width="100%"
        height="100%" />
23      </mx:Panel>
24      <view:PropertiesView id="properties" element="{diagram.selectedElement}"
        width="200" height="100%" />
25  </mx:HBox>
26
27 </mx:Application>

```

### 5.4.2 Developing the WebMF Assignment Plug-in for Moodle

The component oriented approach of WebMF allows for creating diagram editors that can be integrated in various forms (Figure 5.19). For example, WebMF applications could be used in Moodle resources (lecture), forums, or assignments. The latter is realized in this section.

The Moodle e-Learning platform is deployed on a PHP/Apache web server (Figure 5.20). It features a plug-in mechanism that was used to create the WebMF assignment plugin. The WebMF editor application is running in the Flash Player Plug-in in the user's web browser. The communication between client and server is realized with HTTP GET and POST requests. Additionally, the browser communicates with the editor application Flash Player Plug-in with JavaScript.

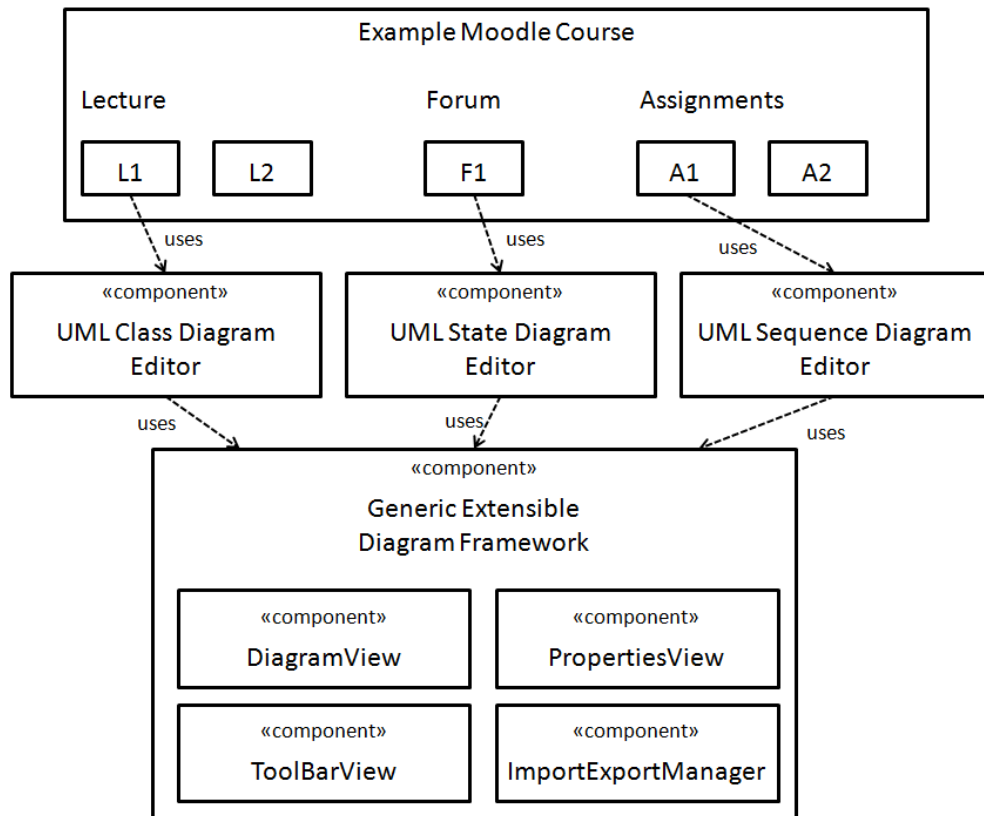


Figure 5.19: Components in the e-Learning Application Environment

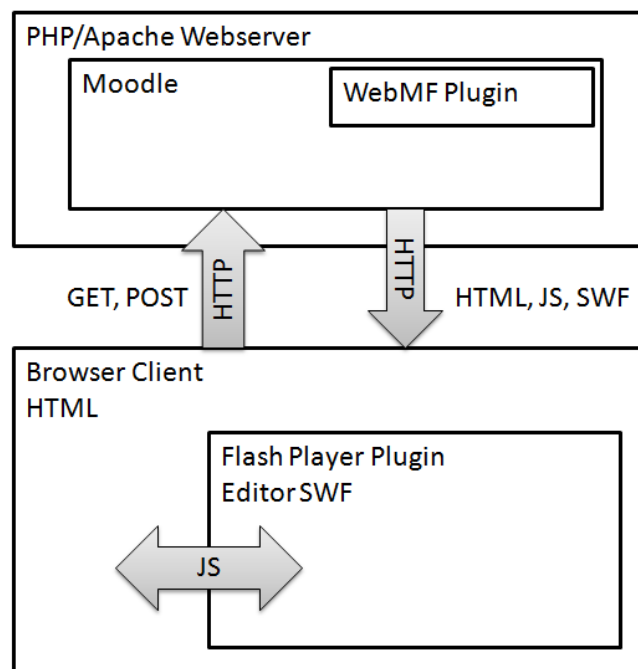


Figure 5.20: Integration of WebMF editor applications in Moodle

Figure 5.21 shows the workflow of a WebMF Moodle assignment. An assignment starts, when the assignment page is loaded (i.e. when the browser requests the assignment HTML page from the Moodle web server). This page includes the WebMF editor SWF and the initial model XML. When the editor SWF is initialized completely (inside the Flash Player Plug-in) it notifies the browser by calling a JavaScript function (`swfReady()`) that is included in the HTML document. The browser then injects the initial model XML into the editor (`javascriptImport(initialXml)`). This is usually the serialized version of an empty diagram, or — if the user already started this assignment — the previously saved assignment of this user.

After the user has created his/her solution by using the editors user interface functions he presses the submit button on the assignment HTML page. Then the serialized model XML of the current state of the editor is requested from the SWF (`javascriptExport() / exportModelXml(xmlString)`) and submitted to the Moodle server.

### 5.4.3 Creating a Sample Assignment

The WebMF assignment Moodle plug-in can be installed on a Moodle installation by unpacking the `webmf-assignment-moodle-plugin.zip` archive into `MOODLEDIR/mod/assignment/type`. To add a WebMF editor application, the SWF file has to be copied to `MOODLEDIR/mod/assignment/type/webmf/editors/` and registered in the `MOODLEDIR/mod/assignment/type/webmf/editors.php` configuration file.

Moodle users with the “Teacher” role can now create a new WebMF assignment activity (Add an activity... → Assignments → WebMF) to a course (Figure 5.22). Each assignment has a name, the description is used for the problem description of the exercise. One of the registered editors can be selected in the “Editors” drop-down menu. The screenshot in Figure 5.23 shows the created assignment that is currently used by a student.

