Hochschule Darmstadt
Fachbereich Media
Studiengang Information Science & Engineering
/ Informationswissenschaft

Masterarbeit

Thema:	User Interface Development based on XML Schema
Bearbeiter:	Maximilian Richter
Immatrikulationsnummer:	724764
Referent:	Prof. Dr. Bernhard Thull
Korreferent:	Prof. Dr. rer. nat. Reginald Ferber
Abgabe:	2. Juli 2012

Abstract

This master thesis describes the potentials of developing a user interface based on an existing XML Schema definition. The focus is on automatic methods for creating interface elements.

The theoretical and practical basics and referenced fields are described and their contribution to the solution approaches are explained. Two such prototypical implementations are demonstrated and compared: a) the Eclipse Modeling Framework (EMF) in combination with the Extended Editing Framework (EEF) as an Eclipse-based and thus Java-based software generation framework, and b) a browser-based prototype that builds on HTML5, CSS and JavaScript/CoffeeScript. It follows the Model-View-Controller paradigm, its generation concept is developed by the author and concretely implemented with the scripting language Python.

Zusammenfassung

In dieser Masterarbeit werden die Möglichkeiten beschrieben, wie auf Basis der existierenden XML-Schema-Definition für die Projektinformationssprache *PrIML* ein geeignetes User Interface erstellt werden kann. Der Fokus liegt dabei auf den Potentialen der automatischen Erzeugung von Oberflächen-Elementen.

Es werden die theoretischen und praktischen Grundlagen und Bezüge dargestellt und deren Beitrag zur Lösung der Aufgabenstellung deutlich gemacht. Zwei Ansätze für konkrete prototypische Umsetzungen werden in der Arbeit vorgestellt und verglichen: a) das Eclipse Modeling Framework (EMF) in Kombination mit dem Extended Editing Framework (EEF) als Eclipse- und damit Java-basierter Ansatz der Software-Generierung und b) eine Browser-basierte Lösung, die HTML5, CSS und JavaScript/CoffeeScript einsetzt, auf dem Model-View-Controller-Paradigma basiert und dessen Generatorschritte selbst konzeptioniert und mit der Skriptsprache Python umgesetzt sind.

Declaration

Erklärung

Declaration Concerning Library Services

Please choose:

O I agree that this thesis is loaned to library users.

O I do not agree that this thesis is loaned to library users. It contains confidential corporate information and is thus not accessible to the public.

Date Signature

Erklärung zur Ausleihe

Bitte ankreuzen:

O Mit der Ausleihe der gedruckten Abschlussarbeit bin ich einverstanden.

O Mit der Ausleihe der gedruckten Abschlussarbeit bin ich nicht einverstanden. Die Arbeit ist gesperrt, da sie in einem Betrieb durchgeführt wurde und ihr Inhalt ausdrücklich durch diesen gesperrt ist. (Vgl. ABPO § 18 (9))

Ort, Datum Unterschrift

List of Figures

List of Listings

List of Tables

Abbreviations

ABPO Allgemeine Bestimmungen für Prüfungsordnungen

AJAX Asynchronous JavaScript And XML

API Application Programming Interface

AUI Abstract User Interface

CD-ROM Compact Disc-Read Only Memory

cf. confer

CRUD Create, Retrieve, Update, Delete; Common tasks in data manipulation

CSS Cascading Style Sheets

CTD Complex Type Definition

CUI Concrete User Interface

CWM Common Warehouse Metamodel

DOCX XML-based document format by Microsoft Word

DOM Document Object Model

DRY "Don't Repeat Yourself"

DSDL Document Schema Definition Languages

DSL Domain-specific language

DTD Document Type Definition

ECMA European Computer Manufacturers Association

EEF Extended Editing Framework

e.g. exempli gratia (engl.: for example)

EMF Eclipse Modeling Framework

EMFT Eclipse Modeling Framework Technology

EMOF Essential Meta Object Facility

EPF Eclipse Process Framework

f./ff.	folio (engl.: on the next [page(s)])
FUI	Final User Interface
GUI	Graphical User Interface
HCI	Human Computer Interaction
HTML	Hyper Text Markup Language
ID	Identifier
IDE	Integrated Development Environment
ISCUE	Nuremburgian software firm; developers of *PrIML*
i.e.	id est (engl.: that is)
JAXB	Java XML Bidnings
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
LESS	[In fact not an abbreviation]
MacOS	Macintosh Operation System
MBSD	Model-based software development
MBUI	Model-based User Interface
MBUID	Model-based User Interface Development
MOF	Meta Object Facility
MVC	Model-View-Controller
OMG	Object Management Group
OOP	Object-Oriented Programming
p./pp.	page/pages
PDA	Personal Digital Assistant
PDF	Portable Document Format
PHP	PHP Homepage Pre-Processor
PIM	Platform-independent model
PrIML	Project Information Markup Language
PrIOF	Project Intermediate Output Format

PSM Platform-specific model

PyXB Python XML Bindings

Qt GUI Toolkit with bindings for many programming languages

RDBMS Relational Data Base Management System

RDF Resource Description Framework

SASS Syntactically Awesome Stylesheets

SGML Standard Generalized Markup Language

STD Simple Type Definition

UI User Interface

UIDL User Interface Description Language

UIMS User Interface Management System

UML Unified Modeling Language

UsiXML USer Interface eXtensible Markup Language

VM Virtual Machine

W3C World Wide Web Consortium

XInclude XML Include; standard for weaving together the content of XML files

XMI XML Metadata Interchange

XML eXtensible Markup Language

XPath XML Path; language for addressing nodes in XML structures

XQuery XML Query; language for programmatically querying XML collections

XSD XML Schema Definition

XSLT XML Stylesheet Language Transformation

Contents

1 Introduction

1.1 Goals

This thesis describes the potentials of developing user interfaces for data management based on eXtensible Markup Language (XML) Schema definitions. It is not the goal to only gather and present theoretical concepts and models from this research area. Taking the Project Information Markup Language (*PrIML*) XML vocabulary as the starting point the thesis does describe fields of research and practice relevant when using XML Schema as the basis for User Interface (UI) development. But based on these relevant fields possible solutions are presented that are generic enough to be at least conceptually reusable with other XML Schema vocabularies but also form a specific solution for the *PrIML* vocabulary. Applying the existing methods pragmatically to a specific use case and identifying the value that model-based paradigms have when practically used is the practical goal of the thesis.

Automating processes and partly generating software components is a key goal of this thesis' prototypical solutions. A discussion of the limitations of generating UIs out of schema definitions is also within its scope and is illustrated with the solution approaches described in chapter 6. Some common sense statements of Model-based User Interface Development (MBUID) research shall be compared to the results that a specific use case can provide.

Being able to give one or more recommendations (be it positive or negative ones) to ISCUE – the firm that developed *PrIML* – in order to support the UI development efforts is another key aspect of the research and the prototypical developments described in the following chapters. To have a better basis for such recommendation and for the reasoning about advantages and disadvantages of environments and frameworks, a comparison between two practical solution approaches seems appropriate.

1.2 Thesis Structure

Chapters' Order The order of the chapters logically reflects the considerations made: the input vocabulary is *PrIML* (chapter 2), for which a UI (chapter 3) is demanded. Automation in the desired software development steps is desired – this leads to *Model-based software development* in chapter 4 –, which raises the necessity of clarifying what models, (meta-)modelling and XML Schema, as a specific meta-model, are (chapter 5). Based on these descriptions the two prototypes can be presented and compared (chapter 6) and conclusions (chapter 7) can be drawn.

PrIML Chapter 2 is about the current state of the *PrIML* model and process framework. *PrIML* is a vocabulary and tool set for project information management. Its model and workflow are described in order to define the basis for the desired UI. Since this thesis uses *PrIML* as the precedence case in matters of applicabilty of platforms and frameworks, there are cases where concessions have to me made concerning the depth of research, comparison of different tools etc. A generic solution also applicable to other non-*PrIML* XML Schema definitions is desirable, but when necessary, trade-offs are made in favor of the applicability to *PrIML*. Examples used to illustrate the explained constructs and methods are often taken from the *PrIML* vocabulary.

Theoretical and Practical Reference Fields Chapters 3 through 5 describe the fields of research and practice that influence and form the overall topic and goal of the thesis. Since the focus is on the semi-automatic UI development based on a specific XML Schema vocabulary there can be no exhaustive evaluations and explanations of the referenced fields. The goal is to give an overview of the related concepts and focus on the ones directly applicable and appropriate for the solution in question. In cases where broader and/or deeper explanations are omitted due to the thesis focus, this is briefly commented as such.

UI Development Chapter 3 describes the field of (graphical) user interface development.[1] The focus is on explaining the common patterns of modern UIs and the possibilities and limitations when formalizing processes through model-based and model-driven approaches in UI development. Note that most aspects of usability are not deeply considered about in this thesis. Potentials and limits of formalizing UI components, systematically processing them and the general model-based approach are essential to the considerations taken throughout the complete thesis. These aspects all influence the usability but it is not the main focus to manually establish usability mechanisms when it comes to the prototype development. Customization approaches are included in order to be demonstrations of applicability rather than to be readily developed add-ons.

Model-based Software Development. Modelling Chapter 4 presents concepts and methods of model-based software development. The main rationale of these concepts in general and for this thesis specifically is to avoid duplication of existing domain information in two or even more places [KXF12]. This leads to the working hypothesis that it is possible and appropriate to reuse information from the XML Schema definition and thus partly automate the development of the user interface for *PrIML*. Such models, modelling and meta-modelling standards are described in chapter 5.

Prototypical Developments After describing the fields, tools and paradigms relevant to this development approach chapter 6 describes concrete solution approaches made in the thesis' context. They are the core of this thesis and illustrate concrete use cases of the theoretical methods. Using these hands-on solution approaches to reveal potentials and limits of MBUID is one goal. Comparing two different ways of UI application generation is the other goal, that is expected to increase learning effects concerning design principles of generator frameworks and processes.

[1] In most cases in this thesis the term *development* is preferred over *design*.

Conclusions and Lessons Learned Chapter 7 evaluates the results of the thesis and compares them with the initial goals. Recommendations in terms of framework appropriateness and possible limits of approaches are the pragmatic aspect of the conclusions that shall be drawn in this chapter. Technical and methodical lessons learned will also be part of this final chapter.

2 Project Information Markup Language (PrIML)[2]

2.1 Goals

PrIML is a project management model and tool. It was developed by software engineers in the software development company *ISCUE* [ISC11] primarily for internal usage. Its goal is to provide a structure for project information and the processes necessary to transform this information into human-readable documents for project reporting and traceability. The information about, for example, the project's requirements and the interdependencies thereof are the basis for a visual and table-based presentation in report documents. See listing 2.1 for an example system requirement from the *ISCUE* demo project. Figure 2.1 shows the extract of one possible resulting HyperText Markup Language (HTML) work report file.

2.2 Model

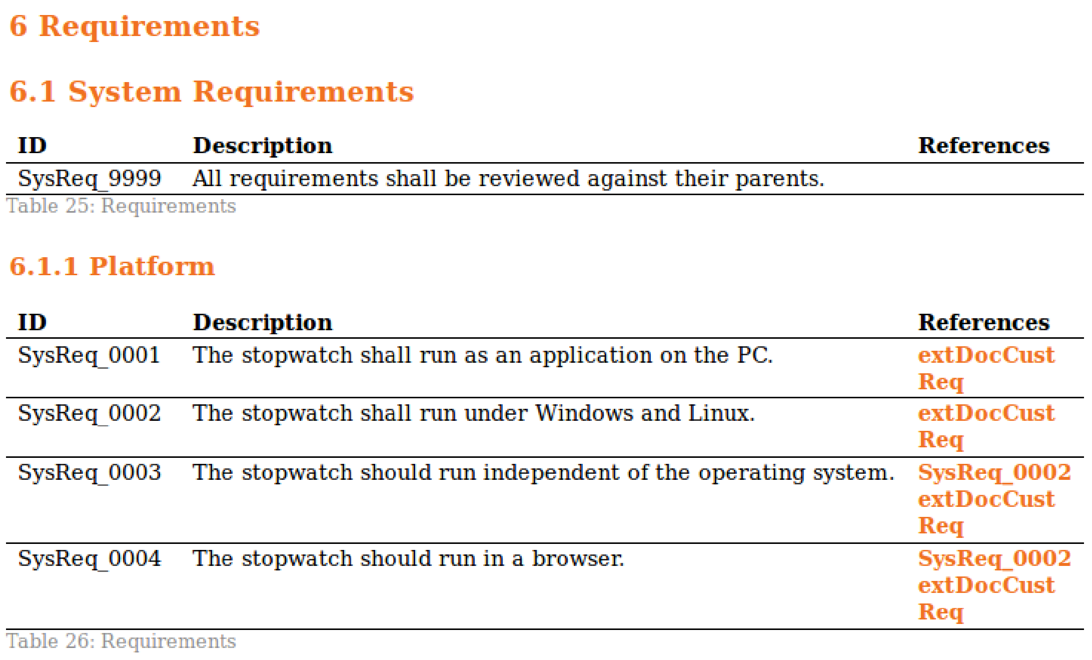
Conforming to the terminology that will be introduced in chapter 4.2, *PrIML* is a Domain-specific language (DSL) for the domain of project information management. It allows users to organize the information about projects and the related entities, i.e., to describe the specific project they are working on. Thus, a concrete project description is an instance of the *PrIML* model. This in turn is described using XML Schema as the meta-model and is divided into the sub-systems

- *Data*
  (all elements for project team, requirements, modification requests, etc.),
- *Filter*
  (constructs for allowing to filter entities by several aspects),
- *Format*
  (basic formatting constructs),
- *Report*
  (elements for defining report document templates),
- *Statemachine*
  (constructs for expressing state machine behavior),
- *Template*
  (for further report templating functionality) and
- *Use Case*
  for the definition of use case diagrams.

[2]For this section cf. [ISC12]

```
<requirement id="SysReq_0003">
  <history version="1" editor="oseidel"
    date="2008-10-23" change="created" />
  <description>
    The stopwatch should run independent of the
       operating system.
  </description>
  <val_criteria>
    The stopwatch runs in a browser.
  </val_criteria>
  <references>
    <refine>
      <external_document refId="extDocCustReq">
        <version version="1" state="done" />
      </external_document>
      <requirement refId="SysReq_0002">
        <version state="done" version="1" />
      </requirement>
    </refine>
  </references>
</requirement>
```

Listing 2.1: Requirement XML extract



6 Requirements

6.1 System Requirements

ID	Description	References
SysReq_9999	All requirements shall be reviewed against their parents.	

Table 25: Requirements

6.1.1 Platform

ID	Description	References
SysReq_0001	The stopwatch shall run as an application on the PC.	extDocCust Req
SysReq_0002	The stopwatch shall run under Windows and Linux.	extDocCust Req
SysReq_0003	The stopwatch should run independent of the operating system.	SysReq_0002 extDocCust Req
SysReq_0004	The stopwatch should run in a browser.	SysReq_0002 extDocCust Req

Table 26: Requirements

Figure 2.1: Requirement extract from a work report trace HTML file

In addition to these there is the declaration of an intermediate XML format called Project Intermediate Output Format (*PrIOF*) that is used in the transformation processes described below. See the thesis CD-ROM for detailed visualization images of the *PrIML* sub-system XML Schema Definitions (XSDs).

2.3 Design Decisions

Creating *PrIML* was in the first step a conceptual process. There existed ideas and concepts of entity types, their possible relations and the general request for model extensibility when needed.

An essential design decision was not to use a Relational Data Base Management System (RDBMS). Efficient versioning and diff[3] mechanisms, back-up functionality and collaboration were key goals of *PrIML* and RDBMSs could not meet these goals sufficiently.

One approach considered in the early development phase was making *PrIML* a Doxygen [Dox12] add-on and extending the commment structure that this tool provides. The lack of a reliable syntax basis and the intermixing of project information with source code discouraged this approach.

In the next steps the following main design decisions were made:

- use the XML standards family as highly standardized and well-supported in terms of tools and adaption in practice,
- use XML as the desired serialization format for project description instances,
- use XML Schema as the meta-modelling language for convenient element, type, attribute and group declaration constructs and
- use XML Stylesheet Language Transformation (XSLT) as the language for describing the transformation (and information enhancement) processes.

Originally *PrIML*-conforming project descriptions had to be stored in one XML file and were processed by one XSLT script directly in the web browser. Thus HTML was in that state the only output format directy available and due to browser rendering XSLT 1.0 was the transformation standard used.[4] With the need for more complex transformations that demanded for temporary result tree construction introduced by XSLT 2.0 and the request for more output formats, a re-design proved to be necessary.

The necessity for an intermediate data format between *PrIML* instance documents and the generated report document files arose and characterized the re-design. Thanks to this intermediate format, introducing new features in the *PrIML* model definition only affects one transformation step, i.e., the one that transforms *PrIML* instances into *PrIOF*. The further transformations are not affected by schema changes, they are "facaded" by the intermediate format. Additional output formats can easily be added to this intermediate format as well.

The XML Schema definitions are separated according to the subsystems mentioned above and stored in respective .xsd files. This modularization reflects the complemen-

[3]The name of the Unix tool *diff* [Chr93] is often used to generally describe the step of tracking data changes.

[4]Web browsers do not fully (or not at all) support XSLT 2.0 transformations.

tarity of the different vocabulary aspects, the parts are interwoven via `xsd:import` directives as needed. Modularity and the ability for recombination and independent editing by potentially more than one developer is also desired on the instance level. Therefore the element definitions in the XML Schema files are constructed as to enable the user to choose many different *PrIML* elements as potential document root elements. This allows for nearly arbitrarily granular modularization. Bringing together these separated instance document fragments is realized via `XInclude` statements. Including via the `XInclude` standard also demands that an `xml:base` attribute is allowed to occur on the level of the to-be-included element.

2.4 Workflow

The current *PrIML* workflow grounds on the XML Schema model files and XSLT processes. It uses an XML processor for performing the tasks of validation and transformation. Eclipse is used as the platform bundling all components. It includes the basic XML perspective,[5] the XML processor *Saxon B* [Kay09] and *Ant* [Apa12] build support.

2.4.1 XML Authoring

XML authoring, i.e., the data entry of XML structures, is accomplished through the XML perspective of Eclipse. It provides users with basically two different views. The design view uses a tree structure for presenting and editing the XML. The source view lets users directly edit the XML source code. Both views provide guidance mechanisms such as context-sensitive suggestions for sub-element and attribute creation or the presentation of documentation text fragments retrieved from the referenced XML Schema file.

The model requires IDs for entities of some types. These IDs have to be unique throughout the complete project description and have to match a defined pattern. IDs have to be manually specified and maintained correctly when it comes to referencing them from other entities. XML Schema's `xsd:unique` construct is used. The validation process uses it to raise errors in case the uniqueness of IDs is not obeyed.