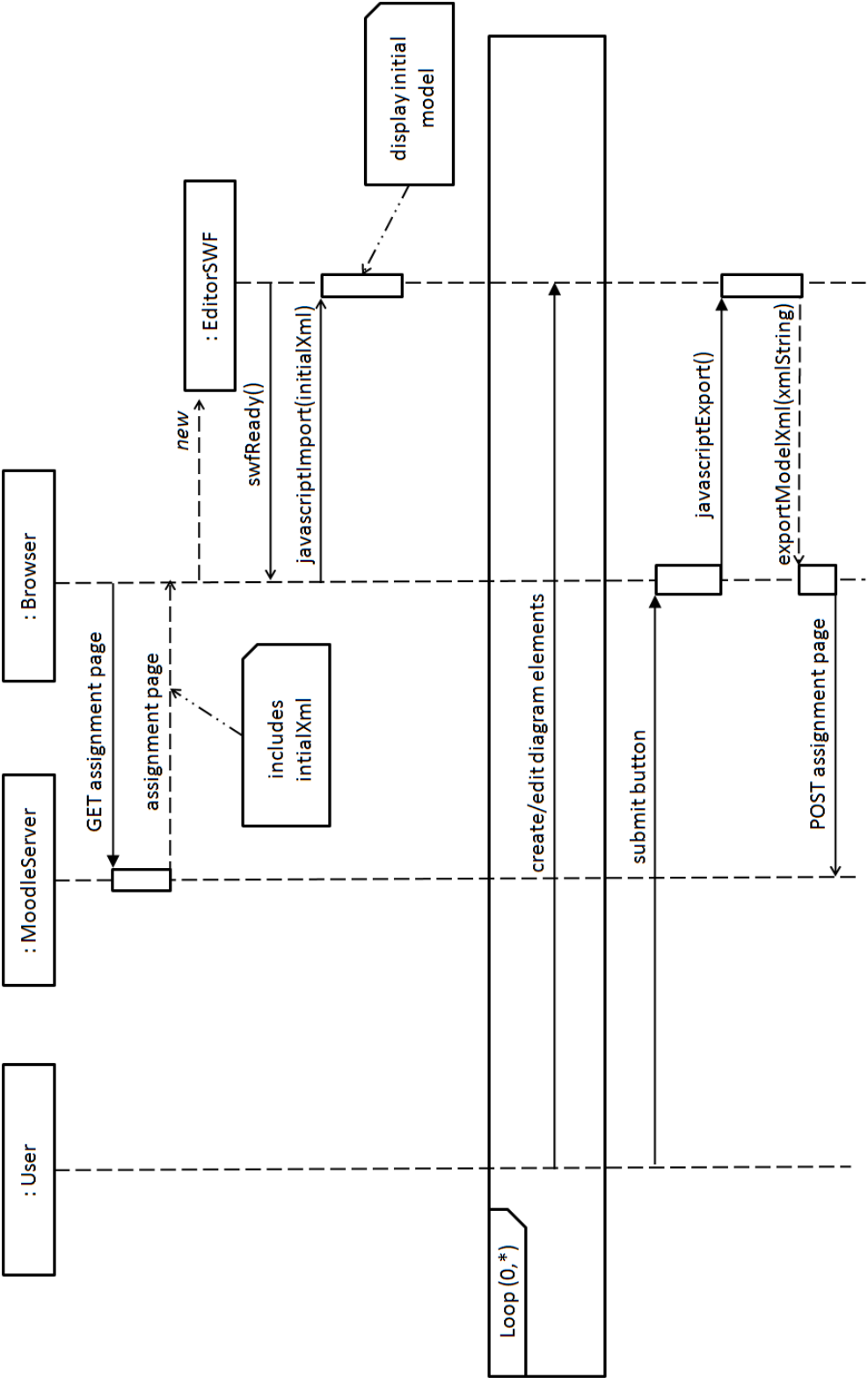


Figure 5.21: Workflow of a WebMF Moodle assignment

OOM: Editing Assignment

http://web.student.tuwien.ac.at/~e0427416/moodle/course/modedit.php?update=15&return=1

Objektorientierte Modellierung (My) You are logged in as Admin User (Logout)

MyTUWEL > OOM > Assignments > Assignment: Supermarket > Editing Assignment

Settings Locally assigned roles Override permissions

Updating Assignment in topic 1

**General**

Assignment name\* Assignment: Supermarket

Description\* A supermarket software allows the administration of departments and employees. Employees can be of type trainee or clerk. For each employee, a distinct social insurance number, staff number, name, address and the department, for which he works, is recorded. Additionally, trainees have a year of apprenticeship. For each department, a distinct name, its employees, the department head (an employee) and the offered products

Grade 100

Available from 4 August 2009 19:00 Disable

Due date 11 August 2009 19:00 Disable

Prevent late submissions No

**WebMF**

Editor UmlClassDiagramEditor.swf

**Common module settings**

Group mode No groups

Visible Show

ID number

Grade category Uncategorised

Save and return to course Save and display Cancel

There are required fields in this form marked\*.

Figure 5.22: Adding a WebMF Moodle assignment activity

The screenshot shows a Moodle assignment page in a web browser. The browser's address bar displays the URL: `http://web.student.tuwien.ac.at/~e0427416/moodle/mod/assignment/view.php?id=15&edit=1`. The page title is "Objektorientierte Modellierung (My)". Below the title, there is a breadcrumb trail: "MyTUWEL > OOM > Assignments > Assignment: Supermarket > Edit my submission". To the right of the breadcrumb, there are buttons for "Jump to...", "Update this Assignment", and a link to "View 1 submitted assignments".

The main content area contains a text box with the following text:

A supermarket software allows the administration of departments and employees. Employees can be of type trainee or clerk. For each employee, a distinct social insurance number, staff number, name, address and the department, for which he works, is recorded. Additionally, trainees have a year of apprenticeship. For each department, a distinct name, its employees, the department head (an employee) and the offered products are recorded. Each department offers multiple products, whereas a product can be offered only by one department. A product is identified by a product number. Additionally, a name and the price are stored. A product is offered by at least one supplier.

Suppliers are identified by their name and address. Each product can be offered from each supplier with particular conditions. Conditions are described by a base price per unit, a supplier intern product number (different to the article number of the supermarket), and a maximum discount. Not all potential suppliers deliver effectively.

A delivery is executed by a supplier. It is proceeded at a certain date and has a distinct serial number. A delivery consists of multiple products. For each product, the amount and the effective selling price per unit is saved.

Create an UML Class Diagram.

Below the text box, there are two buttons: "Save changes" and "Cancel".

The bottom section of the page is titled "WebMF - Class Diagram Editor" and includes a "Show XML" button. It features a "Stencil Set" on the left with the following elements: "Class", "Association", "Generalization", "Attribute", and "Operation". The "Diagram" area in the center is empty. On the right, there is a "Properties" panel with a table showing "Name" and "Value" for the selected element. The table is currently empty, with the header "No element selected" above it.

Figure 5.23: Screenshot of a WebMF Moodle assignment activity



## 6 Related Work

This chapter describes related work in the field of software applications for the visualization and manipulation of models. The Oryx Editor (Section 6.1), SLIM (Section 6.2), and the Web 2.0 Metamodel Browser (Section 6.3) are web based applications, the 2D Meta Model Browser (Section 6.4) is a desktop application. Finally, the presented applications are compared with the approach of this thesis (Section 6.5).

### 6.1 Oryx Editor

The Oryx Editor [42] is a browser based generic model editor which features the modeling of business processes with the standardized Business Process Modeling Notation (BPMN) [33]. It is part of a bigger project called “BPM tool chain” [13] which enables process model instantiation and process instance execution on the web.<sup>1</sup>

The Oryx Editor allows loading and saving diagrams, and the export to various formats. The user interface features manipulation of diagram elements via drag-and-drop and arrangement/alignment functions. Diagrams can be zoomed in and out. Additionally, diagram manipulation steps can be undone and redone.

#### 6.1.1 Metamodel

The editor is implemented in a generic way: The flexible interfaces allow to extend the functionality of the editor and to add other process modeling languages (for example BPEL [24], Petri Nets [36] and FMC<sup>2</sup>) or completely other modeling languages like UML. The specifications of the modeling language’s elements are called stencil sets. These *stencil sets* include the stencil descriptions and a set of rules.

A *stencil specification* contains an element type (node or edge), a unique identifier, the stencils title and a description text. Furthermore, the graphical representation (the SVG [11] file path) and the icon for the toolbox can be specified. It also contains a list of groups to which this stencil belongs, a list of roles (used for specifying rules), and a set of properties that can be set for instances of this stencil.

There are multiple types of *rules* for stencil sets: Connection rules specify which types of nodes can be connected. Cardinality rules specify how much instances of a specific type have to occur in a model. Containment rules describe parent-child relationships of nodes. Rules extension and stencil set extensions provide programmatic mechanisms for rules, which cannot be described with the three basic types of rules. Each rule belongs to a role or a stencil id, which also specifies a unique role. With this concept of stencils and rules, the Oryx Editor provides a powerful mechanism for the description of metamodels.

---

<sup>1</sup>A demo of the Oryx Editor is available at <http://bpt.hpi.uni-potsdam.de/Oryx>

<sup>2</sup>FMC: <http://www.fmc-modeling.org/>

### 6.1.2 Technologies

The client application is based on an XHTML<sup>3</sup> page which runs in the browser. There, GUI components are implemented with the EXT library<sup>4</sup> and the diagram is displayed with Scalable Vector Graphics (SVG). This is an XML based format for the description of vector graphics and is used for displaying the graphical diagram elements. It was preferred to Adobe Flex and the HTML Canvas Tag due to its declarative specification, its availability as an open standard, and the free Integrated Development Environment (IDE). For the manipulation of the diagram (i.e. the Document Object Model (DOM) [10]), JavaScript including the prototype.js library<sup>5</sup> is used.

The diagram data is stored as eRDF annotated XHTML, which is a syntax for writing HTML in a manner that the information in the HTML document can be extracted into RDF. RDF itself is a graph based formal language for the description of a resource. The server application is based on an Apache Tomcat Server<sup>6</sup> which provides RESTful web services [37]. It provides the stencil sets that are described in JSON, a compact text based data format.

### 6.1.3 Communication and Architecture

The client application is divided into three layers (Figure 6.1): The application is embedded in the browser sandbox which contains the RDF metadata (layer 1). Since multiple applications can be located in one page and access its data store, a Data Manager is used to synchronize data and avoid data loss (layer 2). The Oryx application logic (“Oryx” box) is used for viewing and editing functionality and to load stencils from the Stencil Repository Server (layer 3).

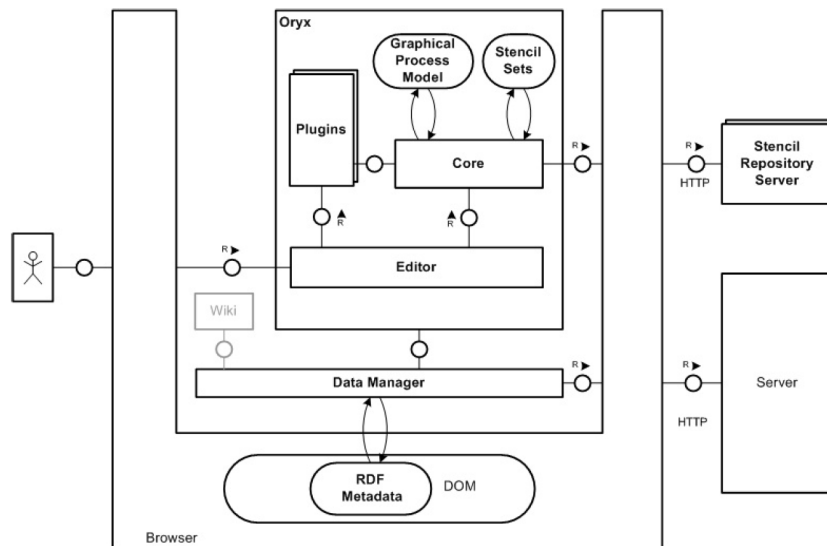


Figure 6.1: Architecture of the Oryx client

Altogether there are three different layers which hold data: the instantiated shapes and their properties, the RDF data in the DOM, and the persistent data storage on the server (the Hibernate layer on the tomcat server, shown in Figure 6.2).

<sup>3</sup>Extensible Hypertext Markup Language (XHTML) [35]

<sup>4</sup>EXT JS: JavaScript GUI library, <http://www.extjs.com>

<sup>5</sup>prototype.js JavaScript library: [www.prototypejs.org](http://www.prototypejs.org)

<sup>6</sup>Apache Tomcat web server: <http://tomcat.apache.org>

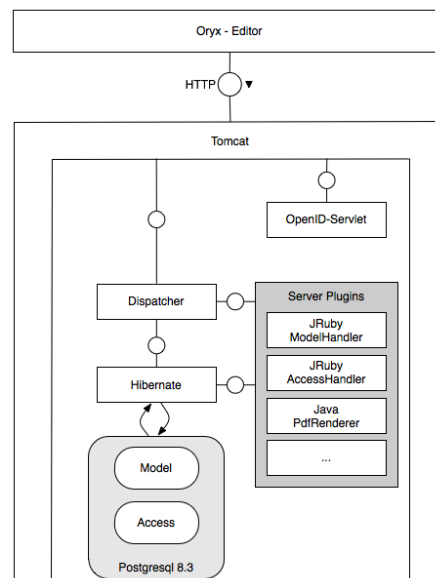


Figure 6.2: Architecture of the Oryx server

The data manager uses AJAX requests through HTTP to synchronize resources between server and client. It saves (PUT), reloads (GET) or deletes (DELETE) them using RESTfull services. The underlying data format of the communication is eRDF annotated XHTML. It also provides event listeners on the managed resources. These listeners are used by the application logic layer to keep the graphical representation up to date.

### 6.1.4 Usability

Figure 6.3 shows the Oryx client application that is running in a web browser. On the top there is a tool bar with common loading, viewing and editing functionality. The shape repository on the left side allows to add items of the loaded stencil set via drag-and-drop. The diagram area is located in the center. The properties bar on the right side allows for viewing and editing the item which is selected in the diagram area.

The Oryx Editor provides very intuitive modeling editing. Drag-and-drop mechanisms are advanced with the concept of dockers and magnets. Dockers are reference points of elements (nodes or edges) to “dock on” other elements. Magnets are areas around the dockers which help to “snap in” into dockers of target elements.

Figure 6.4 illustrates this functionality: The docker (red point on the arrow’s vertex) “snaps in” into a docker when it is dragged over the magnet area around it, i.e. the connection point is automatically displayed at the exact position of the docker. Additionally, alignment functions help to arrange elements clearly. The tool bar provides adding all stencils that are defined in the loaded stencil set. And hovering over an element produces a menu instantly displaying all outgoing elements allowed in the stencil set. The graphical user interface components are implemented with the EXT JavaScript library, which provides desktop feeling for browser applications.

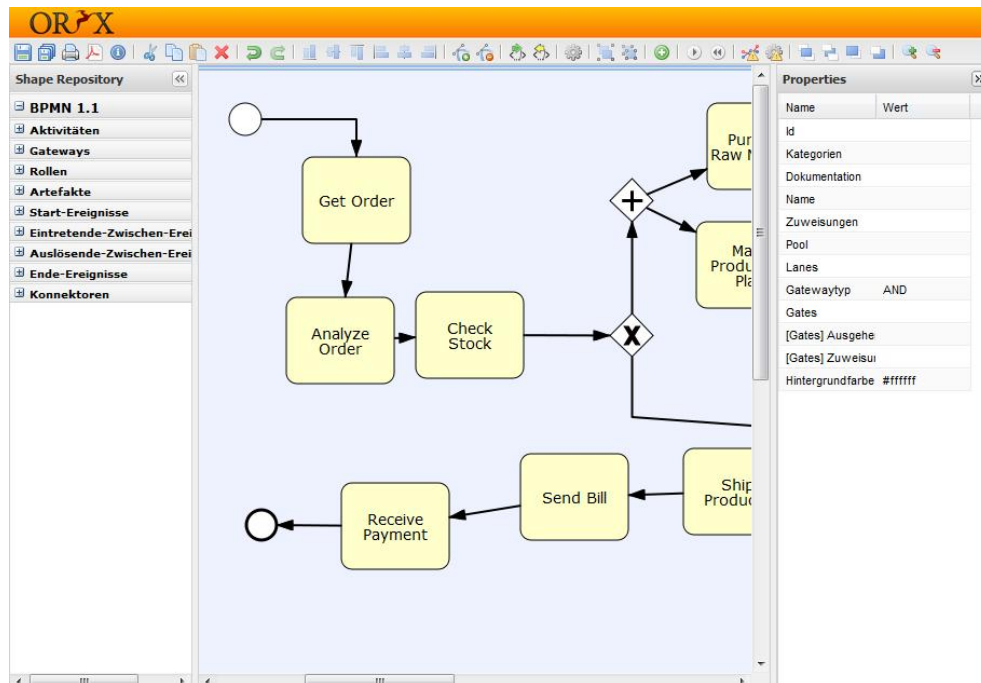


Figure 6.3: Oryx client application

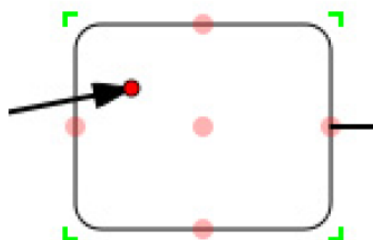


Figure 6.4: Dockers and magnets

### 6.1.5 Rating

The Oryx Editor allows for editing BPMN-based models with very good usability and features an innovative Dockers and magnets concept. But the implementation is targets the specific abilities of the Firefox web browser.

## 6.2 SLIM

SLIM (“Synchronous Lightweight Modeling”) [41] provides synchronous distributed creation and manipulation of UML diagrams within the browser sandbox. In general, it is a lightweight environment for collaborative work.

The approach of SLIM is that the development of diagrams in software design is an inherently collaborative activity with involvement of software developers, domain experts, and other stakeholders. Thus they try to provide a collaborative solution that allows people to participate without barriers. This is accomplished by the usage of the native browser functionality.

### 6.2.1 Metamodel

The presented SLIM prototype covers the creation and manipulation of UML class diagrams. It supports packages, classes, interfaces, attributes, operations, stereotypes, and notes. Unfortunately, none of the documentation provides an insight about how these UML constructs are represented in a metamodel. The documentation focuses the challenges in the fields of collaboration in Rich Internet Applications. However, the SLIM prototype facilitates the export to XMI 2.1 with the UML standard 2.1.2.

### 6.2.2 Technologies

SLIM avoids the need of installing a browser plug-in like the Flash Player or Java Applets by relying on functionality that is supported natively by major browsers. To display the notation elements with browser native functionality, three alternatives had been investigated. The first alternative, the emulation of vector graphics via HTML Div-elements does not satisfy the needs due to low performance and high memory consumption. The second and third alternatives were the HTML 5 Canvas element and SVG [11]. The availability of these two in modern browsers is equivalent. Since there already are several implementations for displaying UML via SVG (e.g. `uml2svg`<sup>7</sup>), this technology was chosen.