2.4.2 Transformation Processes

Based on the aforementioned project descriptions in XML files there are XSLT [W3C07] processes transforming the data via *PrIOF* into structurally pre-defined report documents. The processes are bundled into *Ant* build files and can be executed from within the Eclipse Integrated Development Environment (IDE). Directory clean-ups and eventual sub-directory creations accompany the core XML validation and XSLT transformation steps.

An important step before any of the output is generated is the building of a temporary result tree. It contains information about relationships between entities in the processed instance document(s). This information has to be resolved in a complex

[5]Eclipse uses the term *perspective* for pre-configured user interface subsets that fit the respective context, in this case XML authoring and editing.

manner because it is not directly accessible in all possible directions out of the XML instance document(s). Internally this tree is called *BiDirTraceTree*, indicating the bidirectionality. This means that even if a reference between two entities is only asserted from one of the involved entities, it is made accessible from both directions via the *BiDirTraceTree*.

Possible result formats of the transformation processes are currently HTML, DOCX and the Portable Document Format (PDF). These are viewed by users with the respective viewer tools such as a web browser, office suite and PDF viewer. The *PrIML* subsystem for reports provides the language constructs for defining report "skeletons" for respective report documents. Filter mechanisms are defined in *PrIML*'s filter XSD file for selecting specific instances matching (potentially arbitrarily deep nested) filter rules. The report documents are populated with the concrete data by the XSLT processes.

2.5 Relation to Thesis Scope

The basic design decisions as well as the current state of *PrIML*'s model and workflow have been described. They define the tool's state and the basis for this thesis' considerations for UI solutions. Building upon the premises, existing model definitions and processes, the *PrIML* framework uses, the relevant fields of research and practice can be described.

The focus of these following chapters mainly follows the applicability for the *PrIML* use case.

3 User Interface Development

3.1 Overview

The field of UI devlopment belongs to the more general research field of Human Computer Interaction (HCI). The latter subsumes all aspects of human users interacting with computers and vice versa. Generally put, UIs are the “[...] directly experienced aspects of a thing or device” ([Tra09], p. 9) and for most users “[...] the distinction between interface and system is [...] meaningless” ([Tra09], p. 1). De facto there is a difference between the UI as the front end of an application and the back-end processes carrying out more complex computations and doing data persistence.

Over the last few years there have been developments towards common features of UIs (cf. [Mye00], p. 2). The tendency to provide users with a graphical user interface (GUI) instead of a textual (command line) interface is such a common feature on a particularly low level. The fact that keyboard and mouse are widely accepted as the translating units between physical interaction and computable signals is another basic commonality. Building on this graphical fundament and the peripheral units for interaction there have been developed different sorts of features and interaction patterns, mostly referred to as *widgets*. These are the components that GUIs are constructed of. Frequently used examples are group boxes, buttons, drop-down lists, navigation menus and text input fields.

Research in UI development distinguishes between different interaction patterns such as *direct manipulation*, *menu selection*, *form fill-in*, *command language* and *natural language* (cf. [Shn10], pp. 84ff.). They differ in terms of the level of formalization and specificity respectively. Enabling users to directly manipulate items in a UI usually allows a more natural way of interacting and editing. Generally, using appropriate metaphors (cf. [App11]) in UIs is crucial since it reflects the tasks to be performed in an intuitive manner. An example used throughout operating systems such as Windows and MacOS is the metaphor of a desktop as the basic interaction space for the user [Wik12a]. This is often complemented by adapting the task of putting an object into the trash can to the task of deleting a file, object or folder (cf. [Shn10], p. 192). It is an example of bringing concepts into a UI that are not 1:1 (in a physical way) applicable but intuitively acceptable by users.

Within certain contexts such as operating systems, windowing systems or corporations there often exist guide lines for UI design patterns. An example is [App11] for the MacOS operating system, similar standards exist for Android-based applications [Goo12] and other contexts. Creating elements of recognition and thus lowering the acceptance and understanding threshold are the main goals of such guide lines. The “look and feel” of a system is in most cases desired to be perceived consistently. A UI that behaves and/or looks different from the user’s expectation can easily lead to confusion or even rejection (cf. [Shn10], pp. 23 and 88).

3.2 Model-Based User Interface Development

For several years there have been surveys concerning the costs and efforts in software development. Especially the role of UI development as a sub-category of software development is a prominent one. UIs are the front end of a software system and they reflect the functionality and the way of interacting with it. A great amount of time and effort of software projects is put into UI development (cf. [Mye92], p. 195[6]) and for many years the necessity to make UI development more efficient has been focused. The goals and important concepts of these approaches are described below. In general, model-based UI development aims at formally describing user interfaces and introducing abstraction layers between the domain, the user's tasks and the UI itself – mostly with abstract, concrete and final aspects ([Sch96], pp. 7ff.; cf. [Cal03], p. [5]).

Just recently the World Wide Web Consortium (W3C) founded the *Model-Based User Interfaces (MBUI) Working Group* that in its mission statement formulates the plan to "[...] develop standards as a basis for interoperability across authoring tools for context aware user interfaces for Web-based interactive applications." [W3C11a] The working group's "initial focus is on task models, and UI components and integrity constraints at a level of abstraction independent of the choice of device" [W3C11a] and will later include the more concrete UI levels (cf. [W3C11a]).

3.2.1 Goals

The main MBUID goals reusability, flexibility and platform-independence (cf. [Leh05], p. 9) and multi-modality are described in this chapter.

Reusability Reusing components and re-occurring patterns is crucial in MBUID. The common aspects of UIs can be abstracted out and do not have to be newly implemented in every UI. The fact that data structures are mapped to UI widgets in often similar ways can influence MBUID systems. Such mappings leverage reusability.

Flexibility When applying MBUID patterns it is possible to remain flexible in terms of widget representations and layout decisions (cf. [Pin03], p. 62). Design decisions are out-sourced and declared centrally. Changing it causes direct reflection in all applicable contexts. Refactoring of the UI can be realized without expensive re-development, this increases flexibility.

Multi-Platform/-Device Although the importance of software development for different types of devices such as Personal Digital Assistant (PDAs) and cell phones has been stressed for several years, it has only reached broad attention and adoption through main-stream product families such as smart phones and tablet computers in the last few years.

Apart from device-specific implementation differences there has always been the multi-platform issue, i.e., the development for different operating systems. In a broader sense the term *multi-platform development* can also include implementations based on

[6]Although this survey dates back to 1992 it has been frequently cited since then, cf. [Mei11b], p. 2

different IDE frameworks and/or different web browsers when it comes to web-based applications.

The development for different platforms and/or devices can be simplified with MBUID methods. Using a UI model for platform-specific UI generation processes can avoid duplicating implementation efforts. General platform-independent models (PIMs) are transformed into platform-specific models (PSMs) by adding specific aspects of the respective target platform.

Multi-Modality Declaring UI elements in different levels of abstraction in models makes the development of multi-modal interfaces easier. One UI model (or more than one with each one representing a different aspect of it) can be the basis for different manifestations. Gesture-based and voice-operated are important examples of interaction techniques that can be propelled by model-based development methods. Multi-modality often co-occurs with device specifica; smart phones and tablets for example have accelerometers that form the basis for motion interaction. Desktop computers do not provide such interaction methods, the same application may however be available there as well. One task that a user can accomplish with the application has to be manifested in different ways, dependent on the modalities the respective device offers.

3.2.2 Definitions

There are concepts and model distinctions in MBUID that are frequently used and implemented. These are defined in this chapter. It is worth noting that not every possible model type is demanded and/or applicable in every framework and MBUID context (cf. [Sze96], p. xxiv; cf. [Sch96], p. 7).

The early MBUID approaches rather propagated the concept of one integrated model whereas the further developed recent approaches divide the different aspects of the UI into independent models complementing each other (cf. [Sch96], p. 18).

Domain Model A domain model describes the concepts of reality that are relevant to the UI. The entity types, attributes and references in the domain are formally described. The *PrIML* vocabulary is an example of a domain model as it models the project information management domain.

Task Model A task model contains descriptions of tasks that users can perform with the UI. It formalizes the results of task analysis (cf. [Tra09], p. 41) and represents the "[...] flow of information between the models when carrying out the user's tasks" ([Gri01], p. [3]).

User model / User profile Characteristics that users of a UI share can be formally expressed in a user model. Abilities and possible limitations are examples of expressable features. Preferences concerning the availability of options in the UI are another potential aspect reflected in a user model (cf. [Sch96], pp. 10f.).

Abstract User Interface The Abstract User Interface (AUI) describes the UI in terms of functionality but not taking concrete widget representations into account. See chapter 3.2.3 for an example.

Concrete User Interface The Concrete User Interface (CUI) describes the UI in a concrete manner, defining not only abstract functionality but specific widget types used for presentation as well.

Final User Interface Manifestating a CUI in a specific language, framework or infrastructure is formally called the final user interface (FUI). It builds potentially on all aforementioned UI model types and realizes their aspects in a functional UI.

3.2.3 Standards and Paradigms

User Interface Description Languages (UIDLs) The concept of UIDLs is to describe UIs declaratively. Often there exists more than one model for a UI since different levels of abstraction are separated from each other as described above. Some modern UIDLs are based upon the Cameleon Reference Framework ([Cal03]) which defines four basic levels to be represented (cf. [W3C12a]):

- Concept model and Task model,
- Abstract User Interface (AUI),
- Concrete User Interface (CUI) and
- Final User Interface (FUI).

Formally, UIDLs are DSLs for the domain of UIs. They provide constructs that are commonly needed when describing UI components, layout and sometimes behavior. Many UIDLs – as DSLs in general – use XML as their basis.

UI declarations need to be processed in any way to be functional. The formal description is rendered into the corresponding components on the screen, transformed into a voice recognition system or any other possible sort of interface. This rendering can either be performed at build-time, i.e., statically, or at run-time, i.e., dynamically.

One UIDL that modularizes its model components instead of bundling them up in one model is the USer Interface eXtensible Markup Language (UsiXML) [Usi07a]. Abstract UI, concrete UI, domain and tasks are some of the describable model aspects. Figure 3.1 depicts the main model types in UsiXML.

GUI Toolkits and Libraries In realizing the reusability aspect in UI development GUI libraries and toolkits form a common basis. They provide sets of widgets that can be included into and used by applications (cf. [Mye95], p. 14).

GUI toolkits can either be dependent on a certain platform or programming environment like for example Java Swing [Jav10] or platform-independent like Qt [Nok12], for which several language libraries exist.

In a broader sense web-based libraries and frameworks like jQuery UI [jQu12c] and Twitter's Bootstrap [Twi11] are GUI toolkits as well. They themselves depend on the widgets the HTML standard [W3C12c] provides but they extend them in many cases.

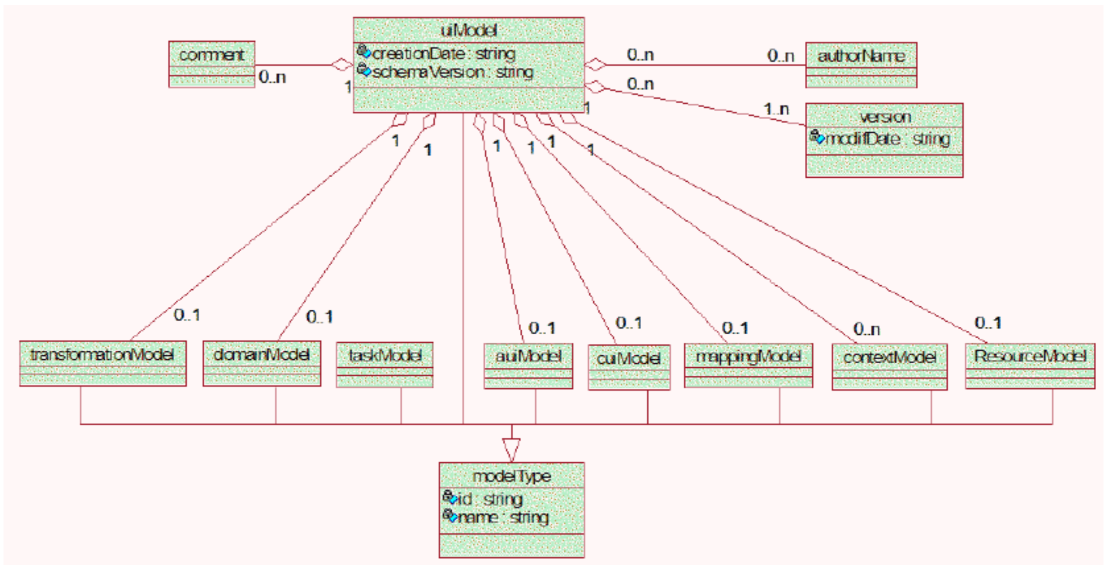


Figure 3.1: UsiXML model components; [Usi07b]

User Interface Paradigms Since UIs are building upon common characteristics these can be listed here with focus on applicability to this thesis' goal. Most UIs are based on pre-defined control components: widgets. They allow for interaction with a pointing device such as a mouse, keyboard input or by touch events. The most famous widgets are click buttons, text input fields and select boxes. Variations for different usages and amounts of data are usual. An example of a comparable functionality but different presentation layout is letting users choose one or more elements from a list of possible entries. For less than five elements a radio button group might be considered an appropriate widget, for many elements a scrollable select box might be a more desired solution. The abstract interaction object in this scenario is the same ("choose one element out of n"), the concrete interaction objects are different (radio-button group and multi-select box) (cf. [Gri01], p. [6]).

From a development perspective the Model-View-Controller (MVC) pattern is important and widely adopted. It was first used in the Smalltalk-80 implementation [Ree78] and introduces the three name-giving components model, view and controller. Models represent the data entities with attributes and basic getters and setters. Views define the presentation layer displayed on screen. Controllers act as the intermediate components between models and views. They listen to events (e.g., interactions in the UI), invoke methods on models and update views in turn. This separation of data, presentation and the intermediate logic allows for flexible combination by forcing decoupling of independent layers of a UI application. Many programming languages, GUI frameworks and application toolkits implement or adapt the MVC pattern – sometimes with slightly different nomenclature and focus (cf. [Gos05]; cf. [Osm12]).

3.2.4 Limitations

The formalization approach inherent to MBUID concepts is based on the advantages it provides in terms of efficiency and easier maintainability. Experiences made over the last 20 years[7] show that there are some aspects limiting the usage of MBUID techniques and mechanisms. Automation always demands for formalization and standardization (cf. [Kla06], p. 2). This implies that the generated UIs basically follow a similar sort of structure. Any "look and feel" features that are highly specific to the tasks and the domain the UI is acting upon have to be manually specified and coded on top of a generated basis ([Gri01], p. [2]). UIs that really support the user's task accomplishments and allow for efficient and context-aware usability experience need to be highly specific. To describe such specific behavior and interaction patterns in a formal model leads to contradiction in most cases. Formalization and specialization are opposites to each other and concrete interaction metaphors are hardly expressible using formal schematization.

Exploiting existing schema and model information for automating UI development will mostly lead to a form-based solution ([Sch96], p. 18) of the UI in question. The principle of direct manipulation is difficult to be supported by an automated UI approach. The prototypes in this thesis are highly form-based, which is acceptable for a basic functionality. Potentials for more elaborate UI techniques and paradigms exceeding forms as the interaction basis will be pointed out in place.

It is often stated that MBUID has not reached a main-stream level of acceptance in the UI development and design sector ([Tra09], p. 12; [Mei11a], p. 404; [Sze96], p. xxii; cf. [Mye00], p. 10). Reasons for the lack of adaptations are the often verbose syntax of UIDLs, the lack of out-of-the-box solutions (cf. [Mei11a], p. 404) and the contra-intuitive paradigm of formalizing UI development processes that much (cf. [Tra02], pp. 12ff.).

3.3 Relation to Thesis Scope

The development of UIs as a common way of human-computer interaction is crucial to any kind of application. Considerations about general paradigms such as the usage of metaphors in UIs and more specific decisions on which UI widgets to choose are essential when designing and implementing a UI for an application. The level of possible formalization and the applicability of systematically generating parts of the UI influences a) the (manual) implementation expense and b) the user experience provided by the UI.

Applying these concepts to the thesis goal of possible UI prototype solutions for *PrIML*, the focus is on the balance between formalization, i.e., basically treating all of the model's entity types in a similar way, and specialization, i.e., finding appropriate metaphors and derived interaction patterns for each entity type. Since the model-driven paradigm is key to this thesis the basic attempt is to automate similar steps in the UI development and to identify exemplary cases where further adjustment can improve user experience.

[7]The earliest approaches in MBUID date back to the early 1990s.

4 Model-Based Software Development

The terms model-*based* software development, model-*driven* software development and *Generative Programming* are closely related but are not considered synonyms. [Kla06] states the difference between *model-based* and *model-driven* by defining that model-based methods are not necessarily aiming for automation while model-driven approaches take models as the basis for transformations, i.e., automation processes (cf. pp. 3f.). The term *Generative Programming* is mainly used by [Cza04] and describes the method of developing software with automation approaches (cf. [Kla06], p. 2). It focuses on software system families rather than on single pieces of software. Usually models are used as the input of generation processes ([Kla06], p. 13) and it aims at a complete automation (cf. [Sta07], p. 37).

Generating software (components) is relevant to this thesis because using an XML Schema model as the basis of a UI follows the approach of automatically generating source code. The model is the input, the UI is the generated output. There can and shall be no 100% automation in this thesis' prototypes. Generators are used for repetitive steps, manual additions and adjustments complement this generated source code.

Model-based software development can be considered to be a super-set to MBUID, since every UI is a piece of software, which is, when developed in a model-based fashion, an example of model-based software development.

4.1 Goals

Possible advantages of generating software in general are stated by [Kla06] as (cf. pp. 50f.):

- consistent ID generation,
- performance,
- type safety and
- platform independence.

Furthermore, the following goals and principles are essential to generating software components.

4.1.1 Efficiency

Generative approaches are considered to increase efficiency because potentially one generator program is able to generate arbitrarily many software programs. Assuming that the generator in question is set up appropriately it performs programming tasks in a systematic, complete and repeatable manner (cf. [Kla06], p. 141).

4.1.2 Consistency

Consistency is a goal that generative approaches can support by performing tasks such as ID generation, applying naming conventions[8] and ensuring the validity of references between different pieces of software.

Generally generating source code can assist programmers by completing systematic and thus automatable tasks. Generators, once set up properly, ensure consistency throughout the components they generate (cf. [Kla06], pp. 2f.).

4.1.3 DRY (Don't Repeat Yourself)

Rather a central principle but indirectly also a goal is the DRY principle ([Hun00], pp. 27f.). It demands to assert data and information only once and to derive further views from that assertion in other contexts if necessary. The DRY principle is one of the crucial aspects that motivate generative programming and model-driven software development. Keeping information in formal representations and generating software based on these models realizes the DRY principle. It applies its deduplication goal for higher efficiency especially when information changes and evolutions occur. Consistency and propagation of information change is ensured when obeying the DRY principle.