The most used web browser, Microsoft’s Internet Explorer, does not support SVG but the equivalent VML (Vector Markup Language). This problem was tackled by the use of the graphics library of the Dojo Toolkit<sup>8</sup>, which provides a uniform programming interface on top of SVG and VML.

There were also problems while implementing a cross-browser procedure to handle events for graphical elements (e.g. responding to mouse gestures). SVG and VML generally provide event handling, but the browser support is limited. The solution was to put a transparent Div-Element on top as an extra layer which invokes DOM events.

Overall the solution for the user interface is a mix of technologies. There is no common API for user input and visualization, which often leads to problems in development, maintenance, and execution. In contrast, WebMF has a common interface

<sup>7</sup>uml2svg project: <http://uml2svg.sourceforge.net>

<sup>8</sup>Dojo Toolkit: <http://www.dojotoolkit.org>

for all GUI aspects including visualization and event handling. For the client-server communication “comet”-technologies were used which are described in the next section.

### 6.2.3 Communication and Architecture

For SLIM, a Rich Client architecture had been taken over a Thin Client architecture due to the need for high responsiveness in collaborative editing of diagrams. Concretely, this means that the application logic is placed on the client (Figure 6.5).

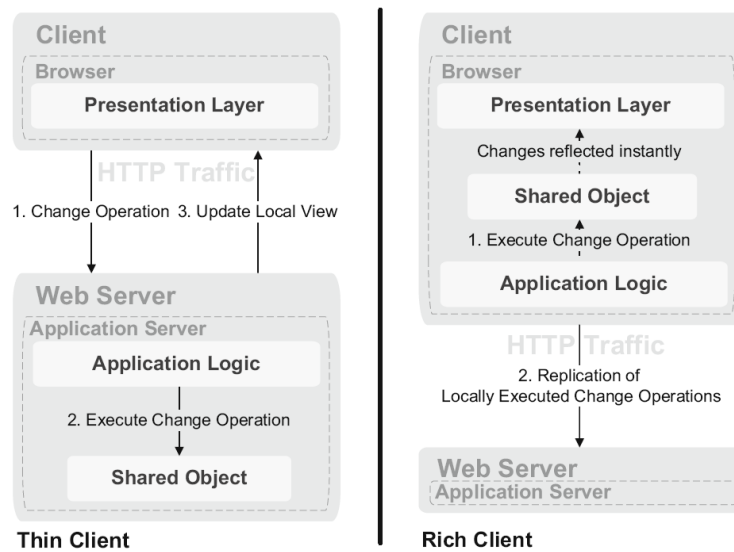


Figure 6.5: SLIM: Thin Client vs. Rich Client [41]

The displayed and manipulated model is placed central on the client (Figure 6.6). Also the edit, user interface, event, and collaboration components are placed on the client and interact with the model. The collaboration component facilitates replication of the change operations of the model on the server.

For the collaboration the server has to inform all clients on updates of the replicated model. The HTTP protocol is built upon the request/response paradigm and therefore stateless. There are a few techniques to tackle this problem, together known as “Comet”. SLIM uses the long-polling approach. It utilizes a so-called persistent or long-lived HTTP connection. Connections to the server are kept open until either a server-side event or a timeout occurs.

### 6.2.4 Usability

The user interface of the SLIM UML editor builds upon three components (Figure 6.7). Centrally, the diagram area visualizes current UML model and allows users to edit it. Resizing as well as repositioning (“move” operation) of elements is supported. On the left, a tool bar facilitates the creation of new elements. On the right, the properties panel shows all properties of the currently selected element and allows for editing them. Further, a chat is offered for the communication of all users that currently view or edit the diagram. Together they form a comfortable user interface for the manipulation of diagrams.

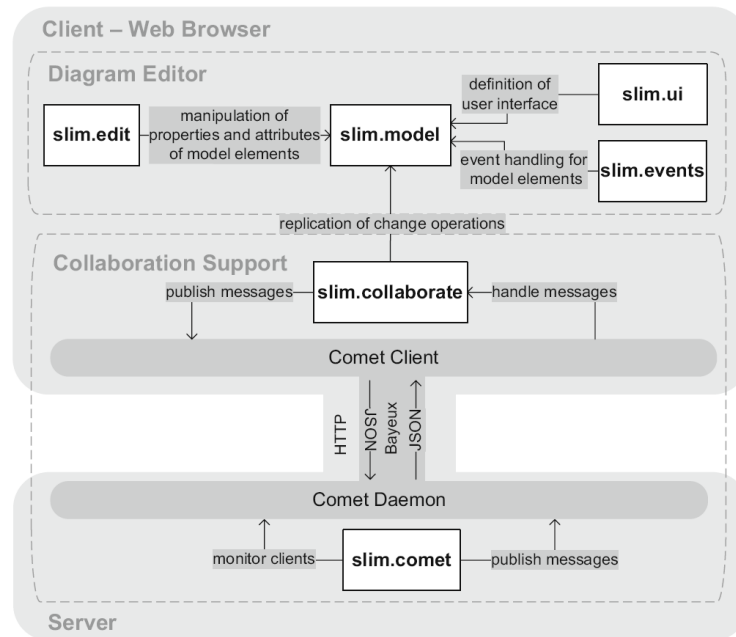


Figure 6.6: SLIM client server architecture [41]

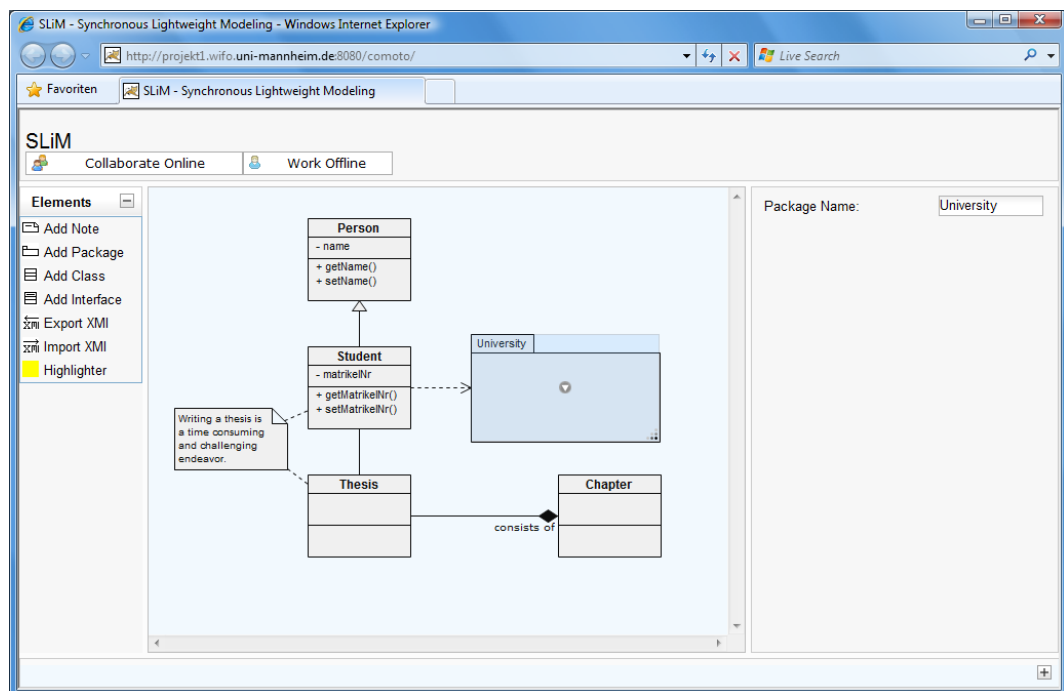


Figure 6.7: Screenshot of the SLIM UML class diagram editor [41]

### 6.2.5 Rating

SLIM brings modeling and collaboration together within the browser environment. It was accomplished to provide a sophisticated user interface and to address all modern web browsers. It can be discussed, whether the user interface technology mix with dependence on SVG / VML is better than the dependence on the Flash Player Plugin or Java Applets.

## 6.3 Web 2.0 Metamodel Browser

The Web 2.0 Metamodel Browser [15] is a Rich Internet Application which provides an AJAX-based tree-viewer for efficiently browsing Ecore based metamodels (M2) and their models (M1). It is an emulation of the Ecore Sample Editor from the EMF framework<sup>9</sup>.

### 6.3.1 Metamodel

The application uses the Eclipse Modeling Framework (EMF) as its metamodel (M3). Any model can be browsed when its metamodel is defined with Ecore. The displayed information about the model items are provided with the EMF label provider and content provider<sup>10</sup>.

### 6.3.2 Technologies

The following technologies are used by the Web 2.0 Metamodel Browser: The dhtmlxTree<sup>11</sup>, an AJAX [16] powered DHTML JavaScript Tree component with rich API, is used for the visualization of the models. It is embedded in the HTML page. It interacts via XmlHttpRequest, XML, JavaScript, and JSON [12]. The server is based on Java servlets<sup>12</sup> and JSP<sup>13</sup> that run on an Apache Tomcat 5.5. Additionally, the Eclipse Platform<sup>14</sup>, EMF and EMF.Edit [8] are used in the server environment.

### 6.3.3 Communication and Architecture

The architecture of the Web 2.0 Metamodel Browser is shown in Figure 6.8: The main application parts are the Servlets and JSPs which are located on the Tomcat web server. They handle the AJAX - requests of the client and store the serialized model data in files on the web server. The process concerning the Eclipse Platform (from the .genmodel file to the JAR-file) is necessary for browsing models (M1) related to a specific metamodel. To provide a specific metamodel, jar files with the specific EMF and EMF.Edit code have to be generated and put on the web server.

The communication between the server and the client is realized with AJAX requests, using XML or JSON as data format. Browsing a model starts with displaying an HTML site with the tree component which only shows the root item. Zooming into an item leads to dynamically loading the item content with an AJAX request. When clicking an item, the editor shows the details of the item in the properties view (also AJAX-based).

<sup>9</sup>The Web 2.0 Metamodel Browser is available at <http://metamodelbrowser.org/>

<sup>10</sup>The basics of EMF and Ecore have been introduced in Chapter 2.1.3

<sup>11</sup>dhtmlxTree JavaScript GUI library: <http://scbr.com/docs/products/dhtmlxTree>

<sup>12</sup>Java Servlet Technology: <http://java.sun.com/products/servlet/>

<sup>13</sup>Java Server Pages: <http://java.sun.com/products/jsp/>

<sup>14</sup>Eclipse IDE: <http://www.eclipse.org/>

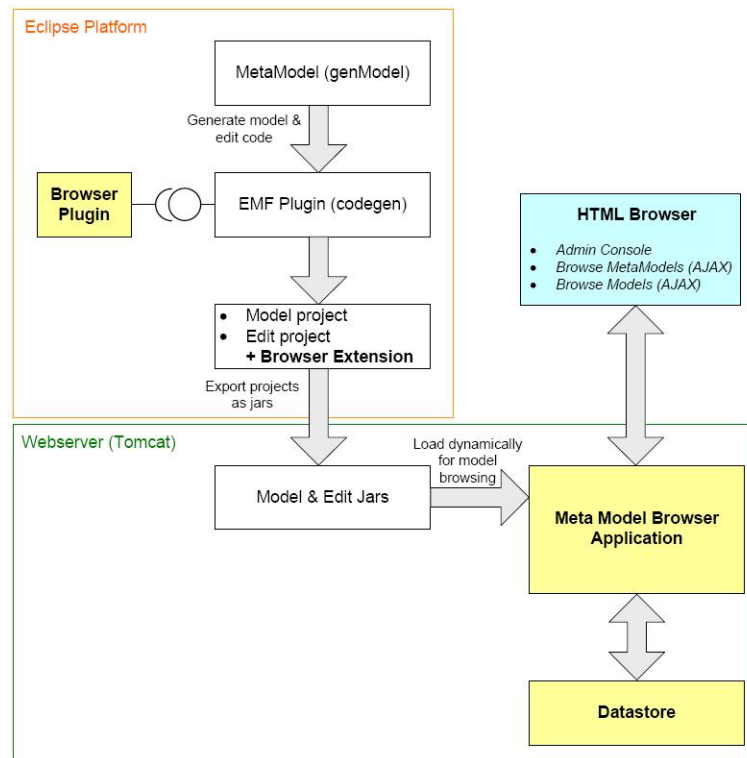


Figure 6.8: Basic architecture of the Web 2.0 Metamodel Browser

### 6.3.4 Usability

Figure 6.9 shows the model view of the Web 2.0 Metamodel Browser and the Ecore Sample Editor. The functionality is simple and clear. It features expanding and collapsing tree nodes, specific icons for tree elements. Once clicked on a tree item, the properties view shows all name-value-pairs of the selected item.

### 6.3.5 Rating

The strengths of the Web 2.0 Metamodel Browser are located in the generic approach using EMF-based metamodeling and the good performance. But the tree view represents models not as clear as a diagram, especially when there are many relationships between elements. And the application does not support editing of models at all.

## 6.4 2D Meta Model Browser

The 2D Meta Model Browser [27] is a generic graphical browser for metamodels. It is a desktop application which visualizes metamodels specified with Ecore or MOF. Therefore it uses an UML Class Diagram-like Notation. The application is integrated in the Model Transformation for Verification (MTV) project [20] from the University of Geneva.

### 6.4.1 Metamodel

The graphical representation in the 2D Meta Model Browser is created via a two-step transformation process (Figure 6.10). In the first step, an Ecore or MOF metamodel

## 6 Related Work

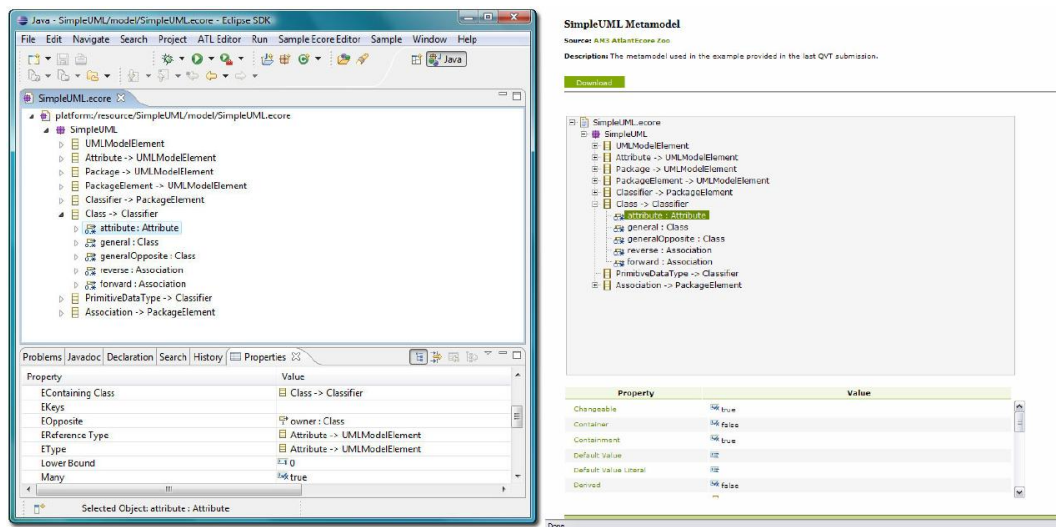


Figure 6.9: Ecore Sample Editor (left) and Web 2.0 Metamodel Browser (right)

(stored in an XMI-File) is transformed to the created intermediate format SVG4MTV. This is needed to cope with the discrepancies between MOF and Ecore. The resulting format SVG4MTV describes graphical diagram objects with their spacial information. Thus it summarizes the logical diagram elements. In the second step, the model expressed in the intermediate format is transformed to SVG. Each logical diagram element is transformed to core SVG graphics that can be displayed in the Graphical User Interface.

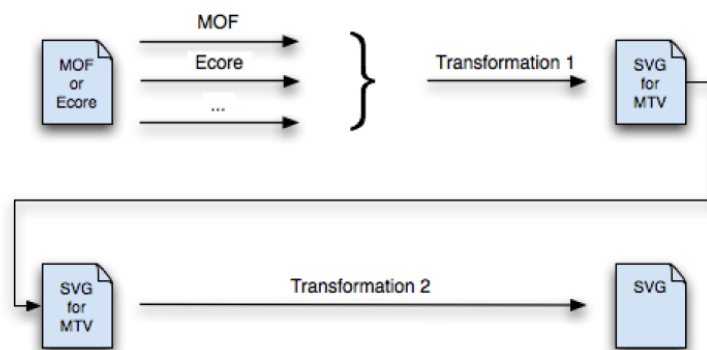


Figure 6.10: Two-step transformation of the 2D Meta Model Browser [27]

### 6.4.2 Technologies

The 2D Meta Model Browser is based on the Java programming language. The Eclipse Modeling Framework (EMF) library [8] is used for the metamodeling functionality. The visualization is based on Scalable Vector Graphics (SVG) [11], Batik (a Java based toolkit for applications or applets that use SVG)<sup>15</sup>, and JGraph (a Java framework for graph visualization and layout)<sup>16</sup>.