Abstraction is the key requirement for DRY. When centrally asserting information in order to reuse it there has to be a shared definition, i.e., abstraction thereof. Examples of such shared definitions are configuration files, build files and models in general.

An example of a common use case of DRY is using one domain model for several components of an application. The data base tables can be generated from it, the objects in the application logic querying the data base, the UI components used for display and manipulation and the documentation (cf. [Hun00], pp. 28f.).

4.2 Methods

The methods that are described here are common in software generation although not all of them are applicable in every Model-based software development (MBSD) use case.

Code Generators Based on models or, more generally put, formal descriptions, code generators create source code (cf. [Sta07], p. 12). The formal description takes the role of a configuration for the generation process. Generation happens before the software is executed, i.e., at compile time. Interpreters have a similar goal as code generators: they create output based on formal descriptions, but at run-time. Since interpreters are not crucial to this thesis they are not described in depth.

Reference Implementation When generating software there exists the development paradigm of setting up a reference implementation as the target of the generation steps (cf. [Kla06], pp. 39ff.). The generator's output is always compared to the reference implementation and it is contiuously developed "against" it. The generator's status

[8] An often named example is a consistent nomenclature for get and set methods in Object-Oriented Programming (OOP).

is thus transparent in all development phases and the generator's scope is precisely comprehensible.

Domain-Specific Languages In software development abstraction is a key component for distributing complexity. DSLs are often used to enable domain experts to assert information about the domain in a more comfortable way without having to know many technical details apart from domain-immanent aspects. XML is often used as the basis for DSLs. It provides a stable syntax and a rich tool support for aspects like structuring (XML Schema, Document Type Definition (DTDs)), querying (XPath, XQuery) and transformation (XSLT).

Model-to-Model Transformations Since in most cases step-wise generation is necessary there are different generators contributing to the end result (cf. [Kla06], cf. p. 6). These different generators often need models of different abstraction levels (cf. [Sta07], cf. p. 195). This makes model-to-model transformation a method frequently used by MBSD.

An example of a model-to-model transformation is the XML Schema to Ecore transformation in the Eclipse Modeling Framework (EMF). The input XSD(s) are iterated and the first result is a representaton of the modelled entities expressed with the Ecore meta-model. EMF reuses constructs of the XML Schema meta-model, expresses them with constructs of the Ecore meta-model which contains information not directly available in XML Schema. This added information can be further used by source code generations based on the Ecore model and would not directly have been present from the model expressed in XSD ([Ste09], pp. 179f.).

Templating When generating source code (or potentially any other kind of text-based content) templates are a common way of pre-defining a structure that can be parametrized with concrete values. Static content is notated as is, complemented with expressions containing parameters and/or more complex expressions to be evaluated and filled in during generation (cf. [Sta07], pp. 146f.). Conditional output, repeated output and the invocation of other templates in order to delegate specific processing belongs to common template functionality.

Examples of software generation systems using templating are Acceleo [Ecl12a] and Xpand/ Xtend [Ecl12c]. XSLT as the main XML processing tool also uses templates that can either be named and called explicitly and/or that can have a matching clause containing XPath expressions identifying objects the template shall be applied to. In the web development context templating mechanisms are frequently used to render HTML fragments parametrized by (mostly) JavaScript Object Notation (JSON) [Cro02] data and inject these produced fragments into the Document Object Model (DOM) [W3C05] tree.

Combination of Generated and Manually Written Source Code Generations are rarely used stand-alone but complemented by manually written source code. There exist different methods of bringing both sorts of code together (cf. [Sta07] pp. 159f.).

An OOP-oriented paradigm is the three-level inheritance (cf. [Sta07], p. 161). The generator creates two levels automatically, the third level is manually provided by the developer. Methods and attributes are inherited and potential overrides and complements are applied. The generated code does not have to be manipulated by developers to fit specific needs.

Another paradigm which is widely used, for example in EMF (see chapter 5.2.2), are protected source code zones. These are marked with e.g. special comments and will not be overwritten in new generation runs. Since EMF for example is Java-based, it exploits JavaDoc comments and takes `@generated` tags into account. Whenever no such tag is present or is changed to `@generated NOT` the respective source code block is protected. This protection practice is criticized for intermixing source code irreversibly and complicating maintenance (cf. [Sta07], p. 160).

4.3 Relation to Thesis Scope

Closely related to the MBUID paradigm described above, the more general approach of MBSD plays a key role in this thesis scope. Generating software components, namely UI components, from formal descriptions is the goal of the thesis. Evaluating different ways of applying such methods on XSD file collections and demonstrating possibilities for a potentially generic and adaptable approach is the connection between the concepts described in this section above and the thesis as a whole.

Generation steps are the connection between models (be it a domain model or a UI model as a special case) and source code. Many MBUID systems lack normative generation/ transformation processes and leave this to specific implementations. This is one limiting aspect of MBUID adaption in UI development (cf. chapter 3.2.4).

5 Modelling

Modelling[9] is a broad field and referenced in many research and practice contexts. It is based on the concept of a model which in most cases is defined as a description of a simplified part of reality (cf. [Kla06], pp. 12f.; cf. [Del07], pp. 10f.).

MBSD and MBUID approaches as described above require models as their configuration input; that makes modelling in general and XML Schema modelling specifically (since it is used for the *PrIML* vocabulary definition) an essential part of this thesis' considerations.

5.1 Languages and Standards

Since describing models is an abstract process there have to be methods and tools for making the results of modelling explicit. Declaratively asserting information about models demands for languages enabling modellers to express the main types of modelling properties (also see chapter 5.2 and cf. [Ste09], p. 17; cf. [OMG11a], p. 8):

- Classes,
- Attributes,
- References,
- Cardinalities and
- Constraints.

Visualizing models is often used for presentation and human reception.

5.1.1 Unified Modeling Language

In order to increase standardization in modelling and to explicitly meta-model (cf. chapter 5.2) the components commonly reused in modelling processes the Object Management Group (OMG) developed the Unified Modeling Language (UML) standard (cf. [OMG11b]). It received broad adaptation for modelling issues in various domains such as software development.

UML is not within the focus of this thesis, which is the reason for not explaining it further.

5.1.2 XML-Related Standards

Especially for modelling XML-based languages (also called XML dialects) there exist standards reflecting the specific XML features. Such features include the distinction between elements and attributes, usage of namespaces and XML entities. The two

[9]This thesis uses the spelling *modelling*, not *modeling*; exceptions are direct citations and product names.

basic approaches of declaring XML dialect structures are schema languages that are themselves expressed with XML (XML Schema, RELAX NG and Schematron) and non-XML-based languages (DTDs).

Attempts to convert XML declaration standard descriptions are made by tools such as Trang [Cla08]. It takes an input definition (in one of the formats RELAX NG (XML syntax), RELAX NG (compact syntax), DTD or XML instance file) and transforms it into the desired output schema language. Possible outputs are both RELAX NG syntax standards, DTD and XML Schema.

DTD XML is a subset of the more generic Standard Generalized Markup Language (SGML) [W3C95] standard for defining markup languages (cf. [W3C08]). To declare an SGML-conform markup language there exists the DTD standard. It enables modellers to define a grammar asserting the permitted structure of instance documents conforming to the DTD (cf. [W3C08]). The components of a DTD are element definitions and its cardinalitites, attribute definitions and XML entity definitions. DTDs themselves are not XML-based but introduce a grammar syntax of their own. The reusability of components is not supported, which can lead to repetitive assertions when designing DTDs. Furthermore, DTDs are namespace-unaware (cf. [Wik12b]), which limits their potential use cases when vocabulary combination is needed.

XML Schema[10] XML Schema is a language for declaring the structure of XML document sets. XML Schema can express any construct also expressable with DTDs, but the reusability of declared structures in more than one context and the support for different modelling patterns and paradigms[11] makes the use of XML Schema in many cases more comfortable than DTDs. Namespace-awareness and the fact that it is itself XML-based are further advantages. The XML Schema component diagram provided by the W3C is depicted in figure 5.1.

Mainly, XML Schema is for declaring data types that instance documents and their elements can use. XML Schema distinguishes between simple and complex data types. Simple types are what in other contexts is referred to as primitive types (strings, booleans, integers etc.). There exists a definition of primitive types, the XML Schema Datatypes [W3C04d]. These are a sort of canonical type vocabulary that is widely used, even in other contexts.[12] Complex types, as XML Schema defines them, allow elements to contain further tree structure and/or attributes. Allowed group structures in complex type definitions are

- sequence: order-sensitive group of sub-elements,
- choice: one of the listed sub-constructs is allowed and
- all: all of the listed sub-elements have to occur, irrelevant in which order.

These can be nested in complex manner to allow for elaborate representation of data structures. Furthermore, group constructs can themselves contain occurrence con-

[10]for this section cf. [W3C04b], [W3C04c] and [W3C04d]

[11]see [Cos12] for considerations about when to globally/ locally declare components of XML Schemas

[12]The Resource Description Framework (RDF) data model [W3C04a] for example reuses the XML Schema data types and is a standard that can, but does not have to, rely on the XML standards.

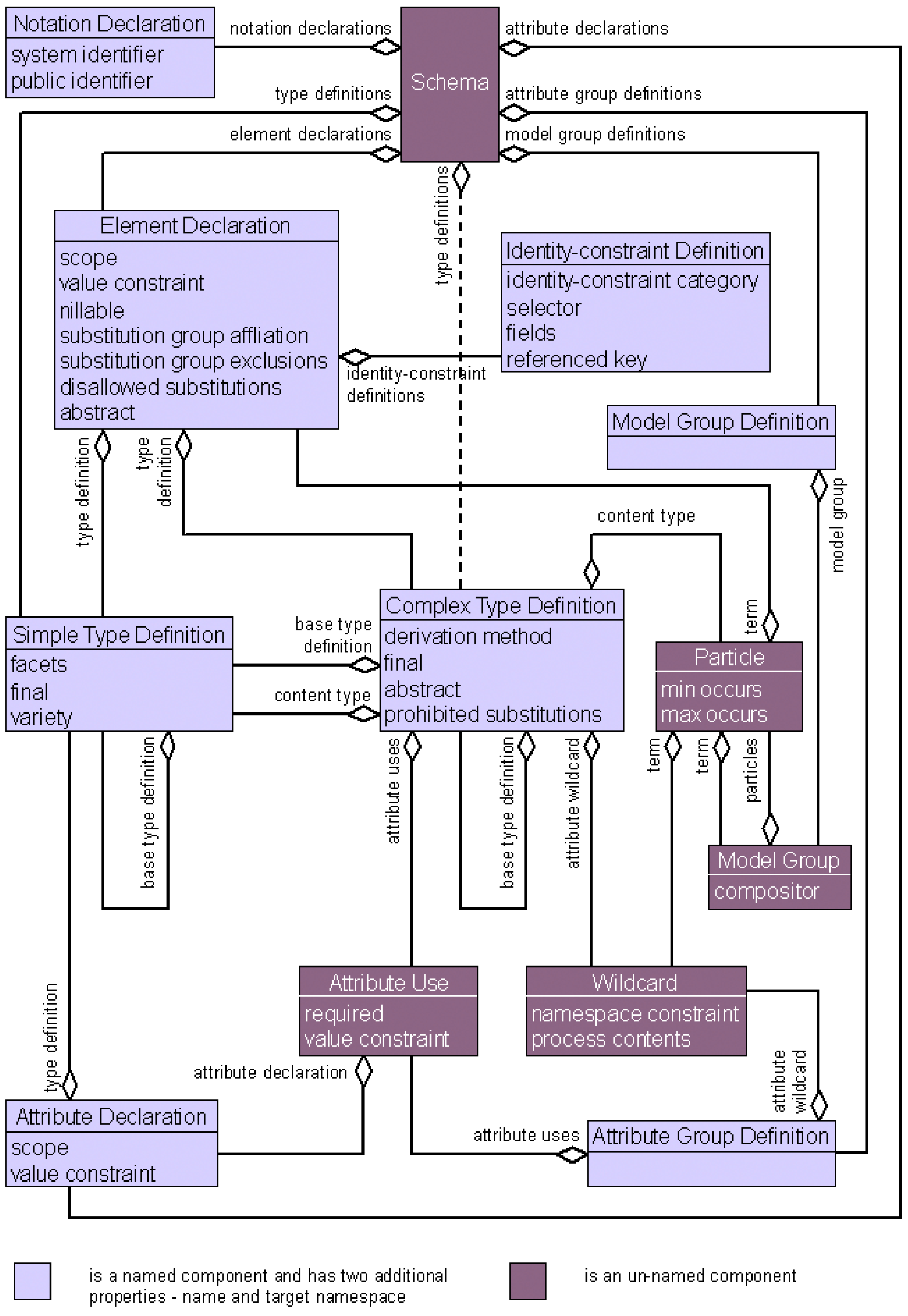


Figure 5.1: XML Schema components; [W3C04c]

```
<xsd:element name="requirement">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="priml_element_basetype">
        <xsd:sequence maxOccurs="1">
          <xsd:element name="statemachine"
            type="chartRefType" minOccurs="0"
            maxOccurs="1" />
          <xsd:element name="usecase_diagram"
            type="chartRefType" minOccurs="0"
            maxOccurs="1" />
          <xsd:element ref="format:image" minOccurs="0"
            maxOccurs="unbounded" />
          <xsd:element ref="val_criteria" minOccurs="0"
            maxOccurs="1" />
          <xsd:element name="references"
            type="references_requirement" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:unique name="Unique_References_Of_Requirement">
    <xsd:selector xpath="data:references/*/*" />
    <xsd:field xpath="@refId" />
  </xsd:unique>
</xsd:element>
```

Listing 5.1: Requirement XML Schema fragment

straints, see below. Any of the constructs *sequence*, *choice* and *all* can be aggregated in named group constructs in order to be referenceable.

Both simple and complex type constructs allow for extending and restricting existing type declarations. Base types for extension and restriction can be XML Schema datatypes or user-specified ones.

Elements defined in an XML Schema are characterized by the content model they are typed with. The content model contains the data type (simple or complex) and the attributes they may carry. Attributes are always of a simple type, i.e., there is no further structure possible within attribute values. However, there exists the possibility to type attributes with list types. Thus, a space-separated list of type-conforming values can be contained in one attribute value and will be interpreted as such.

Attributes can be grouped in *attributeGroups* in order to be referenced together from other element definitions.

Cardinalities in XML Schema can be defined more fine-grained than in DTDs. Exact numerical values can be asserted as lower and upper bounds,[13] complemented by the value *unbounded* applicable to the upper interval bound value. Declarations of attribute use are the same as in the DTD standard: possible values are *required*, *optional* and *prohibited*. Element and group cardinality defaults to *exactly one*, attribute use defaults to *optional*.

Listing 5.1 shows the XSD definition of *PrIML* element `requirement` that holds information about software requirements. The element is defined by an anonymous complex type that is directly inserted as a `xsd:complexType` element (line 2). This type is an `xsd:extension` of the general *PrIML* base type (line 4). The elements `statemachine` (line 6), `usecase_diagram` (line 7), `format:image` (line 8), `val_criteria` (line 9) and `references` (line 10) are specific to the type definition. All of them can, but do not necessarily have to, occur once (attributes `minOccurs` and `maxOccurs` contain these cardinality constraints). The elements are wrapped in an `xsd:sequence` element (line 5) that itself has a `maxOccurs` attribute. In addition to the sub-element structure defining the element type there is an `xsd:unique` construct (line 15) ensuring that requirement IDs are referenced uniquely.

XML Schema describes the structure of the modelled domain and defines the valid syntax for schema-conforming documents. Validating processes first check documents for well-formedness and in case they pass this check positively they are validated against the referenced schema (cf. [W3C08]).

RELAX NG and Schematron Two XML-based specification languages for the description of XML structures that are not as widely adopted as DTDs and XML Schema are RELAX NG [REL11] and Schematron [Sch12].

RELAX NG formulates the goal of providing a simple and easy-to-use way to describe the structure of XML documents. Its instances are themselves XML documents. A full syntax and a simplified syntax are included in the RELAX NG standard. It reuses the XML Schema datatypes and enables support for any external XML-based datatype vocabulary.

Schematron is in contrast to DTDs, XML Schema and RELAX NG not a grammar-based but a tree pattern-based approach for defining XML document structures (cf. [Sch12]).

RELAX NG and Schematron are part of the Document Schema Definition Languages (DSDL) standard family (cf. [Bro10]). They are mentioned here for reasons of completeness, the *PrIML* design decisions in chapter 2.3 made XML Schema the document-describing language for the vocabulary. Thus, XML Schema is the only such language relevant to this thesis' considerations.

5.2 Meta-Modelling

As described above, modelling is the process of representing "real" concepts in an abstract manner. Taking this abstraction one step further, meta-modelling in turn describes the components used in models (cf. [OMG11a], p. 29). The prefix "meta-" is of Greek origin and means – amongst other connotations – "after", "beyond" (cf.

[13]Exact cardinalities are possible in DTDs as well but require a verbose notation.

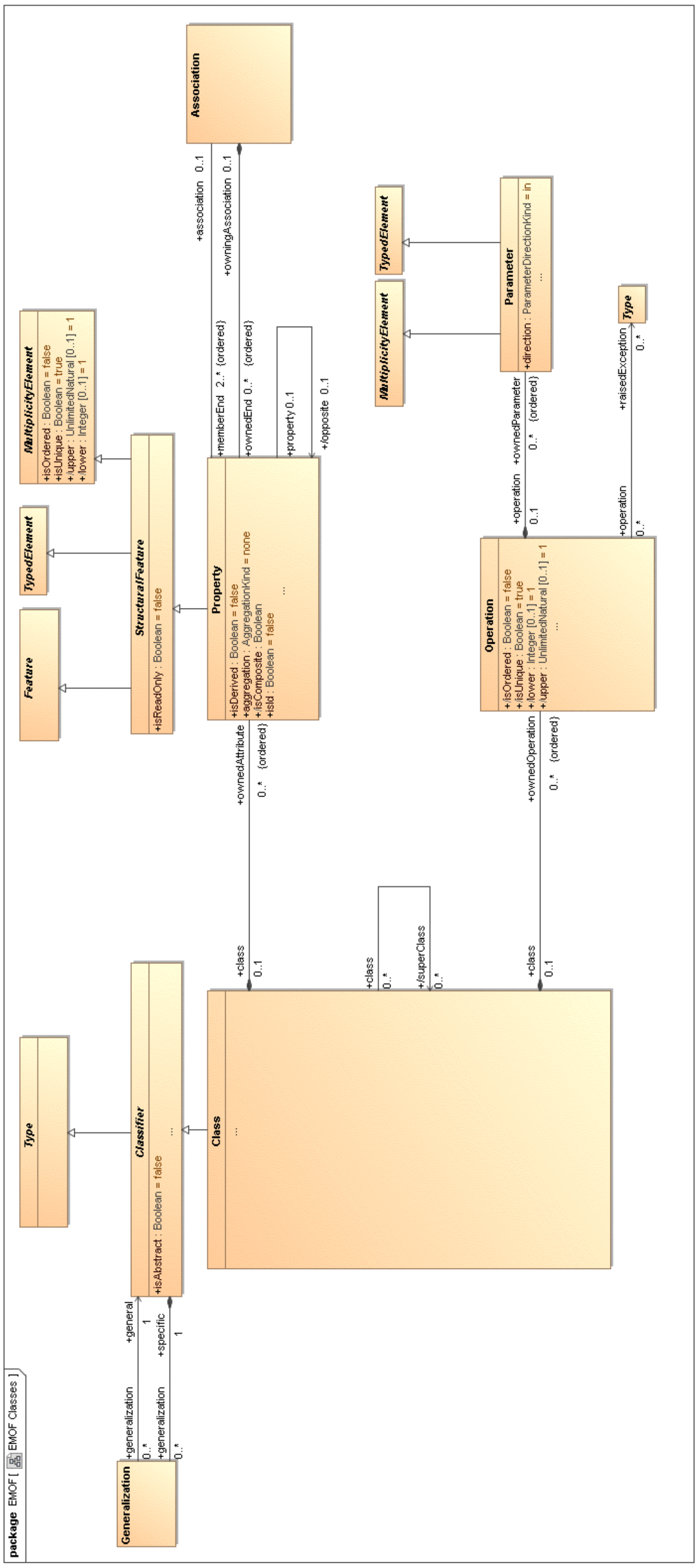


Figure 5.2: OMG EMOF class diagram; [OMG11a], p. 33

[Lid12]). It is a common prefix to express an increased level of abstraction compared to the term prefixed.

The languages used to describe models such as UML and XML Schema are themselves models containing components which form the basis for modelling. Thus they are referred to as meta-models. Taking the abstraction even further, models that describe the structure (i.e., the common features) of meta-models are called meta-meta-models. Although there is theoretically no limit for ever-further abstraction (cf. [Sta07], p. 62), meta-meta-models are usually described with their own concepts and thus the meta-meta-model is the most abstract level explicitly described (cf. [Sta07], p. 31; cf. [Ste09], p. 17). The most prominent meta-meta-model is OMG's Meta Object Facility (MOF). Ecore from the EMF context is its own meta-meta-model ([Ste09], p. 17).

5.2.1 Meta Object Facility

From the OMG standardization consortium [OMG12] responsible for standards like UML and XML Metadata Interchange (XMI) comes a meta-model called MOF. Figure 5.2 depicts the entity types described by OMG's Essential MOF (EMOF) meta-model.

It is not further described because of its lack of applicability but see the similarities to EMF's Ecore below.

5.2.2 Eclipse Modeling Framework and Ecore

In the context of the Eclipse IDE there is the EMF project dealing with modelling, model standard integration and model-driven Java application building (cf. [Ste09], p. xxiii). It has reached a good industry adoption and forms the basis for further UI and general software development steps. Thus it enables the integration of modelling and programming (cf. [Ste09], pp. 15f.).

The core of EMF is the standard Ecore. It is a meta-model that integrates UML, Rational Rose, XML and Java (cf. [Ste09] p. xxiii, p. 14). Mappings from XML Schema to Ecore, UML to Ecore and vice versa respectively make model-to-model-transformations and model-to-code-transformations possible. Furthermore EMF provides mechanisms for generating Eclipse-based editors conforming to the underlying model. Such editors are functional Eclipse plugins that enable users to *Create*, *Retrieve*, *Update* and *Delete* (*CRUD*) model-conforming data instances.

Chapter 6.3 describes the generation steps on top of Ecore and other complementing models. EMF practices the concept of dividing different modelling aspects and referring to these separate models in the generation processes. Figure 5.3 shows the hierarchy of the Ecore meta-model.

5.2.3 Similarities

Ecore[14] and EMOF share some entity types: both contain a `Class` construct which inherits from a more general `Classifier`. Also `StructuralFeatures` are part of both models. EMF calls the specialized manifestations `Attribute`s and `Reference`s, EMOF

[14]The leading *E* in Ecore's entity names is omitted respectively since it carries no semantics.

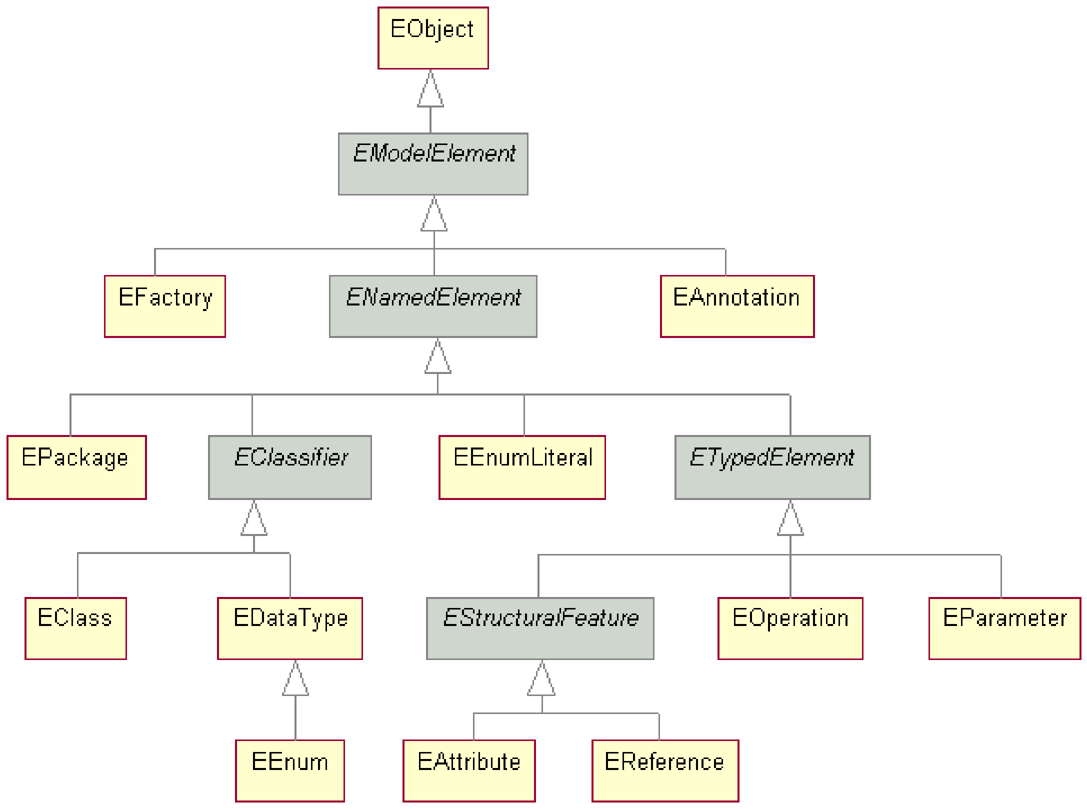


Figure 5.3: Ecore meta-model hierarchy; [Ecl06b]

calls it `Property`. `Operations` and its `Parameters` are another similarity in the constructs of both models.

Ecore has a practical use case, i.e., forming the basis for mappings to XML Schema, UML and especially Java, EMOF does not directly target such a specific use case.

5.3 Relation to Thesis Scope

The approach of generating UI components requires information about the nature of these components. Which kinds of widgets have to be generated and in which way the entity types of the underlying domain relate to each other is necessary for the generation processes. Models like the *PrIML* XSDs contain this sort of information and are used as the parametrization for the code generators.

Meta-modelling and meta-meta-modelling are relevant to this thesis because whenever models and modelling languages (e.g. XML Schema) are used, their structure determines the expressivity of the models described.

6 Practical Solution Approaches

The implementation described in this thesis can be formally expressed as the attempt to weave together different transformation and generation steps that produce editors for XML Schema-defined DSLs. *PrIML* is used as the concrete example of such an XML Schema-based DSL. The goal of this chapter is the description of two different prototypes of semi-automatically developed tools for *CRUD* operations on XML data conforming to the source schema.

Using the conceptual and technical basis of the chapters above this chapter describes the process of choosing the development and runtime framework, the steps taken in the code generation processes and the manual adjustment aspects. Two different approaches are taken to achieve the goal just described: a browser-based HTML / Cascading Style Sheets (CSS) /JavaScript MVC application and an EMF/ Extended Editing Framework (EEF) Eclipse plugin. The prototypes have been developed in the thesis period by the author. The thesis CD-ROM contains screen casts demonstrating the processes and the generated prototypes complementing the descriptions of this chapter and the detailed explanations in appendix A.

Being able to compare is the most important reason for choosing these two parts in the thesis' practical development instead of focusing on only one of them. EMF and the accompanying standards are the example use case for using an existing modelling and generation framework with very high abstraction and several levels of software components interacting with each other. Forming a concept and manifesting it in self-developed generation processes for a web application is the complementary approach – it focuses on light-weight development paradigms. Understanding and using external code generators on the one hand and developing own code generators on the other hand are the two aspects this thesis' chapter demonstrates. In case of the web application development aspect of this chapter there are always two layers of interest that have to be considered: a) the desired architecture of the output application and b) the necessities for the generator functions creating the application components. It needs to be clear what features the application shall have and how they are achievable in order to set up the generator appropriately. These sorts of concepts already exist for EMF/EEF; following and comprehending rather than reinventing this given abstraction hierarchy is essential when using EMF and EEF.

When developing the concepts for the prototypes, the idea of using a UIDL (see chapter 3.2.3) came to mind. Reasons for not using UIDL concepts and languages were:

- anticipated extra costs in terms of several needed transformation steps (according to the author's brief considerations)
  - XML Schema to AUI,
  - AUI to CUI and

  - CUI to FUI
- the lack of a real benefit, since the often-mentioned reuse paradigm does (at least in the direct thesis scope) not apply; EMF/EEF uses its own models and transformation methods and a UIDL is not needed or supported there.

Picking up the working hypothesis that an XML Schema provides enough information for a UI, using a UIDL such as the briefly described UsiXML (see chapter 3.2.3) is omitted and processes directly working with XML Schema information – or directly derived information – are chosen. A transformation of XML Schema constructs to AUI elements is surely contained in the generation processes of both practical solutions, but it is not explicitly expressed separately from the final UI components. An example for such implicit transformations is the design decision to integrate simple-typed subelements of an element type and attributes (that are by definition simple-typed) into object properties. The mechanism of representing elements with an enumeration simple type as "select one of n" interaction objects that are manifested as drop-down menus in the UI is another example for skipping an explicit AUI abstraction layer.

6.1 Requirements for User Interface Solution

The analysis of *PrIML*'s current state and its workflow integration as well as the research on common practices in generative (UI component) programming led to two appropriate prototype solutions. The decision to compare two different approaches has the following reasons:

- it forms a concrete demonstration case for the ability to generate different application prototypes out of **one** source model and hence stresses an advantage aspect of model-based methods,
- each of the two approaches can illustrate individual aspects of generation processes and
- one approach reuses an existing code generator framework, the other one is a manually developed generator combination.

These three main aspects enable a broader and more manifold perspective on generating UI components and the underlying concepts and design decisions that are possible.

The requirements for a *PrIML* user interface and its generation are:[15]

- The UI is able to run on Windows, Linux and MacOS X
- The UI is generated as completely as possible; manual customizations can be applied to fit needs more exactly
- Schema changes are reflected in the UI after a regeneration
- Manual customizations are protected from being overwritten by a regeneration
- The user does not need to enter XML markup
- The user can combine XML data entry and UI-based data entry
- The UI suggests components and/or values where applicable

[15] These were gathered by the ISCUE management and staff together with the author at the beginning of the thesis period as the basic requirement set. Potential refinements are described in the EMF/EEF chapter.

- The UI allows user to deep-copy (and manually adjust) existing components
- The UI provides undo and redo functionality
- Approaches of direct manipulation (e.g., drag&drop) are enabled where applicable
- Element values of mixed content type (especially `format:formattedText` and its sub-types) can be entered with an appropriate rich text editor widget
- *PrIML*'s XSLT processes can be invoked from the UI

The following two chapters 6.2 and 6.3 describe the characteristics of the two approaches taken. Similarities and differences between them are pointed out in chapter 6.4. It will also be discussed to what extent the requirements stated above can be met by the two solutions.

6.2 Browser-Based Application

Applications that run in web browsers build on technologies that exist for several years but have evolved much since then (cf. [Mac11b], p. 1). The most important languages and paradigms are defined first in order to allow for a better comprehensibility.

HTML(5) and CSS(3) HTML is the basic markup language for hypertext documents and its latest development version is 5. The W3C is mainly responsible for its state, drafts and Application Programming Interfaces (APIs). HTML5 [W3C12b] targets a greater flexibility in terms of syntactical constructs, introduces language elements for better semantic markup reflection (`section`, `article`, etc.) and provides APIs for e.g. client-side data storage and canvas-based graphics rendering. While HTML is responsible for the markup, structure and content of web pages, CSS provides constructs for styling these elements. Positioning, visibility, colors, borders and the appearance of form elements are key components of this style sheet standard. There exist libraries for cross-browser support and richer features in the style sheet context as well – Syntactically Awesome Stylesheets (SASS) [Cat12], LESS [Sel12] and Stylus [Sty12] are some of the attempts to create super-sets of CSS and enriching it with variable support, function definitions and thus a higher reusability and consistency.

JavaScript/ECMAScript/CoffeeScript JavaScript is a scripting language originally only used for singular use cases of letting a web page respond to events or dynamically change content. With its standardization as *ECMAScript* [ECM12] it could gain importance over the years and is by now the most essential programming language in client-side web development. Other languages such as CoffeeScript [Cof11] are designed as super-sets of JavaScript, introducing syntax shortcuts, providing a native class support that JavaScript itself lacks and offering a clearer syntax in general. CoffeeScript compiles into JavaScript and is becoming popular in web application development (cf. [O'G12]).

Document Object Model and JavaScript Frameworks The aspect of dynamically changing web page content in reaction to user interaction (*events*) and data updates makes access to the page's structure necessary. The W3C standard DOM encapsules

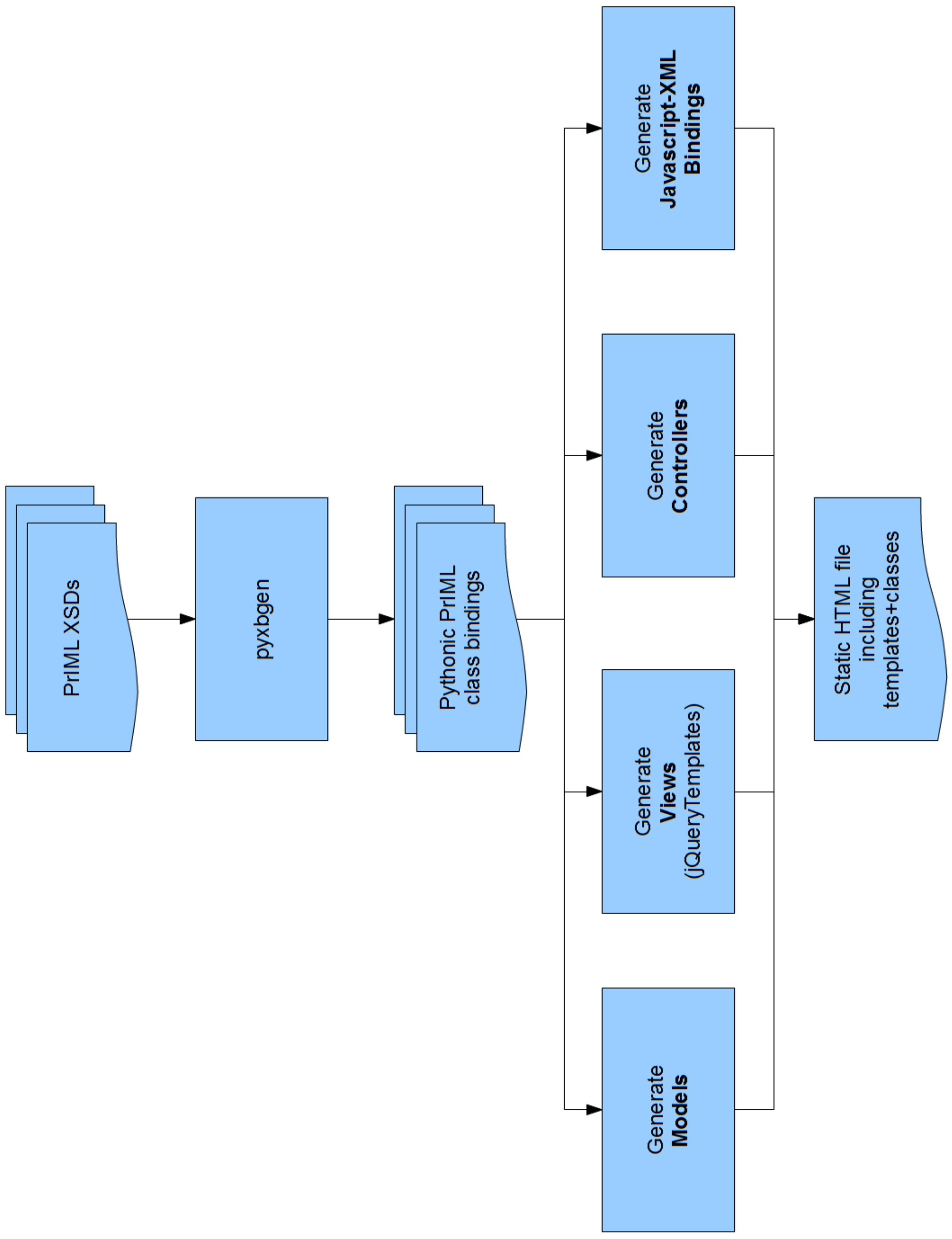

PrIML XSDs
pyxbgen
Pythonic PrIML class bindings
Generate Javascript-XML Bindings
Generate Controllers
Generate Views (jQueryTemplates)
Generate Models
Static HTML file including templates+classes

Figure 6.1: Workflow generation of the browser-based application

the HTML page structure and allows access via a JavaScript API. Every browser engine supports its own manipulation methods and uses a different nomenclature. For the integration and consistent access methods there are JavaScript framework libraries such as Prototype [Pro12], mootools [moo12] or the wide-spread jQuery [jQu12a] that introduce an intermediate abstraction layer. The API they provide is internally mapped to the specific browser engines and versions available and is thus hiding these differences from the developer.