<sup>15</sup>Batik SVG Toolkit: <http://xml.apache.org/batik/>

<sup>16</sup>JGraph Java Open Source Graph Drawing Component: <http://www.jgraph.com>

### 6.4.3 Communication and Architecture

The 2D Meta Model Browser is a single desktop application. There is no documentation or indication for communication functionality.

### 6.4.4 Usability

The user interface of the 2D Meta Model Browser allows repositioning of the diagram nodes via drag-and-drop, zooming in and out, and changing the level of details. Changing the level of details means to choose which diagram element types are visible. For example, in the case of UML Class diagrams, methods and attributes of classes could be hidden to provide a better overview of the existing classes. Further, diagrams can be loaded, saved, and exported to various image formats.

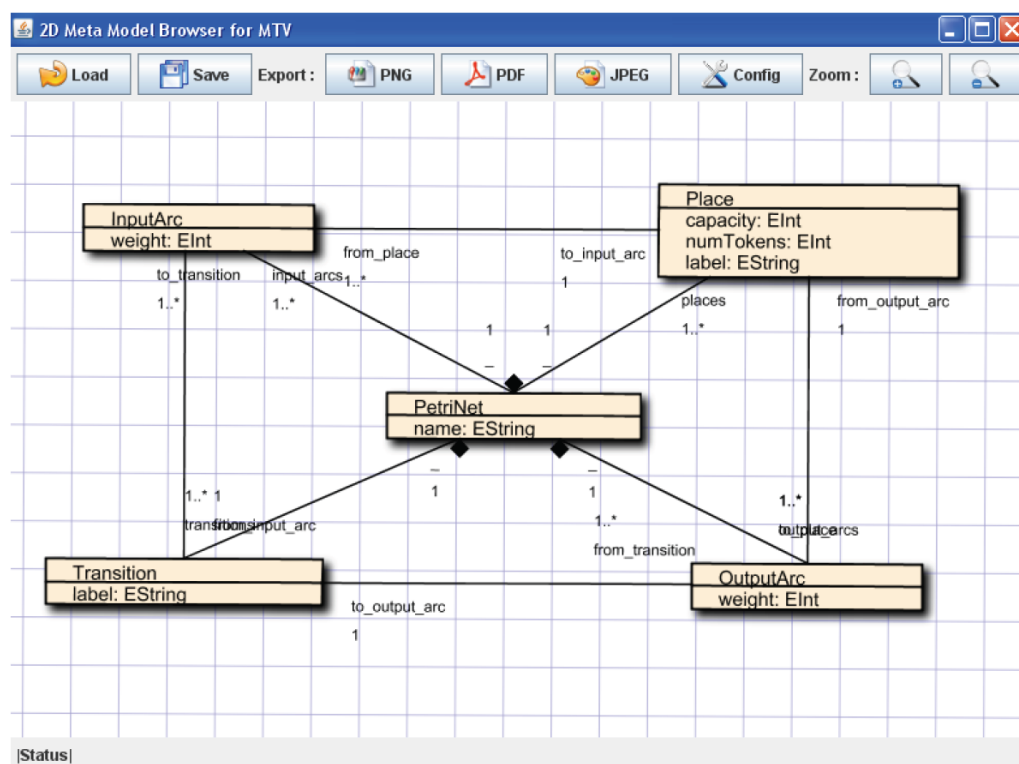


Figure 6.11: 2D Meta Model Browser [27]

### 6.4.5 Rating

The 2D Meta Model Browser features a clear metamodel mechanism with model transformations. The whole application is based on a uniform platform (Java). The editor has no model editing functionality and a rudimentary user interface.

## 6.5 Comparison

A comparison of the approaches presented in this chapter with WebMF is presented in Table 6.1. Some of them are only for the visualization of (meta-) models, others

## 6 Related Work

allow also editing. Though they all are targeted on different goals, some similarities are interesting:

All approaches except the Web 2.0 Metamodel Browser display metamodels as two-dimensional diagrams. The tree-based approach suffices for simple metamodels, for more complicated metamodels the diagram view is much clearer.

The metamodel of WebMF and the Oryx Editor provide proprietary languages which allow to create “stencil sets”. The Web 2.0 Metamodel Browser and the 2D Meta Model Browser are based on Ecore. The Ecore metamodel is a de-facto standard and a functionality for transforming models to this standard is also desirable for WebMF.

Rich Internet Application technologies are used in four of five approaches. SLIM, the Web 2.0 Metamodel Browser and the Oryx Editor use AJAX for the user interface, the latter exhibits well-known browser compatibility problems. WebMF uses Adobe Flex as a result of an evaluation of RIA technologies. The implementation with this technology was quite comfortable with no technology-related bugs.

	<b>WebMF</b>	<b>Oryx Editor</b>	<b>SLIM</b>	<b>Web 2.0 Metamodelbrowser</b>	<b>2D Metamodelbrowser</b>
<b>Focus</b>	Generic diagram editing	Business process modeling	Collaborative UML diagram editing	Generic browser for models and metamodels	Displaying MOF and Ecore metamodels
<b>Manipulation supported</b>	Yes	Yes	Yes	No	No
<b>Visualization</b>	Diagram	Diagram	Diagram	Tree	Diagram
<b>Metamodel</b>	Stencil set language	Stencil set language	-	Ecore	Ecore
<b>Visualization technology</b>	Adobe Flex	Scalable Vector Graphics (SVG)	Scalable Vector Graphics (SVG), Vector Markup Language (VML)	AJAX, dhtmlXtree	Java, Scalable Vector Graphics (SVG)
<b>Major pros</b>	Quick creation of editors with specific notation elements	Rich UI (Dockers/Magnets)	Collaborative manipulation of diagrams	Displays all elements of arbitrary models	Clear model and transformations
<b>Major cons</b>	Metamodel limited to node-subnode-edge schema	Browser problems with SVG and JS	Technology Mix	Unclear for models with many relationships	Poor UI

Table 6.1: Comparison of related work

# 7 Evaluation

In this chapter, the implemented framework is evaluated. First, the implementation effort for building editor applications using the WebMF framework is analyzed (Section 7.1). Then, the realization of the framework features defined in the requirements catalog is evaluated (Section 7.2). After that, the issues that occurred during the implementation of the UML editors are presented (Section 7.3). Finally, the results are summarized (Section 7.4).

## 7.1 Implementation Effort for WebMF Applications

In the course of this thesis, a total of four applications have been realized using the WebMF framework. For a WebMF-experienced developer the implementation takes about 100 to 120 minutes for one application. This is pretty little effort, particularly when compared to the implementation effort it would cost when developing the whole editor from scratch.

TaiPan		UML Class Diagram		UML State Diagram		UML Sequence Diagram	
<i>Stencil Set</i>	90	<i>Stencil Set</i>	118	<i>Stencil Set</i>	115	<i>Stencil Set</i>	82
Ship	18	Class	60	State	31	Lifeline View	53
Item	11	Attribute	19	Activity	11	SynchronousMessage	13
Port	30	Operation	20	InitialState	17	AsynchronousMessage	11
Route	7	Association	12	EndState	21		
ShipDestinationPort	7	Generalization	11	Transition	11		
<b>Total</b>	<b>163</b>	<b>Total</b>	<b>240</b>	<b>Total</b>	<b>206</b>	<b>Total</b>	<b>159</b>

Table 7.1: Lines of code of the implemented WebMF applications

For realizing WebMF applications, developers need only basic knowledge in XML (for implementing the metamodel) and Adobe Flex (for implementing the views).

Table 7.1 shows the Lines of Code (LOC) of the developed applications <sup>1</sup>. For each application, one file for the structure (Stencil Set) and one file per notation element have to be implemented. The files for the stencil set definitions have about 100 LOC, with a clear structure. The LOC of stencil node views reach from 11 (Item View in TaiPan Diagram, Activity View in UML State Diagram) to 60 (Class View in UML Class Diagram). Node views that are based on the `UIComponentNodeView` (e.g. Lifeline View in UML Sequence Diagram) view base class are commonly more complex than those which are based on the `BoxNodeView` base class. All stencil edge views can be implemented without much effort (7 to 12 LOC).

<sup>1</sup>The listings in Chapters “Sample Application” and “UML Applications” are partly shortened.

## 7.2 Evaluation of the Requirements Catalog

At the beginning of this thesis (Section 3.1) a requirements catalog was created for WebMF. The realization of the features including usability issues is evaluated here.

### 7.2.1 Diagram

#### Display Nodes, subnodes, and edges

All elements are displayed correctly. Some issues appeared during the implementation of applications. (Chapters 4, 5).