AJAX and JSON For asynchronous communication between server and client side JavaScript provides XMLHTTPRequests [W3C11b]. As with DOM manipulation, every browser has slightly different notions and syntax conventions. jQuery for example offers comfortable ways for wrapping these specifica. The paradigm of asynchronous requests from the client to the server, the mechanisms of reacting to responses and the processing of these data responses in callback functions is called Asynchronous JavaScript and XML (AJAX). Originally XML was the designated serialization format because of its standardized syntax and wide-spread tool support. Over the years the advantages of the more light-weight and immediately JavaScript-digestible JSON serialization format [Cro02] has lead to often JSON-focused AJAX transfers.

Model-View-Controller on client-side The evolution of web pages to often fully featured applications in the browser raises the necessity of software architecture, especially on the client-side. Since for example PHP Homepage Pre-Processor (PHP) and Ruby development libraries already support the MVC paradigm for some time (cf. [Zen12]; cf. [Han12]) it became only recently common on the client-side to apply architectural concepts. The needs for maintainabilty, stability and flexibility apply for web applications as well as for native client software (cf. [Mac11b], p. 2). In the last years many MVC libraries for JavaScript were developed with comparable concepts but with different levels of formalization (cf. [Osm12]). Spine [Mac12] and Backbone [Doc12] for example are more light-weight than frameworks such as ExtJS [Sen12].

These standards briefly defined above form the general basis for (especially the client-side of) web applications. The prototype introduced in this chapter builds on top of them as well and adds the aspect of how to generate many parts of such an application based on the existing *PrIML* XSDs. The desired architecture is the MVC paradigm, Spine and Backbone are both adapted. Spine is itself written in CoffeeScript and thus seamlessly supports CoffeeScript, it is the scripting language of choice for the Spine components. Backbone (and specifically the extension Backbone-relational, that is used in the prototype) has reported limitations in supporting CoffeeScript, hence JavaScript is also a target generation language. As stated before, the graphical and usability aspect of the application is not the main focus, the development is designed to be a proof of concept for generating models, views and templates in order to have a working interface. Another goal of the web application is to demonstrate the power of client-side applications – the prototype does not need a server side in order to be functional. For persistence reasons a server side would become necessary but the current state of the application is not depending on it. For considerations about moving state to the client-side cf. [Mac11a].

Figure 6.1 shows the workflow to generate the browser-based application. The *PrIML* XSDs are processed by the Python XML Bindings (PyXB) generator script `pyxbgen` which produces one Python module per input XSD file. Out of these there are generated three kinds of application components: models, views and controllers. The models are manifested in the form of `Backbone.RelationalModel`s, views are realized through jQuery templates [jQu12b] and the controllers conform to the `Spine.Controller`s class [Mac12]. In addition to these three sorts of components, Jsonix bindings complement the application in order to serialize the data to XML.

Figure 6.2 shows a screen shot of the browser window containing a requirement group with its XML serialization.

Following the MVC paradigm, the following three chapters describe the steps taken to generate models, views and controllers. CoffeeScript is chosen complementarily to JavaScript as the browser scripting language because of the following reasons:

- native class support (calling `super` calls the same method of a potential super class; this feature is more verbose in plain JavaScript),
- cleaner syntax and thus a better readability,
- more elaborate language constructs and
- consistent compilation into JavaScript.

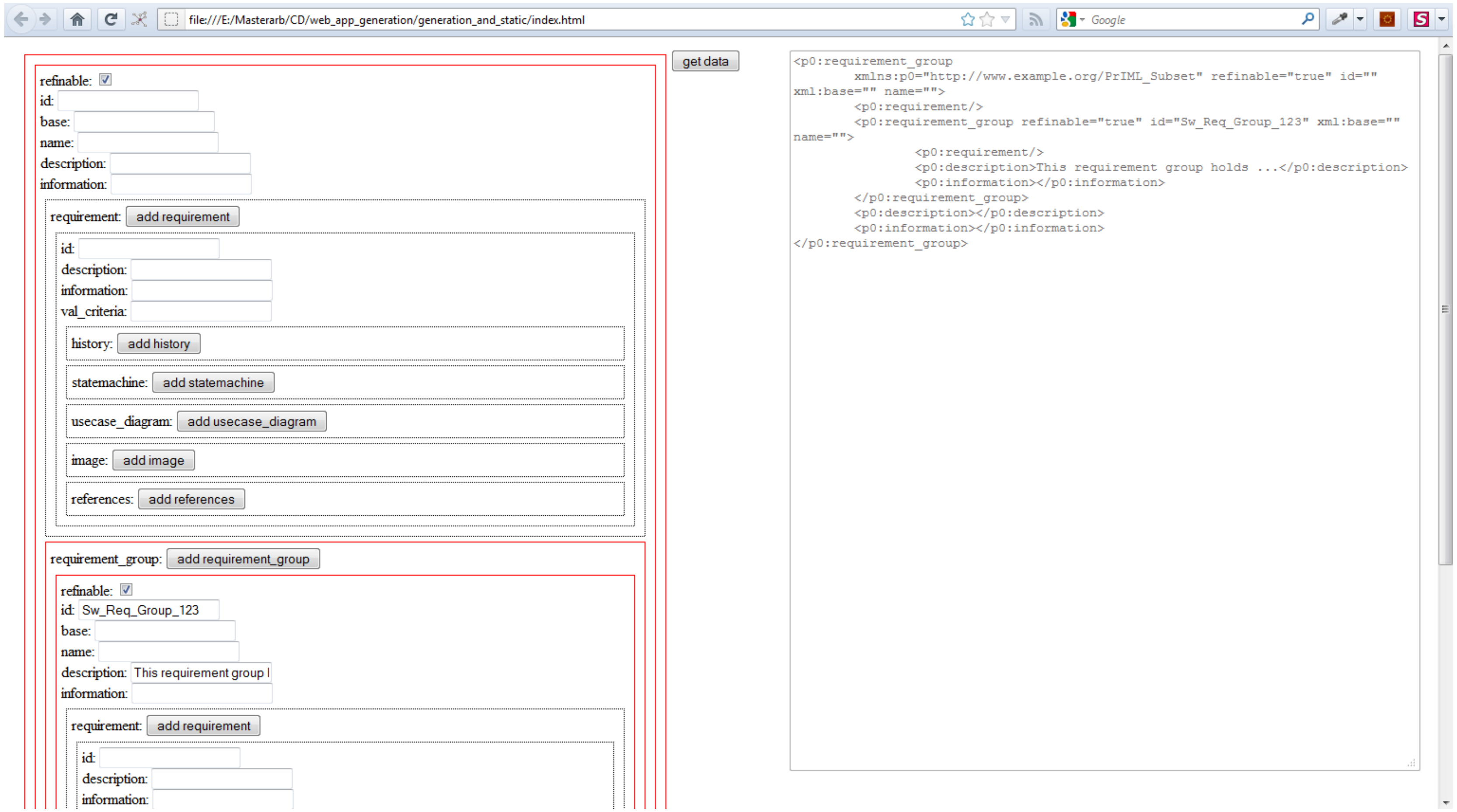
6.2.1 Generating Models

The original approach in creating models was to only use the Jsonix library [Jso12] for JavaScript-XML bindings. The library is designed to provide light-weight mechanisms to

- enable users to define mappings from XML to JavaScript (and vice versa) by hand,
- provide methods to marshal and unmarshal data and
- generate bindings automatically through a Java command line tool.

Jsonix is also available as a Java XML Bindings (JAXB) [Jav12] plugin and can be customized in the same way as JAXB through `.xjb` binding directives. In first test generations Jsonix proved very promising in finding a way to marshal and unmarshal data directly on the application's client side. More detailed analysis showed that a direct, uncustomized form of the XSD-to-JavaScript generation script contained inconsistencies. Elements with the same name (but in different schema context, i.e., of different anonymous types) were not resolved to different classes and thus information loss occurred. Manually writing or generating such bindings is anyway possible and these processes can themselves pay attention to consistency. See chapter 6.2.4 for a description of this prototype's approach to the generation of this kind of mappings.

Further research for similar tools and having the further use with generation scripts in mind led to the PyXB Python module [Big12]. It has the same goal, generating bindings, but targeting Python classes. It did not show similar inconsistencies as described above but resolves elements and types bi-uniquely and is even more verbose



file:///E:/Masterarb/CD/web_app_generation/generation_and_static/index.html

refinable: ✓
id:
base:
name:
description:
information:

requirement: add requirement
id:
description:
information:
val_criteria:
history: add history
statemachine: add statemachine
usecase_diagram: add usecase_diagram
image: add image
references: add references

requirement_group: add requirement_group
refinable: ✓
id: Sw_Req_Group_123
base:
name:
description: This requirement group l
information:
requirement: add requirement
id:
description:
information:

get data

```
<p0:requirement_group
        xmlns:p0="http://www.example.org/PrIML_Subset" refinable="true" id=""
xml:base="" name="">
        <p0:requirement/>
        <p0:requirement_group refinable="true" id="Sw_Req_Group_123" xml:base=""
name="">
                <p0:requirement/>
                <p0:description>This requirement group holds ...</p0:description>
                <p0:information></p0:information>
        </p0:requirement_group>
        <p0:description></p0:description>
        <p0:information></p0:information>
</p0:requirement_group>
```

Figure 6.2: Web application screen shot

when it comes to reflecting cardinalities and grouping in the XSD definitions. Further advantages are a) Python's convenient mechanisms for inspecting its own structures (i.e., modules, classes, etc.) as an elegant basis for further generation steps and b) the fact that Python is more and more widely adapted as a server-side scripting language in web applications.

The author developed the Python module `pyxb_to_mvc_web_app` with the processes and methods necessary for generating the models as extensions of `RelationalModel` out of the Python classes provided by PyXB. A two-fold process was applied:

1. resolving a Python dictionary which directly reflects the (potentially deeply nested) XML Schema groups, elements, attributes, their cardinalities and fixed/ default values and
2. flattening these dictionaries into structures that provides a list of possible child elements and attributes with their "acummulated" `minOccurs` computed as the product of all `minOccurs` from the respective element up to the root XSD group and the `maxOccurs` computed analogously (any `unbounded` value for a `maxOccurs` turns the end result into `unbounded`)

Flattening the constructs from XSD leads to informatin loss since a flat element-attribute list does not in all cases reflect the expressivity that XML Schema's grouping mechanisms provide, especially in combination with the cardinalities applicable on almost all construct levels. Accummulating the cardinalities (see listing 6.1) transports most of the original cardinality information, but the order of groups is an aspect that is lost by the flattening. This trade-off is made in order to prevent the processes and the resulting UI from becoming too cumbersome and verbose. Reflecting every group and cardinality detail in the UI would possibly create contra-intuitive widget combinations since the group level of XSDs is not designated to be explicitly perceived by end users. This simplification thus is a method of hiding originally existing complexity from the UI user – the generation processes do need to handle the full complexity to such a degree as the flattening needs to be applied to the original complex structures.

A model is created for each complex XML Schema type and each such model is an extension of `Backbone.RelationalModel`. Within a model class every simple-typed sub-element and every attribute of the flattened structure derived from XSD leads to a data field. Complex-typed sub-elements are reflected as relations linking to the respective sub-element's model class. The model for the requirement group is shown in listing 6.2 (see appendix A.1.2 for explanations on nomenclatures for anonymous types).

6.2.2 Generating Views

In light-weight web development libraries such as Spine, MVC-conforming views are often HTML fragments rendered into the DOM (cf. [Mac12]). These fragments can be represented by templates (cf. chapter 4.2), in this case jQuery templates. These offer common template functionality such as conditional rendering of content, invoking other templates for rendering sub-parts and loop constructs. There are basically two ways of storing jQuery templates: a) in `script` elements of their own, marked with a `type="text/x-jquery-tmpl"` attribute in order to distinguish them from usual Java-

Script `script` elements, and b) in string variables. In this case string persistence seems more appropriate since it allows for the processes to store all templates in one JavaScript file (potentially even in one data structure) and reference this one file from the static HTML file that bundles the application parts. This abstracts out the concrete number of templates generated and their specific names. Including the template JavaScript file which can follow a static nomenclature will always ensure covering all generated templates.

Data type	**HTML construct**
`xsd:string`	`<input type="text" />`
`xsd:boolean`	`<input type="checkbox" />`
`xsd:decimal`	`<input type="number" />`
`xsd:float`	`<input type="number" />`
`xsd:double`	`<input type="number" />`
`xsd:duration`	`<input type="text" />`
`xsd:dateTime`	`<input type="dateTime" />`
`xsd:time`	`<input type="time" />`
`xsd:date`	`<input type="date" />`
`xsd:gYearMonth`	`<input type="date" pattern="([-]?[0-9]{4,}[-][0-9]{2})" />`
`xsd:gYear`	`<input type="date" pattern="([-]?[0-9]{4,})" />`
`xsd:gMonthDay`	`<input type="date" pattern="([0-9]{2}[-][0-9]{2})" />`
`xsd:gDay`	`<input type="date" pattern="([0-3][0-9])" />`
`xsd:gMonth`	`<input type="date" pattern="([01][0-9])" />`
`xsd:hexBinary`	`<input type="text" />`
`xsd:base64Binary`	`<input type="text" />`
`xsd:anyURI`	`<input type="url" />`
`xsd:QNAME`	`<input type="text" pattern="([a-zA-Z_][a-zA-Z_0-9-.]* ([:][a-zA-Z_][a-zA-Z_0-9-.]*)?)"/>`
`xsd:NOTATION`	`<input type="text" />`
`xsd:normalizedString`	`<input type="text" pattern="([^\t^\n^\r ]*)" />`
`xsd:token`	`<input type="text" pattern="([^\t^\n^\r ]*)" />`
`xsd:language`	`<input type="number" pattern="[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*" />`
`xsd:NMTOKEN`	`<input type="text" />`
`xsd:NMTOKENS`	`<input type="text" />`
`xsd:Name`	`<input type="text" pattern="([a-zA-Z_:][a-zA-Z_0-9-.:]*)" />`

`xsd:NCName`	`<input type="text" pattern="([a-zA-Z_][a-zA-Z_0-9-.]*)" />`
`xsd:ID`	`<input type="text" pattern="([a-zA-Z_][a-zA-Z_0-9-.]*)" />`
`xsd:IDREF`	`<input type="text" pattern="([a-zA-Z_][a-zA-Z_0-9-.]*)" />`
`xsd:IDREFS`	`<input type="text" pattern="(([a-zA-Z_][a-zA-Z_0-9-.]*[ ])*)" />`
`xsd:ENTITY`	`<input type="text" pattern="([a-zA-Z_][a-zA-Z_0-9-.]*)" />`
`xsd:ENTITIES`	`<input type="text" pattern="([a-zA-Z_][a-zA-Z_0-9-.]*([ ] [a-zA-Z_][a-zA-Z_0-9-.]*)*)" />`
`xsd:integer`	`<input type="number" min="-2147483648" max="2147483647" step="1" />`
`xsd:nonPositiveInteger`	`<input type="number" max="0" step="1" />`
`xsd:negativeInteger`	`<input type="number" max="-1" step="1" />`
`xsd:long`	`<input type="number" min="-9223372036854775808" max="9223372036854775807" step="1" />`
`xsd:int`	`<input type="number" min="-2147483648" max="2147483647" step="1" />`
`xsd:short`	`<input type="number" min="-32768" max="32767" step="1" />`
`xsd:byte`	`<input type="number" min="-128" max="127" step="1" />`
`xsd:nonNegativeInteger`	`<input type="number" min="0" step="1" />`
`xsd:unsignedLong`	`<input type="number" min="0" max="18446744073709551615" step="1" />`
`xsd:unsignedInt`	`<input type="number" min="0" max="4294967295" step="1" />`
`xsd:unsignedShort`	`<input type="number" min="0" max="65535" step="1" />`
`xsd:unsignedByte`	`<input type="number" min="0" max="255" step="1" />`
`xsd:positiveInteger`	`<input type="number" min="1" step="1" />`

Table 6.1: Mapping between XML Schema types and HTML constructs; cf. [W3C04d]

The Python module for generating the models based on PyXB's output was complemented by functions for generating the view templates consistently. The flattened elements and attributes dictionary described above was reused for deriving the template strings from it. Every type (simple, complex and even each of the native XML Schema data types) gets its own template. Those for simple types contain form elements reflecting the underlying type definition (see table 6.1). Enumeration types are transformed into `<select>`-`<option>` blocks.[16] Every extension of `xsd:string` with a pattern attribute is mapped to an `<input type="text" pattern="..."/>` where the dots in the pattern attribute are replaced by all patterns the XSD provides (there might be arbitrarily many) separated by pipe characters (|) and wrapped by brackets altogether. Thus all patterns are taken into account when it comes to checking the validity of the entered text.

Complex type templates contain areas for possible sub-elements. Complex sub-elements can be added via buttons, simple ones are directly inserted – once in case they are optional or required exactly once, `minOccurs` times in all other cases. It is necessary to use a Pythonic templating system to generate jQuery templates out of the type definitions. The templating library *mako* [Bay12] was chosen because of its rich feature support. Since both templating systems (*mako* and *jQuery tmpl*) share common expression syntax constructs for evaluation contexts such as dollar signs ($) and curly brackets ({ and }), escaping these characters in the Python template strings is necessary in order to transport them into the jQuery template output. Listing 6.3 shows the template for requirement groups.

6.2.3 Generating Controllers

Being the connection between models and views, the controllers have to listen to a) UI interaction events as the clicks on addition and deletion buttons and b) model events (creation, update, etc.) of data instances and updates of sub-elements. Spine provides a `Spine.Controller` class with common methods. Mouse events as clicks on *add ...* buttons can be mapped to class methods as the event handlers in the `events` map. Methods for the creation and addition of sub-elements and their respective controllers are asserted on the class and connected to events by attaching additional event handlers. Rendering is accomplished by the `render` method, `save_changes` sets values on simple-types elements and attributes. See figure 6.4 for the requirement group controller.

For each newly added sub-element a new controller of the appropriate kind is generated and is responsible for handling updates of the sub-element's properties.

6.2.4 Generating JavaScript-XML bindings

Jsonix requires three parts in a JavaScript-XML mapping package. First of all, all complex types have to be declared in order to make them referencable. After these declarations that only set up the respective object and a name property, all complex types are defined in terms of their content model. Attributes and sub-elements are listed and their types are referenced (targeting the aforementioned declarations). The last information necessary for marshalling JavaScript objects into XML is a list of all possible top-level elements.

Generating these three parts is achievable in a straight-forward way, iterating all complex types of the PyXB package in question. Listing 6.5 contains the parts for mapping requirement group JavaScript objects to their XML serialization.

[16]Mechanisms for a conditional rendering of five or less enumeration values as radio button groups were considered but omitted; this distinction could be added for usability reasons

```
{
  "attributes": [
    {
      "dataType": {
        "namespace": "http://www.iscue.com/xml/2008/11/priml_data",
        "local": "primlID"
      }
      "defaultValue": null,
      "required": true,
      "name": {
        "namespace": null,
        "local": "id"
      },
      "fixed": false
    },
    [...]
  ],
  "elements": [
    {
      "elementName": {
        "namespace": "http://www.iscue.com/xml/2008/11/priml_data",
        "local": "information"
      },
      "minOccurs": 0,
      "maxOccurs": 1,
      "type": {
        "namespace":"http://www.iscue.com/xml/2008/11/priml_format",
        "local": "formattedText"
    },
    {
      "elementName": {
        "namespace": "http://www.iscue.com/xml/2008/11/priml_data",
        "local": "requirement"
      },
      "minOccurs": 0,
      "maxOccurs": null,
      "type": "CTD_ANON_72"
    },
    [...]
  ],
  "name": "CTD_ANON_60"
}
```
Listing 6.1: Complex type for requirement groups flattened (serialized in JSON; extract)

```
var CTD_ANON_3 = Backbone.RelationalModel.extend({
  urlRoot: '/CTD_ANON_3',
  defaults: {
    "base":
      null,
    "id":
      null,
    "refinable":
      true,
    "name":
      null,
    "description":
      null,
    "information":
      null
  },
  relations: [
    {
      type: Backbone.HasMany,
      key: "requirement",
      relatedModel: "CTD_ANON",
      includeInJSON: true
    },
    {
      type: Backbone.HasMany,
      key: "requirement_group",
      relatedModel: "CTD_ANON_3",
      includeInJSON: true
    }
  ]
});
```

Listing 6.2: Backbone.RelationalModel definition for requirement groups

```
<div id="CTD_ANON_3_${id}">
  name:
  {{if $data.name!=null}}
    {{tmpl(null, {templates: $item.templates, value:
       $data.name, name: "name", fixed: false}) $item.
       templates['XSD__string']}}
  {{else}}
    {{tmpl(null, {templates: $item.templates, value:
       null , name: "name"}) $item.templates['XSD__string
       ']}}
  {{/if}}
  <!-- ... -->
  <div>
    requirement:
    {{if $data.requirement!=null && $data.requirement.
       length>0}}
      {{tmpl(requirement, {templates: $item.templates})
         $item.templates['CTD_ANON']}}
    {{/if}}
    <button type="button" class="add_requirement">add
       requirement</button>
  </div>
  <div>
    requirement_group:
    {{if $data.requirement_group!=null && $data.
       requirement_group.length>0}}
      {{tmpl(requirement_group, {templates: $item.
         templates}) $item.templates['CTD_ANON_3']}}
    {{/if}}
    <button type="button" class="add_requirement_group">
       add requirement_group</button>
  </div>
</div>
```
Listing 6.3: jQuery template for requirement group

```
class CTD_ANON_3_Controller extends Spine.Controller
  events:
    "click div > button.add_requirement": "create_requirement"
    "click div > button.add_requirement_group": "create_requirement_group"
    "change input": "save_changes"
    "change select": "save_changes"

  save_changes: (event) =>
    try
      event.stopPropagation()
    catch e
      #

    @item.set({base: $(@el).children("*[name='base']").val(), id: $(@el).children("*[name='id']").val(), refinable: Boolean($(@el).children("input[name='refinable']").is(':checked')), name: $(@el).children("*[name='name']").val(), description: $(@el).children("*[name='description']").val(), information: $(@el).children("*[name='information']").val() }, { silent: true })

    @item.change()

  render: =>
    @replace($.tmpl(templates['CTD_ANON_3'], @item.toJSON(), {templates: templates}))
    @

  remove: =>
    @el.remove()

  constructor: (options) ->
    super

    @item.bind('add:requirement update:requirement remove:requirement add:requirement_group update:requirement_group remove:requirement_group', @save_changes)
    @item.bind('destroy', @remove)
    @item.bind('add:requirement', @add_requirement)
    @item.bind('add:requirement_group', @add_requirement_group)

  add_requirement: (requirement) =>
    requirement_ = new CTD_ANON_Controller(item: requirement, el: $('#requirement_' + requirement.id))
    @el.children('div:has(button.add_requirement)').append(requirement_.render().el)

  create_requirement: (event) =>
    event.stopPropagation()
    tmp=new CTD_ANON()
    @item.get('requirement').add(tmp, {silent: true})
    @item.trigger('add:requirement', tmp)

  add_requirement_group: (requirement_group) =>
    requirement_group_ = new CTD_ANON_3_Controller(item: requirement_group, el: $('#requirement_group_' + requirement_group.id))
    @el.children('div:has(button.add_requirement_group)').append(requirement_group_.render().el)

  create_requirement_group: (event) =>
    event.stopPropagation()
    tmp=new CTD_ANON_3()
    @item.get('requirement_group').add(tmp, {silent: true})
    @item.trigger('add:requirement_group', tmp)
```
Listing 6.4: Spine.Controller for requirement groups in CoffeeScript

```
_nsgroup.CTD_ANON_4 = new Jsonix.Model.ClassInfo(
  {
    name: '_nsgroup.CTD_ANON_4'
  }
);
[...]
{
  {
    _nsgroup.CTD_ANON_4.properties = [
      new Jsonix.Model.ElementPropertyInfo({
        name: 'requirement',
        collection: true,
        elementName: new Jsonix.XML.QName('http:\/\/www.
           example.org\/PrIML_Subset', 'requirement'),
        typeInfo: _nsgroup.CTD_ANON,
      }),
      new Jsonix.Model.ElementPropertyInfo({
        name: 'information',
        elementName: new Jsonix.XML.QName('http:\/\/www.
           example.org\/PrIML_Subset', 'information'),
        typeInfo: Jsonix.Schema.XSD.String.INSTANCE,
      }),
      new Jsonix.Model.AttributePropertyInfo({
        name: 'refinable',
        typeInfo: Jsonix.Schema.XSD.Boolean.INSTANCE,
        attributeName: new Jsonix.XML.QName('refinable')
      }),
      [...]
      ];
  },
  [...]
}
_nsgroup.Mappings.elementInfos = [
  {
    elementName: new Jsonix.XML.QName('http:\/\/www.
       example.org\/PrIML_Subset', 'requirement_group'),
    typeInfo: _nsgroup.CTD_ANON_4,

  },
  [...]
]
```

Listing 6.5: Jsonix mapping for requirement groups (extract)

6.3 Plugin Using Eclipse Modeling Framework

6.3.1 Overview

Using Eclipse and its modelling framework components such as EMF, EEF and others has the advantage of providing a very high-level infrastructure that can be built upon. Being itself a combination of plugins, Eclipse offers its functionalities such as window management with perspectives and views, file operations, undo stack handling and deep Java integration. From ISCUE's institutional point of view, Eclipse is the essential software development tool. Integrating the *PrIML* UI into Eclipse is an obvious consideration since it exploits an already set-up infrastructure and avoids adding more software tools to the workflow chain.

The EMF project provides a framework for generating Eclipse plugins that are executable using the Eclipse workbench. Such plugins can be generated based on different sorts of model definitions. XML Schema is one of the possible input formats, which makes it highly relevant to this thesis' goal. There exists a wizard that accomplishes all basic generation tasks from XSD input to the executable Eclipse plugin (cf. figure 6.3).

Without the necessity for manual adjustment there is a fully functional editor based on a tree view representing the instance's hierarchy. This tree provides all common operations such as adding children elements, copying, pasting, moving and deleting. It reflects the underlying XML structure quite directly. Syntactical characteristics as XML's distinction between elements and attributes are abstracted out – all these properties are considered to be *EAttributes* as EMF and Ecore call them. These are mapped appropriately to the desired serialization form (XML element or attribute) by applying the XSD2Ecore mapping directives generated along with the Java classes.

The edition of the attribute and element values happens in the respective property views. These are table views reflecting the key-value structure of objects.

Strings and derived types thereof are represented by text input fields with potential pattern constraints applied, enumeration types are transformed into drop-down select boxes by default.

In the background, this Eclipse plugin consists of several Java class hierarchy levels. Classes for all Ecore `EClass`es (which are in turn created for each complex XSD input type) are generated. These are bundled in packages, one for each namespace contained in the XML Schema definition. EMF.Edit adds ItemProvider classes for each complex type which supplies the editor with the instance data. This is for example exploited when creating the tree view. The respective `getText()` method on ItemProviders is responsible for providing a human-readable label displayed in the tree (and later in the EEF-generated input forms). EMF.Editor adds all classes providing UI functionality including for example event handlers for tree view and property views.

Building upon this EMF plugin there can be enhanced property views provided by the EEF project, which is also part of the Eclipse modelling context. EEF has the goal of enabling the developer to generate more comfortable editing components than the default Eclipse table-based views. Composition of forms and thus aggregating otherwise split-up form fragments lies in EEF's focus. EEF complements EMF with two further models: the EEF `.components` model describing the views and model bindings and

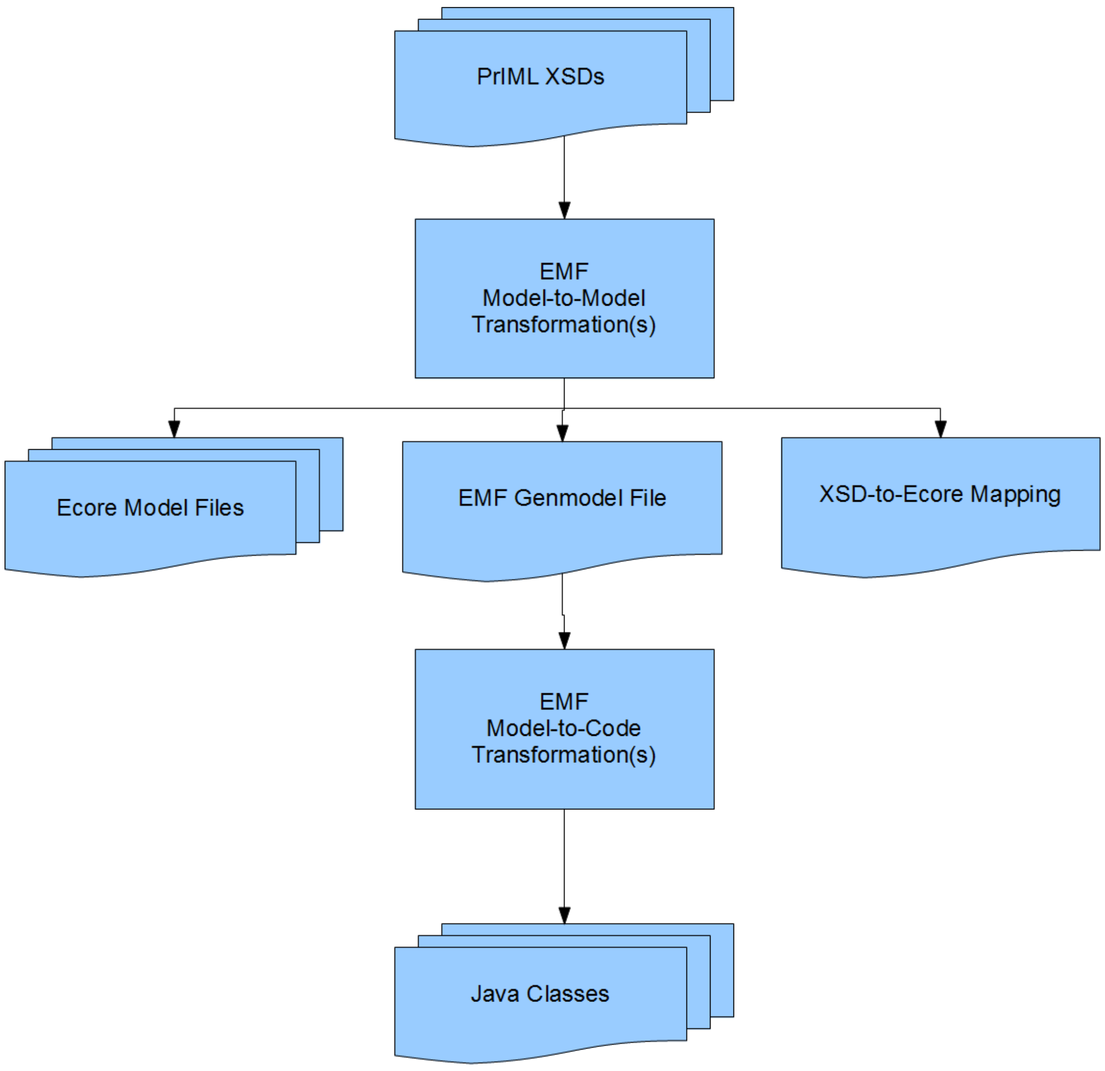
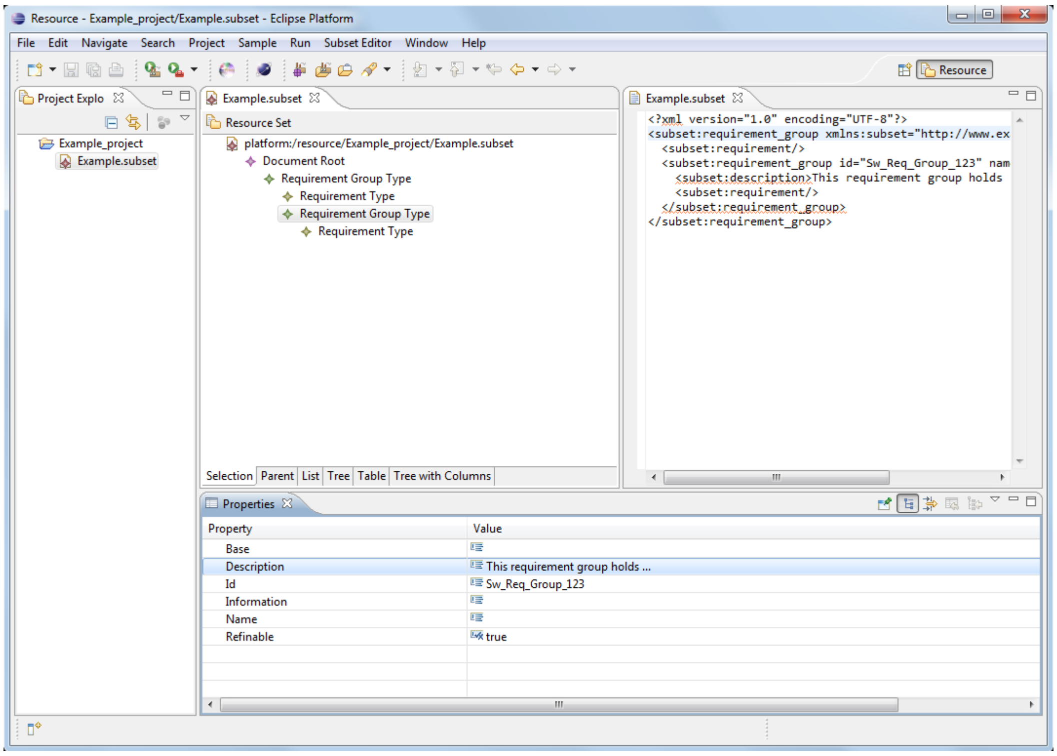


Figure 6.3: Generation steps EMF


Resource - Example_project/Example.subset - Eclipse Platform
File Edit Navigate Search Project Sample Run Subset Editor Window Help
Resource
Project Explo
Example_project
Example.subset
Example.subset
Resource Set
platform:/resource/Example_project/Example.subset
Document Root
Requirement Group Type
Requirement Type
Requirement Group Type
Requirement Type
Selection Parent List Tree Table Tree with Columns
Example.subset
Properties

```
<?xml version="1.0" encoding="UTF-8"?>
<subset:requirement_group xmlns:subset="http://www.ex
  <subset:requirement/>
  <subset:requirement_group id="Sw_Req_Group_123" nam
    <subset:description>This requirement group holds
    <subset:requirement/>
  </subset:requirement_group>
</subset:requirement_group>
```

Property	Value
Base	
Description	This requirement group holds ...
Id	Sw_Req_Group_123
Information	
Name	
Refinable	true

Figure 6.4: Generated PrIML EMF Editor showing tree, XML and property view

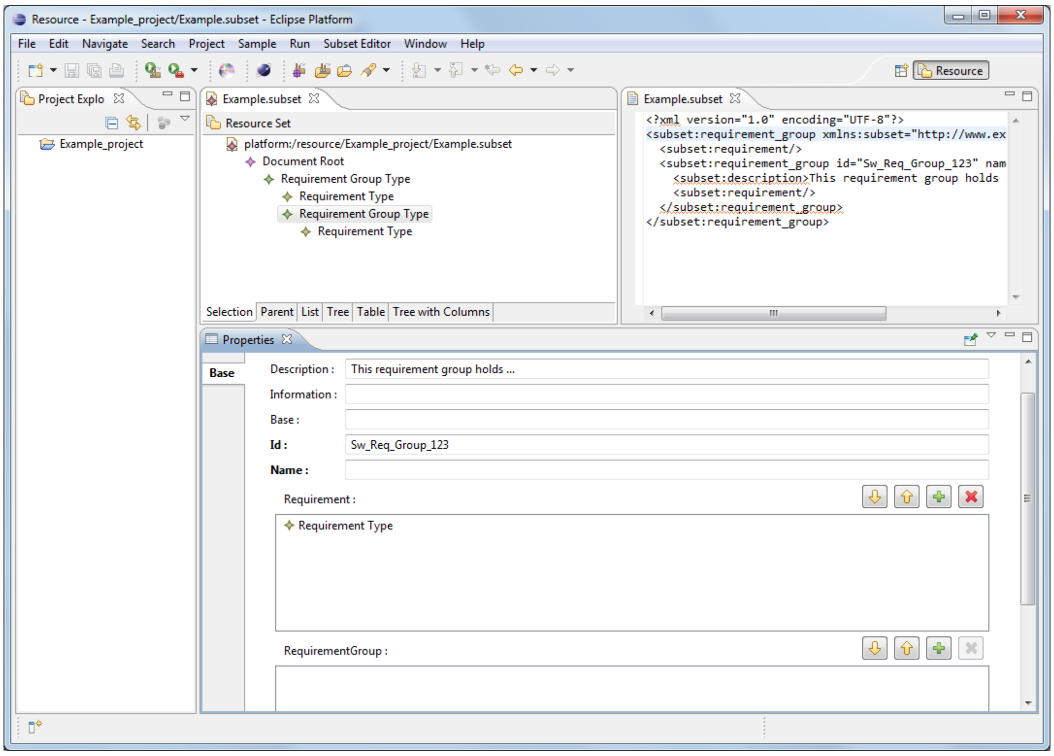
the `.eefgen` model which takes a similar role as the `.genmodel` in EMF. These models can be edited as plain XML files or with the EEF-specific model editor that arranges the components in a tree view.[17] The generation is accomplished by Acceleo [Ecl12a] templates. Developers can apply extensions in order to complement the generated classes and thus customize the resulting views. See screen shots of the basic EMF editor (figure 6.4) and the enhanced EEF editor (figure 6.5). The property views are depicted in the respective lower halves of the figures.