In some cases, it is a problem, that nodes cannot be resized manually (e.g. for UML Sequence Diagram lifelines).

The framework does only support two types of adjustment: subnodes to parent nodes, and edge endpoints to nodes. Other adjusting mechanisms can not be implemented (some elements could not be realized in the UML Sequence Diagram editor).

Also edge labeling features are missing: Labels that are automatically positioned beside an edge (at start point, end point, or central) that can show properties of the edge (e.g. name of association in a UML class diagram).

Finally, edges that have the same node as source and target are not displayed properly (feature was explicitly not requested). The common solution of this issue is the use of edge bend points.

#### Move nodes (drag-and-drop)

(Top level-) Nodes can be moved from one point in the diagram to another by dragging them. While dragging, the node is also shown (semitransparent) at the potential new position.

If the selection of multiple elements would be supported, the user also could move multiple elements at once. Also, moving edges via drag-and-drop should be supported.

#### Connect edges to nodes (drag-and-drop)

Edges can be connected and disconnected to/from nodes by dragging an endpoint onto the node. A symbol indicates, if the specific connect operation is allowed or forbidden in the specific metamodel. Once an edge is connected to a node it is readjusted when the position or size of the node changes.

An issue is that while dragging an edge endpoint, the edge is not displayed with the new endpoint position. This behavior would be expected.

#### Delete nodes and edges (DEL key)

When a node, subnode, or edge is selected in the diagram and the user presses the DELETE key than this element is deleted. When a node is deleted, also all subnodes are deleted. Edges that are connected to the node are disconnected.

If the selection of multiple elements would be supported, the user also could delete multiple elements at once. An alternative to disconnecting edges from nodes that are deleted is to cascade deletion.

### **Edges that connect two nodes**

Edges that connect exactly two nodes (source node and target node) have been implemented.

### **Edges that connect more than two nodes**

Edges that connect more than two nodes are quite unusual and have not been implemented.

### **Connection of edges on different points of nodes**

This feature was implemented properly. Details are described in Section 3.3.2.

### **Drag and Drop usability: different mouse cursors**

This feature was implemented properly. Details are described in Section 3.4.2.

### **Select element (highlight)**

If an element (node, subnode, or edge) is selected, it is highlighted.

The selection of more than one element for moving / or deleting them is desirable for faster editing. This could be done with a rectangular selection tool with which the user can mark several elements at once.

### **Hover highlighting**

This feature was implemented properly. When the user moves the mouse cursor over an element (node, subnode, or edge), the element is highlighted.

### **Different arrow types for edges**

This feature was implemented properly. Edge arrows can be displayed with different styles: The size and the angle can be specified. And the arrow can have block (3rd line) style and/or can be filled (arrow is filled black).

### **Displaying property labels in nodes**

This feature was implemented properly. Binding expressions can be used to show the current values.

### **Displaying images in nodes**

This feature was implemented properly. The `mx:Image` component can be included in node views.

## **7.2.2 Tool bar**

### **Add new nodes/edges from tool bar (drag-and-drop)**

This feature is implemented properly.

### Show icons of elements in tool bar

This feature is implemented properly.

### 7.2.3 Properties Editor

#### Show properties of selected element

This feature is implemented properly.

#### Edit properties of selected element

The feature is implemented. Since the framework's metamodel does not support data types for element properties, the property editor cannot validate property values.

### 7.2.4 Import / Export

#### Import / export interface for browser sandbox

This feature is implemented properly (see Section 5.4).

#### Import / export to file system

This feature was not implemented.

## 7.3 Lessons Learned

In Chapter 5 three UML editor prototypes were implemented. The issues that appeared during implementation are summarized in the following.

Two of the three UML editor prototypes could be implemented satisfactorily. The class diagram and the state diagram, which have a similar optical appearance fit into the schemes of WebMF and only showed some possibilities for future enhancements of WebMF. In contrast, the sequence diagram could not be realized as desired and spotlighted limitations of the framework. In the following the issues that appeared during the implementation are summarized and categorized.

Recommendations for enhancements are features that could extend the existing framework, which can be easily implemented in future versions. On the other side, limitations are problem areas that impede the realization of specific diagram functionality. This functionality can only be realized, when the basic functionality of the framework is changed or adapted.

### 7.3.1 Recommendations for Enhancement

Data types should be provided for element properties, the current solution that stores all property values as string literals is error-prone. Also, additional rule types should be provided. For example, one would want to limit the number of diagram elements of the same type, such as OCL constraints<sup>2</sup> for defining well-formedness rules. For specific node types, the size has to be set by the user, instead of the automatic adjustment to included elements. To support this, user interface has to support resize operations, and the size of nodes has to be persisted in the underlying data structures. In many

---

<sup>2</sup>OCL: A OMG standard language for defining rules on UML diagrams.

diagram types attributes of edges are shown nearby edges, e.g. the association name of UML class diagram. This “edge labeling” functionality is not yet supported and recommended for future work. Finally, it occurs that diagram elements are almost equivalent in their behavior. A facility for the implementation of inheritance between syntax definitions would help to avoid redundancy.

### 7.3.2 Limitations

Sometimes, diagram elements relate to each other, although they are not connected in the visualization, e.g. classes in class diagrams that are used as data types of attributes. These cannot be linked together, i.e. only one element of the concrete syntax can refer to one element in the abstract syntax. There are no mapping facilities provided by the framework since a basic idea was to avoid complicated mappings and to focus on the visualization features.

Parent-child relationships are limited twofold: First, there is no way for specifying the location of a child node inside the parent node. The children of a node are just shown one after the other, in the order of the adding as children. Second, an element can only be a child of exactly one parent. Sometimes, the alignment of an element is affected by more elements.

Currently, there are two types of composition mechanisms that are supported by WebMF: parent-child relationships and the connection of two nodes with an edge. Other relationships could be established beside them. But it has to be taken into account that more flexibility in the application often causes more complexity and more implementation effort. The framework supports a few simple constructs, with which editors can be built with little effort. A solution with a compromise of the aspects is left as subset for future work.

## 7.4 Summary

The realization of WebMF editor applications is rapid and straightforward. Commonly, only little programming effort is needed. Developers need only basic knowledge in XML (for implementing the metamodel) and Adobe Flex (for implementing the views).

The editors that were built in the course of this thesis show that the framework can be used for a variety of diagram types. WebMF concentrates on simple structures of diagrams with nodes, subnodes, and edges. Other, more complicated structures are not supported. To allow them, developers would have to extend or adapt the framework.

Limitations of the framework are located at the *abstract syntax*. The parent-child relationship is limited so that a node can only be child of one parent. Other composition mechanisms that allow nesting of nodes are not supported. Further, there are no mapping mechanisms provided by WebMF. The use of the Flex MXML components for defining the *concrete syntax* is quite powerful and universal.

The integration of WebMF based diagram editors into the Moodle e-Learning platform works without problems. Together, this allows quick creation of e-Learning resources.



# 8 Conclusion

This chapter summarizes this thesis (Section 8.1) and points out problem areas for future work (Section 8.2).

## 8.1 Summary

E-Learning is on the advance, also in teaching UML. The usage of a modeling tool that allows for aligning the notation elements to the taught contents promises a better understanding of the practical application of UML.

The major goal of this thesis was to develop a web-based modeling tool that is integrated into the learning platform of the Vienna University of Technology. A generic, metamodeling based approach was claimed, where complete editors can be created by specifying the structure of the diagram elements (abstract syntax) and the visual notation elements (concrete syntax).

Modeling tools have a rich and highly responsive user interface. To provide a modeling tool on an e-Learning platform — running directly in the browser environment — requires the use of Rich Internet Application (RIA) technologies. It was evaluated, which technology is best-suited for the specific requirements. Further, E-Learning basics and e-Learning platforms have been studied and the use of a modeling tool for teaching UML has been reasoned.

By the use of the chosen RIA technology Adobe Flex the Web Modeling Framework (WebMF) was developed. It allows for the creation of diagram editors by specifying the structure of a diagram type and the visual appearance of the notation elements. Therefore, it features generic diagram editing functionality and provides two domain specific languages (DSLs) for the definition of the diagram types.

WebMF was employed for the creation of four applications, one example application for the demonstration of the framework’s functionality and three UML diagram editors for teaching UML, which also showed strengths and weaknesses of the framework. The strength of WebMF is the rapid and easy creation of diagram editors with only little programming skills and effort.

Limitations of the framework are located at the *abstract syntax*. The parent-child relationship is limited so that a node can only be child of one parent. Other composition mechanisms that allow nesting of nodes are not supported. Further, there are no mapping mechanisms provided by WebMF. The use of the Flex MXML components for defining the *concrete syntax* is quite powerful and universal.

Finally, for the integration into the Moodle e-Learning platform, a plug-in was developed. This plug-in allows creating “practical modeling exercise”-assignments which are completely handled by the learning platform.

## 8.2 Future Work

For future work, several interesting topics arise. Some improvements in the field of metamodeling are suggested: The created framework could be complemented with ele-

## 8 Conclusion

ment structures that are currently not supported. By the use of model transformation, the serialized models of the drawn diagrams could be transformed to data models that can be reused in other applications, e.g. commercial UML editors. Furthermore, the export of diagrams to the model exchange formats XMI [31] and XMI-DI [32] are preferable.

It is also suggested that the developed UML editors are enhanced and used within the eLearning platform of the “Object Oriented Modeling (OOM)” course at the Vienna University of Technology. Lecturers as well as students shall comment the application of the tool in an evaluation. The results of the course rating tool of the TU Vienna <sup>1</sup> will give further feedback.

---

<sup>1</sup>Students rate courses they attend: [http://www.tuwien.ac.at/lehre/evaluation/lvbw\\_handbuch](http://www.tuwien.ac.at/lehre/evaluation/lvbw_handbuch)

# Bibliography

- [1] ADOBE SYSTEMS, INC. Flex quick start: Building custom components.  
[http://www.adobe.com/devnet/flex/quickstart/building\\_components\\_using\\_code\\_behind/](http://www.adobe.com/devnet/flex/quickstart/building_components_using_code_behind/), August 2008.
- [2] ADOBE SYSTEMS, INC. Flash player penetration.  
[http://www.adobe.com/products/player\\_census/flashplayer/](http://www.adobe.com/products/player_census/flashplayer/), 2009.
- [3] ATKINSON, C., AND KÜHNE, T. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.* 12, 4 (2002).
- [4] BARNES, D. J., AND KÖLLING, M. *Objects First with Java—A Practical Introduction using BlueJ*. Pearson Education, Edinburgh Gate, UK, 2003. Hinweise, Materialien, Software, Diskussionsforen, etc.  
<http://www.bluej.org/objects-first/>, biburl = <http://www.bibsonomy.org/bibtex/2349a55484979776ac66031fe469ae014/n770>, keywords = imported.
- [5] BAUMGARTNER, P., HÄFELE, H., AND MAIER-HÄFELE, K. *E- Learning Praxishandbuch. Auswahl von Lernplattformen*. Studien Verlag, September 2002.
- [6] BÉZIVIN, J. On the unification power of models. *Software and Systems Modeling* 4, 2 (May 2005).
- [7] BRANDSTEIDL, M., SEIDL, M., WIMMER, M., HUEMER, C., AND KAPPEL, G. Teaching models @ big - how to give 1000 students an understanding of the uml. Warsaw University of Technology.
- [8] BUDINSKY, F., BRODSKY, S. A., AND MERKS, E. *Eclipse Modeling Framework*. Pearson Education, 2003.
- [9] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc. New York, NY, USA, 1996.
- [10] CHANG, B., LITANI, E., KESSELMAN, J., AND RAHMAN, R. Document Object Model (DOM) Level 3 Abstract Schemas Specification. World Wide Web Consortium, Note NOTE-DOM-Level-3-AS-20020725, July 2002.
- [11] CHRIS LILLY, D. J. Scalable Vector Graphics (SVG).  
<http://www.w3.org/Graphics/SVG/>, 2004.
- [12] CROCKFORD, D. RFC4627: JavaScript Object Notation, 2006.
- [13] CZUCHRA, M., PETERS, N., POLAK, D., AND TSCHESCHNER, W. BPM Toolchain - Oryx in Action.  
[http://images.apple.com/science/poster/pdf/130\\_czuchra.pdf](http://images.apple.com/science/poster/pdf/130_czuchra.pdf), 2006.

- [14] ECLIPSE FOUNDATION. GMF Tutorial.  
[http://wiki.eclipse.org/index.php/GMF\\_Tutorial](http://wiki.eclipse.org/index.php/GMF_Tutorial), August 2008.
- [15] FLANDORFER, J. Web 2.0 metamodelbrowser, 2007.
- [16] GARRETT, J. J. Ajax: A new approach to web applications.  
<http://adaptivepath.com/ideas/essays/archives/000385.php>, February 2005.
- [17] GREENFIELD, J., SHORT, K., COOK, S., AND KENT, S. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.
- [18] GRONBACK, R. C. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.
- [19] HAREL, D., AND RUMPE, B. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Tech. rep., Jerusalem, Israel, Israel, 2000.
- [20] HECK, S. Model transformation for verification: Building the basis for a generic tool. <http://smv.unige.ch/student-projects/finished-projects/files/heck.pdf>, 2005.
- [21] HÄFELE, H., AND MAIER-HÄFELE, K. *101 e-learning-Seminarmethoden : Methoden und Strategien für die Online- und blended-learning-Seminarpraxis*. ManagerSeminare-Verl.-GmbH, Bonn, 2004.
- [22] HITZ, M., AND KAPPEL, G. *UML @ Work*. Dpunkt Verlag, September 2005.
- [23] HOWLETT, S. Microsoft’s Silverlight vs Adobe’s Flash: November 2008 Update. Tech. rep., imason inc., 2008.
- [24] JORDAN, D., AND EVDEMON, J. Web services business process execution language version 2.0. OASIS Standard, April 2007.
- [25] KROEGER, H., AND REISKY, A. *Blended Learning - Erfolgsfaktor Wissen*. Norbert Meder, Bertelsmann, 2004.
- [26] MACVITTIE, L. *XAML in a Nutshell*. O’Reilly, Sebastopol, 2006.
- [27] MONNET, X. E. Development of a 2d meta model browser for the mtv framework, October 2006.
- [28] NIEGEMANN, H. M. *Kompendium E-Learning*. Springer, Berlin, Heidelberg u.a., 2004.
- [29] NODA, T., AND HELWIG, S. Rich Internet Applications - Technical Comparison and Case Studies of AJAX, Flash, and Java based RIA. Tech. rep., UW E-Business Consortium, 2005.
- [30] OBJECT MANAGEMENT GROUP (OMG). Meta Object Facility (MOF) Specification, April 2002.
- [31] OBJECT MANAGEMENT GROUP (OMG). OMG XML Metadata Interchange (XMI) Specification Version 2.0.  
<http://www.omg.org/cgi-bin/doc?formal/2003-05-02>, May 2003.

- [32] OBJECT MANAGEMENT GROUP (OMG). OMG XML Metadata Diagram Interchange Information (XMI-DI) Specification Version 1.0. <http://www.omg.org/cgi-bin/doc?formal/06-04-04>, 2004.
- [33] OBJECT MANAGEMENT GROUP (OMG). *Business Process Modeling Notation (BPMN)*, Version 1.0, 2006.
- [34] OSTERMAYER, L. Evaluation and comparison of ajax frameworks regarding applicability, productivity and technical limitations. Master's thesis, Vienna Technical University, 2008.
- [35] PEMBERTON, S. XHTML 1.0: The Extensible HyperText Markup Language (Second Edition). World Wide Web Consortium, Recommendation REC-xhtml1-20020801, August 2002.
- [36] REISIG, W. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Scie. Springer-Verlag, Berlin, Germany, 1985.
- [37] RICHARDSON, L., AND RUBY, S. *RESTful Web Services*. O'Reilly, Sebastopol, CA, USA, 2007.
- [38] RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991.
- [39] SCHMIDT, D. C. Model-driven engineering. *IEEE Software* 39, 2 (February 2006).
- [40] SELLS, C., AND GRIFFITHS, I. *Programming Windows Presentation Foundation*. O'Reilly Media, Inc., 2005.
- [41] THUM, C., SCHWIND, M., AND SCHADER, M. Slim - a lightweight environment for synchronous collaborative modeling. In *MoDELS* (2009), vol. 5795 of *Lecture Notes in Computer Science*, Springer.
- [42] TSCHESCHNER, W. Oryx dokumentation, 2007.
- [43] ULLENBOOM, C. Java ist auch eine insel, 2006.
- [44] W3 SCHOOLS. Web statistics and trends. Tech. rep., 2009.
- [45] WESCHKALNIES, N. *Adobe Flash CS3: Das Praxisbuch zum Lernen und Nachschlagen*, 1 ed. Galileo Press, 2007.
- [46] WIDJAJA, S. *Rich Internet Applications mit Adobe Flex 3*, 1 ed. Hanser Fachbuch, 2008.