6.3.2 Generation Processes and Development Steps

The wizard for generating EMF.Edit and EMF.Editor plugin components out of XML Schema collections does not take many parameters except the file name(s) of the input XSDs. Influencing the output of the generation can be accomplished by complementing the underlying XML Schema definitions with hints. For all transformation cases there exist default values and behaviors but these do not always deliver the desired output.

First generations were applied to the complete *PrIML* XSD collection without any Ecore hints or manipulations. The produced EMF plugins were fully functional and provided a combination of tree view and property view. The functionality directly visible did not exceed the functionality of Eclipse's native XML perspective much. Property views as EMF uses them by default are highly formalized, the tree view

[17]This editor seems to be not working under Linux, but no such bug report could be found.


Resource - Example_project/Example.subset - Eclipse Platform
File Edit Navigate Search Project Sample Run Subset Editor Window Help
Resource
Project Explo
Example_project
Example.subset
Resource Set
Selection Parent List Tree Table Tree with Columns
Properties
Base

```
platform:/resource/Example_project/Example.subset
  Document Root
    Requirement Group Type
      Requirement Type
      Requirement Group Type
        Requirement Type
```

```
<?xml version="1.0" encoding="UTF-8"?>
<subset:requirement_group xmlns:subset="http://www.ex
  <subset:requirement/>
  <subset:requirement_group id="Sw_Req_Group_123" nam
    <subset:description>This requirement group holds
    <subset:requirement/>
  </subset:requirement_group>
</subset:requirement_group>
```

Description :	This requirement group holds ...
Information :	
Base :	
Id :	Sw_Req_Group_123
Name :	
Requirement :	Requirement Type
RequirementGroup :	

Figure 6.5: Generated PrIML EEF Editor showing tree, XML and property view

reflects the XML tree hierarchy.

In the next step, EEF's generation processes were applied to EMF's output. See figure 6.6 for an overview of the steps taken and for detailed textual descriptions of all these generation steps cf. appendix A.2. As with the basic EMF generations, the EEF processes were tried unparametrized. The result after applying the steps described in EEF's user guide [Ecl12b] was a partially working, enhanced version of the EMF.Editor plugin. The property views did not consistently behave as expected, for some elements the properties were not represented by form widgets. In these cases there was one text input field and eventual child elements were expanded in textual form into this input. Any property views for elements further down the hierarchy could not be displayed, instead the Eclipse default error message "Properties not available." was displayed.

In order to resolve this behavior the approach was to isolate a distinct subset of the *PrIML* vocabulary. The subset was chosen with respect to having one or more seemingly misbehaving element structures. Reducing the amount of involved elements improved generation performance during the often repeated steps and the maintainability in general. Based on the `requirement_group` element which bundles the descriptions and relations of associated system or software requirements the subset was formed. All referenced types and elements were collected and persisted into one separate XML Schema file. Two simplifications were applied for this test subset: the occurrences of the mixed type `format:formattedText`[18] were replaced by `xsd:string` and the `xml:base`

[18]This type allows for combinations of text content and markup elements.

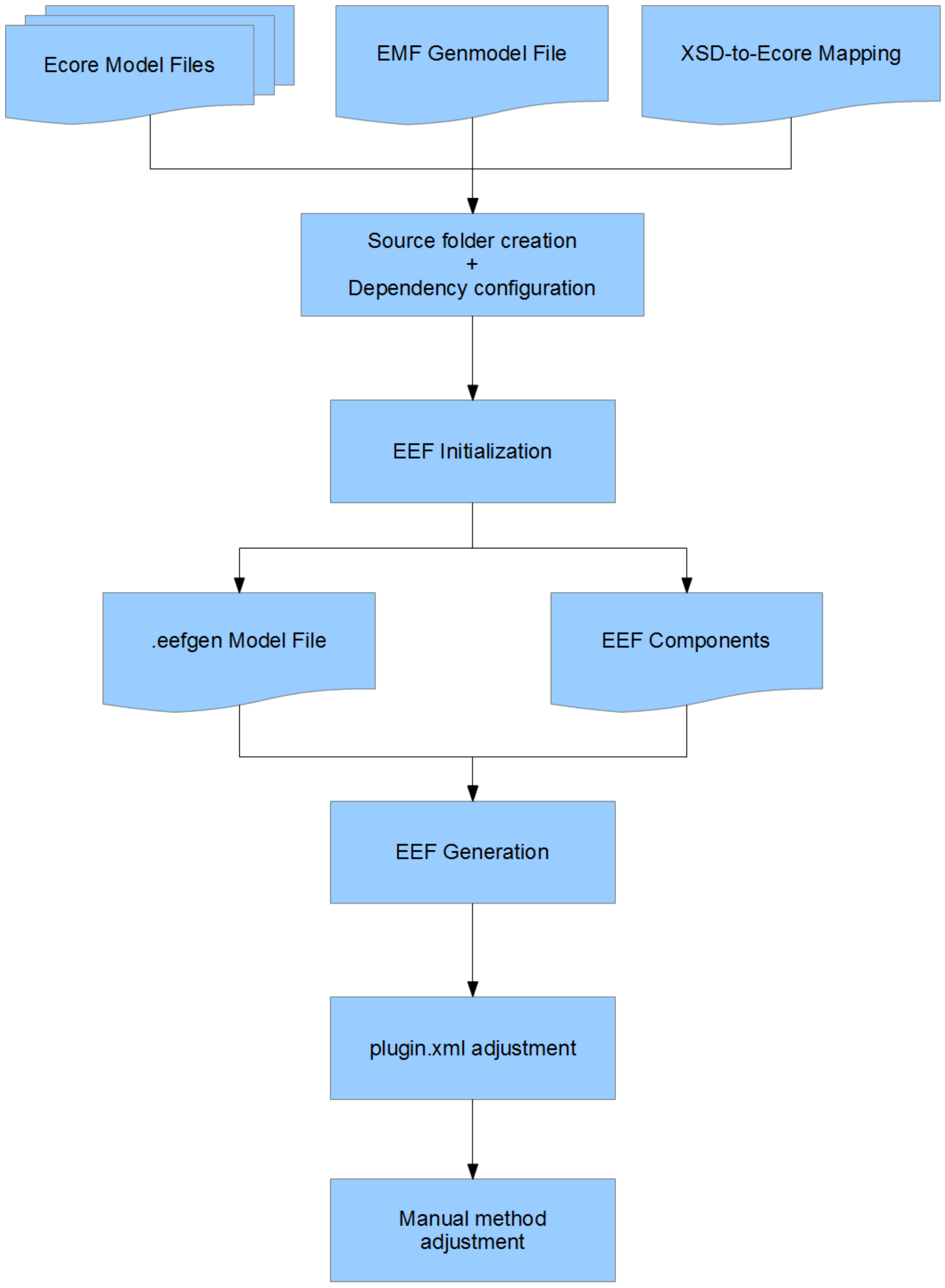


Figure 6.6: Generation steps EEF

attribute was omitted.

Analyzing this smaller subset of seven global element declarations, nine complex type definitions (including types extending/ restricting each other) and two simple type definitions led to the insight that repeatable XML Schema model groups (i.e., `xsd:sequence`s and `xsd:choice`s) were correctly handled by EMF, but EEF seemed to incorrectly transform these into widgets. EMF uses a mechanism called *FeatureMap* for such model groups with a `maxOccurs` attribute value greater than 1. This causes the element containing the respective repeatable model group to get an extra `group` property in the Ecore model representation. EEF's generation step interprets this as a property to be converted into a widget and does not take the components within the group (i.e., element declarations) into account. Adding hints in the input XSD file prevents EMF from creating *FeatureMap*s for repeatable XSD model groups in the first place and thus provided proper input for EEF's generation.

Based on this the EEF generation was applied again and the result was discussed with ISCUE's management and staff responsible for *PrIML*. In addition to the general requirements for a *PrIML* UI the following aspects were brought into focus specifically for the Eclipse case:

- behavior of manually applied code changes when re-generations are applied,
- behavior and possible functional limitations of property views and tree views in Eclipse's workbench,
- introduction of a Rich Text editor for mixed types (especially `format:formattedText`),
- changing the textual labels of elements in tree and table view,
- representing sub-tables with more than one column for detailed view,
- possibilities to enable widget's resizability (not enabled by default),
- directly creating mandatory sub-elements,
- methods for integrating more than one XML element level on one form and hence deliberate limitations of the tree view depth,
- possibilities of reflecting file management (i.e., modularization) in the UI.

The ideas behind these aspects and their possibilities of implementation are described below.

6.3.3 Customization Steps

After completing EEF's generation and the gathering of exemplary adjustment necessities (see list above) the processes of implementing and possible problems encountered during these attempts are described in this chapter.

Code protection when re-generating The fact that EMF uses protected source code zones marked by JavaDoc `@generated NOT` tags, leads to the issue of the behavior of the generation processes when encountering methods, that have to be overwritten, but are marked as protected. A case where this comes into focus, is, when the order of widgets on a form is adjusted by changing the order of `addStep` method invocations. Changing the underlying model (be it the XSD or the Ecore model) leads to a mandatory re-generation of the *EditionPartForm Java classes in order to appropriately reflect all of the model's attributes. The generation then overwrites the protected method but creates a file with the same local name and the file extension .lost in the same source folder. It contains all source code fragments that were marked in order to be protected but overwritten anyway. Restoring otherwise lost content is thus possible. For a simple order changing use case it is worth considering whether protecting the method is useful at all, because in any case a re-ordering after a new generation run becomes necessary.

Property view behavior and possible pin-on commands Eclipse treats the property view as one view that can either be visible, minimized or invisible. Similarly behaving views are the Outline, Error Log and other generic views used in many contexts. By default, there can be one of each of these views in the Eclipse workbench. Since EEF expands the usual key-value character of property views to fully-featured forms, this one-at-a-time constraint limits the user. Monitoring the details of two entities at the same time is not directly possible. It is however possible to pin-on a property view, i.e., fixing its content, regardless of the possibly changing tree view selection. This feature is not fully functional in EEF, since not all event handlers, especially for handling attribute changes or updates, are reflected appropriately. This bug is reported but could not be resolved by the time of this thesis.

Rich text editor The documentation and change log of EEF states, that it is possible from version 0.9.0 onwards, to use a rich text editor plugin from another Eclipse project, the Eclipse Process Framework (EPF). However, none of the EEF-related documentation pages and tutorials provided any information on how to replace a usual text input widget by this sort of rich text widget. The EEF components model editor that is enabling the user to change widget representations for object types and attributes, does not offer the rich text widget in question. Programmatically plugging it into an EEF property view form proved to be not working.

Textual labels for elements The default textual label for an element is a concatenation of the element's type name and one of its attribute values if present. This does provide a starting point but is in most cases dissatisfactory. It can be changed by overriding the `getText()` method for the element's type in its ItemProvider class. Checks for null values are important in order to prevent NullPointerExceptions to occur at run time. See the screen casts for an example demonstration of such a text label customization.

ReferencesTable enhancements Tables for multi-valued containment EAttributes are by default represented as AdvancedTableCompositions in EEF. They are populated by ContentProviders with a textual label consisting of a string. The respective labels can be customized according to the pieces information that shall be representing an item in the tree view and in potential table views on EEF forms (see paragraph before). This is helpful for similarly structured sorts of items but gets uncomfortable in cases where the label strings become inconsistent in terms of length and thus comparability. So the decision was to define more than one column on table views.

The example for which this was tested was the requirement group form. In a first step it was possible to add an instance of `org.eclipse.swt.widgets.Table` to the form

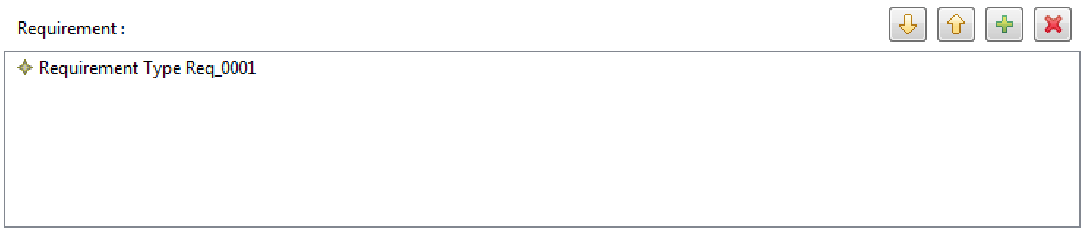
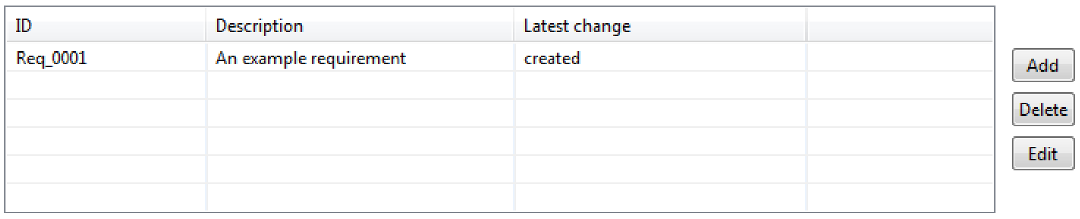


Figure 6.7: EEF widget AdvancedTableComposition for requirements as children of requirement groups



ID	Description	Latest change	
Req_0001	An example requirement	created	

Figure 6.8: EEF widget TableComposition for requirements as children of requirement groups

and set three columns with appropriate headers. This table was complemented by a button which got a `MouseUp` event handler iterating over the table items in the original requirement table. For each table item the `getData()` method provided the underlying item data which could then be used for the population of the manually added table. Displaying the ID, description and history entries in one table row was thus possible. But cloning the table in such a manner is not feasible for a functional UI, so other widget representations were attempted. Contrary to the nomenclature, the TableComposition widget offered in fact more advanced entry points for programmatically adding columns and populating them with further data attributes of the underlying instance object.

So, changing the widget representation for requirements on requirement group forms in the EEF .components model to TableComposition and invoking a regeneration was approached. See a comparison of the widgets AdvancedTableComposition (figure 6.7) and TableComposition (figure 6.8). Both represent the same underlying requirement (i.e., with the same attribute-value set) that is a child of a requirement group.

See the screen casts for a demonstration video of changing an AdvancedTableComposition to a TableComposition.

Widget sizes The layout of widgets on EEF-generated forms and the layout of the latter as a whole can be influenced on source code level. In EEF's `src-gen` folder reside *[type name]PropertiesEditionPartForm.java* files for each respective type. Every attribute and sub-element of such a type has methods in such a class for creation, initialization, etc. The sort of widget that is demanded to be resized in order to better exploit the existing screen space is the EEF *AdvancedTableComposition*. It is used for presenting a list of equally-typed sub-elements on the parent's form view. In the

PrIML subset starting from the requirement group level this is used for e.g. the list of requirements as the children of a requirement group. Since this table shows only five items by default and offers a scroll bar for more than five items, this is taken as the precedence case for changing a widget's size. (The same necessity applies to the sibling table view presenting sub-requirement groups.)

In the `createRequirementTableComposition` method exists the default layout configuration for tables of this kind. The constant `GridData.FILL_HORIZONTAL` is set as the basic layout information and the `horizontalSpan` property is set to value 3. It is worth noting that this layout configuration applies to the grid that holds the table, not the table itself. Changing layout parameters is possible by addressing the `getTable` method on the ReferencesTable object and invoking `setLayoutData` with the respective `FormData` instance as the argument. See listing 6.6 for the example code.

```
protected Composite createRequirementTableComposition(
    FormToolkit widgetFactory, Composite parent) {

  // ...

  GridData requirementData = new GridData(GridData.
      FILL_HORIZONTAL);
  requirementData.horizontalSpan = 3;

  // start of manually added layout code
  FormData requirementTableLayout = new FormData();
  requirementTableLayout.height = 700;
  requirementTableLayout.width = 700;
  this.requirement.getTable().setLayoutData(
      requirementTableLayout);
  // end of manually added layout code

  this.requirement.setLayoutData(requirementData);

  // ...
}
```

Listing 6.6: Layout configuration for EEF references table

Explicitly setting height and width properties on the table has the desired effect but brings the side effect of hiding the button group for adding, moving and deleting items from the table. Note also that this size changing method applies to AdvancedTableComposition widgets; achieving a similar effect on TableComposition widgets seemed not directly possible.

Automatic sub-element expansion When creating new elements in the EMF (or EEF) editor, there exists no automatic creation of mandatory sub-elements. This allows for instance data that does not conform to the underlying schema to be saved. It is impossible to introduce mechanisms that provide valid structures in *every* case. Consider a mandatory `xsd:choice` group with several child elements of which more than one has a `minOccurs` >0. The only way of ensuring an always-valid sub-element creation would be to randomly choose one of those elements, of which one has to occur. This can easily lead to confusion if the user does not know about the random character of the expansion or to rejection if the behavior is comprehensible but provides often not-needed results. It is however possible and useful to automatically expand mandatory sub-elements in *unambiguous* cases. An example is the direct creation of a history element when creating a new requirement. This history element can get default values as the current date, version number 1 and the changed value of "created". See listing 6.7 for the source code of this example.

Integration and pulling up element information The list of elements that shall be displayed as children of an element in question contains all child elements of complex type by default. Changing this assertion is possible in the `getChildrenFeatures()` method of the respective *[type name]ItemProvider* class. Eventually this method is invoked for the super-type and specific children features for the type are added. Manipulating these assertions causes a limited children display (when features are removed) or an enhanced display when others are added. Ideally, the right-click context menu offering to create *New Child*ren is held consistent to that: the method `collectNewChildDescriptors()` is responsible for assertions about possible elements that can be created as new children of the respective element. Such changes can and should be asserted on the .genmodel level in order to have the (re-)generation process handle all necessary source code changes correctly. The attributes *Children* (influencing whether the elements in question shall be displayed as children nodes in the tree) and *Create child* (influencing whether the "New child" context menu shall offer an item for the element in question) are the relevant configuration flags for this case. In one of the screen cast videos there is a demonstration for limiting the tree display for elements that are already represented via table views.

Whenever there are cases where more than one XML element level has to be skipped from the tree, this indermediate level has to be taken care of when customizing the tree. This can either be accomplished by manually overriding methods in the Java classes for the respective elements or in case a schema change is acceptable, this might ease the efforts. One example from the *PrIML* XSDs is the element `references` on a requirement. There has to exist exactly one such element on every requirement and it can contain the elements `realize`, `refine`, `copy` and `link_to`. These in turn can each contain arbitrarily many sub-elements. Integrating this kind of two-level element hierarchy cannot be handled by model adjustments, EMF does not offer a feature for hiding wrapper elements[19] in the UI and only applying the wrapping for serialization. Elaborate XSD changes would have to be undertaken in order to form a structure that a) reflects the domain reality of refinements, realizations, etc. appropriately and b) is automatically transformed into comfortable UI representations by EEF.

An attempt to integrate the `references` of requirements directly on the require-

[19]This is a common practice in XML structures for aggregating elements of the same kind in a (often pluralized) form of wrapper element.

ment's form has been made. Not all constraints that the original XSD constructs make have been realized in the reorganization but it however forms an example how to rebuild schema parts in order to influence EEF's form generation. In this changed schema (its XSD and the generated editor plugin is contained on the thesis CD-ROM for more detailed insights) the sub-element `references` is replaced by an `xsd:choice` that has to occur exactly once and the elements within the `xsd:choice` are `modification_request`, `requirement`, `open_point`, etc. These have the same sub-element and attribute structure as before, only that the kind of reference (`refine`, `realize`, `copy` or `link_to`) is now reflected in a `refType` attribute. That does not allow for constraining the sometimes applicable `state` attribute depending on the reference type. This attribute hence is settable in all cases now, only its use is not "required" anymore. This is the basic validation loss that comes with this schema change, among the necessity to change the XSLT scripts that expect the structure to be like the original schema states it. Generating an EEF editor based on this schema it is possible to see and edit all references that the requirement in question makes on the requirement's form itself. Conforming to the multi-column table customization description above an even more informative view can be achieved where the type of reference is directly visible etc. It has to be emphasized that this attempt is only one possible way of introducing a schema change. Another way would be to take the kind of reference as the entry point under each requirement and assert the references target type in an attribute.

File management Being able to separate instance data on more than one file is one of the key *PrIML* advantages and necessities. Efficient versioning, collaboration and the logical modularization of data components are the rationale for such separation. Integrating resource-specific aspects into the UI such as file names and the tree view population over more than one file is thus a key aspects to be evaluated.

EMF offers the possibility to edit more than one file in the same EMF editor. Loading new resources into the editor can be done by right-clicking in the tree view and selecting *Load Resource* from the context menu. Every resource is represented by a node under the (virtual) document root node in the displayed tree. This does not yet reflect the XInclude mechanisms used in *PrIML* but such functionality might be added programmatically.

```
// method getHistory() in class PrimlElementBasetypeImpl
public EList getHistory() {
  // in case the history property is not already set ...
  if (history == null) {
    // ... one is created
    history = new EObjectContainmentEList(HistoryType.
       class, this, SubsetPackage.
       PRIML_ELEMENT_BASETYPE__HISTORY);

    // a new HistoryType is being instantiated,
    HistoryTypeImpl default_history = new
       HistoryTypeImpl();

    // its version property is set to value 1
    default_history.setVersion(1);

    // its changed value is set to "created"
    default_history.setChange("created");

    // a new Calendar instance provides the current date
        and time information
    Calendar date = Calendar.getInstance();

    // the date is set conforming to the format YYYY-MM-
       DD; leading zeros are eventually added for month
       and day
    default_history.setDate(date.get(Calendar.YEAR) + "-
       " + (date.get(Calendar.MONTH) < 10 ? "0" : "") +
       date.get(Calendar.MONTH) + "-" + (date.get(
       Calendar.DAY_OF_MONTH) < 10 ? "0" : "") + date.get
       (Calendar.DAY_OF_MONTH));

    // for editor the default value is set to the empty
       string
    default_history.setEditor("");

    // the entry is added to the history reference list
    history.add(default_history);
    }
    return history;
}
```

Listing 6.7: Manually added source code for history expansion on requirement addition

6.4 Comparison

After describing and explaining both solutions in the two chapters above, a comparison is drawn. This is done using the aspects and requirements listed in chaper 6.1 and in a more general manner afterwards. For both protoypes the extent, to which the respective requirement is implemented, is described. This comparison is based on the state that the prototypes have in the end of the thesis period.

6.4.1 Requirement-Based Evaluations

See table 6.2 for a tabular comparison complementing the textual comparion in this chapter.

"The UI is able to run on Windows, Linux and MacOS X" Both approaches follow a platform-independent paradigm. Web applications are dependent on the existence of a (modern[20]) web browser. All three operating systems Windows, Linux and MacOS X support such modern web browsers, thus the web application is able to run on all of these platforms. Differences between the browsers can occur, since every vendor implements features slightly different but the functionality for the application developed in this thesis shall be given.

Eclipse is a Java-based platform and hence can be run on any platform supporting a Java Virtual Machine (VM). All three operating systems offer Java support, so the requirement is met by Eclipse and thus by EMF and EEF as well.

Both solutions meet the requirement of being able to run on the three main operating systems for desktop devices. Since web applications can in most cases also be used on mobile devices, this can be considered an extra benefit that was not explicitly demanded. The perpetual evolution of web technology and the lack of reliable cross-browser feature support is a disadvantage in terms of web application use. Eclipse as an open-source IDE and its projects being under continuous developments as well is also not completely reliable and does not in all cases offer backwards compatibility.

So both approaches provide a platform-independent basis.

"The UI is generated as completely as possible; manual customizations can be applied to fit needs more exactly" The extent to which a UI and its processes can be automatically generated is one aspect of this requirement; the other one is the extensibility.

The web application is (apart from the removal of comments in order to enable the functionality) designed for a complete generation. Manual extensions can be applied but do not have to.

A similar case is EMF: the generation processes provide a fully-functional editor plugin that does not need any manual input. EEF is slightly more demanding when it

[20]The term of a modern web browser is not clearly definable. All current versions of the three main browsers Mozilla Firefox, Google Chrome and Internet Explorer to some extent can be considered to be "modern". Internet Explorer versions <= 7 are commonly perceived as critical browsers in terms of feature support.

comes to manual steps complementing its generations. Copy-pasting generated fragments in the respective destination folders and files and overriding specific Java methods belongs to these necessary manual steps.[21]

When it comes to extensibility, the web application can be further developed with functions such as drag&drop and introducing domain knowledge for cases like ID references of certain element types only. The amount of files, lines of code and abstraction layers is comprehensible. Both the generation methods and the resulting classes and files can be modified quite directly. EMF and EEF build on a far more verbose architecture than the web application. This leads to a high entry threshold but supplies the user with a highly modularized infrastructure, once he understands the components interacting with each other.

Summing up, both prototypes have a high generation potential, EEF lessens it slightly for the Eclipse approach. The learn-curve is steep for the EMF/EEF frameworks, their advantages lie in the long-run when such a stable architecture allows for reliable components to apply extensions to. The light-weight web solution allows for a better maintainability in the beginning but demands for a more extensive amount of hand-coding features when they become necessary.

"Schema changes are reflected in the UI after a regeneration" This requirement is mostly relevant for the distinction between generation time and run time. In both approaches schema changes are not reflected at run time but require a regeneration.

Invoking such a regeneration for the web application is accomplished by executing the command-line Python script again. EMF has different schema layers that can change. The input XSD(s) can change, the Ecore model derived from the XSD(s), the .genmodel, the .xsd2ecore and EEF's two model files. EMF and EEF allow for partial (re-)generation, i.e., regenerating only the parts that require updates. This is comfortable when changes and their consequences are completely comprehensible but often even small schema changes have consequences in several classes not directly visible from a developer's point of view. Making general decisions on how to introduce schema and model changes and how to make them explicit for (later) comprehensibility can be helpful in order to consistently preserve manual changes. Including generated source code into versioning systems is usually not recommended but when generated and manually written source code intermix, it might become useful.

"Manual customizations are protected from being overwritten by a regeneration" In the Python generation processes producing the web application no such feature as protected zones or merging of source code files is available, so manually applied changes are by default overwritten if not saved in other folders or files. EMF and EEF use protected source code zones and thus manual changes can be preserved from being overwritten by new generations. This requirement is thus better met by EMF/EEF than by the self-developed Python methods.

[21]These steps are automatable but EEF does not itself provide mechanisms to do so. Ant build scripts for supporting EEF's generations are discussed but not available during the time of research.

Requirement	**Web Application**	**EMF and EEF**
The UI is able to run on Windows, Linux and MacOS X	yes	yes
The UI is generated as completely as possible; manual customizations can be applied to fit needs more exactly	yes	yes
Schema changes are reflected in the UI after a regeneration	yes	yes
Manual customizations are protected from being overwritten by a regeneration	no	yes
The user does not need to enter XML markup	yes	yes
The user can combine XML data entry and UI-based data entry	no	yes
The UI suggests components and/or values where applicable	partly	partly
The UI allows user to deep-copy (and manually adjust) existing components	no	yes
The UI provides undo and redo functionality	no	yes
Approaches of direct manipulation (e.g., drag&drop) are enabled where applicable	no	yes
Element values of mixed content type (especially `format:formattedText` and its sub-types) can be entered with an appropriate rich text editor widget	no	no
PrIML's XSLT processes can be invoked from the UI	no	yes
	yes: 4 no: 7 partly: 1	yes: 10 no: 1 partly: 1

Table 6.2: Comparison of requirements

"The user does not need to enter XML markup" Developing a GUI based on an XML Schema vocabulary has in *PrIML*'s case the main reason of introducing more comfortable interaction methods than entering XML markup directly. Abstracting from the serialization that characterizes XML the UI is perceived as the intermediate interaction layer between user and data. In both solutions the user does not need to enter XML markup in order to edit the XML instances in question. Hence, both solutions do not demand XML knowledge on end user side.

"The user can combine XML data entry and UI-based data entry" Releasing users from entering XML markup is in most cases a comfortable and desired aspect, but some users might want to combine XML markup entry with UI support aspects. This makes two views on the same data necessary: one UI view and one XML or plain text view. These have to be updated whenever data changes occur and kept in sync with each other. Eclipse has its concepts of perspectives containing several views. This allows for exactly this kind of manifold editing as just described. Opening an XML instance file with the generated editor and simultaneously viewing it with an XML view is possible. The Eclipse platform listens to change events and notifies all listeners of data changes in order to give these listeners the chance to update accordingly. The web application puts out the XML serialization form of the current elements into a textarea element. This is not automatically updated and changing the content of it does not lead to a respective data update. Enabling it would directly be coupled to the ability to load existing instance data into the UI and re-rendering the views. Since this is not possible in the state of the prototype, the combination of UI data entry and markup edition is only possible in the EMF plugin solution.

"The UI suggests components and/or values where applicable" Value suggestions and widgets that only allow for the entry of valid values fall into this category of user support. Both EMF and the web application provide such aspects via the choice of drop-down menus when encountering enumeration simple types. Choosing one of the suggested values ensures valid data entry. Automatically setting eventual default or fixed values is another such aspect that is implemented in both solutions. The direct expansion of required attributes and sub-elements is a feature that is only partly implemented in both solutions. The web application takes `minOccurs` values into account when rendering sub-elements of elements and pre-fills inputs with default and/or fixed values. EMF does directly fill in default and fixed values but does not pre-render required sub-elements.

Summing up, both approaches implement some potential features, but none of them offers completely satisfactory mechanisms that exhaust the suggestion potential.

"The UI allows user to deep-copy [...] existing components" When it comes to repetitive data entry where only details in different sub-elements differ, possibilities to easily copy and adjust existing elements can be helpful. Eclipe's tree view natively supports copying of either single tree nodes or whole tree branches. EMF triggers copy actions when such copying (and deletions analogously) is observed. The web solution does not support direct cloning of elements.

"The UI provides undo and redo functionality" Providing an undo stack that tracks the last changes applied to the data is a common practice in many kinds of applications. Eclipse as an IDE framework offers undo and redo functionalities by default and plugins in the framework such as the generated editors benefit from them. Browser applications do not build upon such an infrastructure and have to introduce interaction stacks themselves if demanded. The web prototype in this thesis does not contain an undo stack since it would require complete loading, creating, updating and deleting mechanisms. These are not part of the prototype's state and thus an undo mechanism is not implemented.

"Approaches of direct manipulation [...] are enabled where applicable" In terms of direct manipulation aspects such as drag&drop the situation is similar to the undo features described above. Applications that exploit an existing infrastructure like the Eclipse framework that provide many low-level functions do not need to take care of these functions' implementation. This holds true for dragging and dropping tree nodes in Eclipse as well. Drag&drop is also common in browser applications for some time but is subject to manual development rather than just using anyway existing mechanisms. jQuery UI for example offers library constructs for drag&drop interaction patterns but this has not been introduced into the web-based approach in this thesis.

"Element values of mixed content type [...] can be entered with an appropriate rich-text editor widget" One of the powerful features in *PrIML* is the usage of mixed complex types for most textual elements. The type `format:formattedText` and its derived types allow for the intermixing of text content and sub-elements that mark up the text. These elements contain constructs for emphasis, tables, links and other structuring aspects. Reflecting such mixed-typed elements is by default handled straightforwardly by both EMF and the web application. The special character of marking up is not reflected since it requires the domain knowledge that wrapping text content in an `emph` element would ideally lead to typographically emphasizing it. Being able to represent mixed-typed elements with a rich text editor widget was hence a formulated requirement. The EEF documentation does mention a rich text editor widget that can be included in an EEF property view but attempts to de facto using it in the generated plugin, failed. Neither asserting it in the EEF .components nor programmatically adding it to a generated form view could be achieved. Rich text editor plugins exist for web applications, some only usable for HTML markup, some customizable to fit specific needs. This thesis protoype web application does not introduce such a widget.

None of the solutions can currently provide a comfortable rich text editor for mixed-typed elements.

"PrIML's XSLT processes can be invoked from the UI" The character of *PrIML* as a framework bundling XML Schema definitions and XSLT transformation scripts makes a direct invocation of these scripts a useful feature of the UI in question. Ant is used for *PrIML*'s transformations complemented by directory clean-up etc. Since Ant is available as an Eclipse view, the integration of the EMF/EEF editor and *PrIML*'s XSLT scripts is directly possible. In web browsers, neither Ant support nor XSLT 2.0 are available and thus a seamless integration cannot be achieved.

Specific EMF/EEF customizations In addition to the general requirements and the explanations on their implementation status in both prototypes, the specific customization demands for EMF and EEF are briefly summed up. These have been described in chapter 6.3.2.

Not all of the customizations that have been identified as useful or even required, could be realized. The non-successful aspects are mostly caused by the verbose architecture of EMF and EEF and especially the lack of detailed EEF documentation and basic literature. This led to extensive trial-and-error experimenting that did not in all cases produce satisfactory results. The rich text editor plugin usage is such a case where the desired solution could not be achieved. This can be perceived as a blocking aspect because *PrIML*'s textual elements mainly build upon mixed types that represent text content with potentially formating markup. The subset of *PrIML* elements does deliberately exclude this element facet and reduces text content to `xsd:strings`. For a productive usage there has to either be a solution that implements a sort of rich text editor or at least a widget type that supports the entry of markup tags.[22]

An aspect that was implementable is the change from AdvancedTableComposition to TableComposition widgets in order to display a more detailed list of sub-elements on forms. The features of a table are thus exploited in a more efficient way. It bundles anyway existing information in a more structured way than just displaying a flat textual label for each item. A demonstration that can be used as the basis for reorganizations in EEF is the automatic sub-element creation as shown for history entries of requirements. The issues related to element level integration and tree view clipping are more complex and are ideally approached specifically. Schema changes seem to be an often appropriate solution in order to avoid cumbersome multi-level Java source code changes.

Essential for enhancing EEF editors to really usable editors is the ability to detach elements' representation from their underlying XML structure. Concepts like wrapper elements that are useful from an XML serialization point of view might be perceived too verbose or even confusing when it comes to a tree and property view representation.

The general impression is, that a customization requires the programatic edition of Java code – not all aspects are realizable via model assertions. Whenever such Java changes are in question, the different source code packages have to be checked for potential interdependencies. Overriding a method in one class led in some cases to incomprehensible `NullPointerExceptions` at run time.

6.4.2 Overall Comparison

Throughout the complete prototyping phase there proved to be two main differences between the approaches taken.

The first is, that whenever using an existing heavy-weight framework or framework combination, the potentials of reusing many low-level infrastructure features grow. This holds true for the complete Eclipse framework that EMF and EEF directly exploit and provide further to the concrete editor plugins based on their processes. All aspects mentioned above like file handling, window management, change notifications and undo functionality are examples for infrastructural advantages of a system like Eclipse. However, EMF and EEF add their own generic classes for ItemProviders, editors, forms, etc. These are matured constructs interacting with each other in well-defined ways and ready to be used. The extent to which such architecture exists when using Eclipse, it has to be designed and implemented nearly from scratch when attempting to prototype a similar editor using browser and web standards. The differences between heavy-weight and light-weight approaches drastically show when directly contrasted with each other as practiced above.

The second difference realized, that is associated with the heavy-weight/light-weight aspect, is the importance of an understanding of the processes that are used. [Hun00]

[22]It would be possible to replace a default text input by a StyledText widget and attach markup buttons to it. These would wrap the text selection made with respective markup tags. Such an approach would contradict the goal of not having to handle XML markup in the UI.

states that source code wizards are able to produce large amounts of program code in nearly no time. What is pointed out as well, is, that using such wizards is only advisable when the developer understands their processes and especially their output (cf. [Hun00], pp. 198f.). This applies to EMF and EEF as well. At least when it comes to either misbehavior of the generated editor or the case that customizations are demanded.

As a consequence, the expectation of having two prototypes that each demonstrates different software generation aspects, has proved to be true.

Similar to both solutions is, that generating an editor with a model-based concept by default leads to a tree-like view and form-based paradigm. There are mechanisms of transforming XSD types into widgets that seem straight-forward and fit the nature of the underlying types. Nevertheless, any such model-based approach focuses on the model, *not* primarily on the user. It is essential for the complete development and design process whether the premise is somewhat similar to “The model contains the canonical structure to be used” or “The user shall be able to edit the information he is confronted with”. Developing model-to-UI mechanisms nearly from scratch directly stresses the fact that model-based methods always assume the role of inspecting what the existing formal description provides rather than what is usable for the end user. Both solutions, web application and EMF/EEF, share this kind of perspective. Following the input XML structure is surely straight-forward but not in all cases the most intuitive way possible. UIDLs attempt to take the user’s task perspective into account by expressing potential tasks in a model. This is one step towards a more user-centered way of developing but such task models have their fix structural constructs as well.

The tenor that is often formulated in terms of MBUID results and potentials is, that most UI components can be generated since the methods to express models and the mechanisms to transform these into source code exist. But the argument against MBUID approaches is, that they formalize essential aspects of interaction too drastically. This aspect can be corroborated from the practical solution perspective. The two prototypes applied algorithmic structures to formal descriptions to generate a UI. The output does reflect the underlying entities and properties but the algorithmic paradigm is quite obviously apparent. EMF is designed to form the basis for a further development using its API and abstraction layers. When the result is not claimed to be a directly usable (in sense of *usability*) interface, the potential of EMF as a starting point for UI development is very high. EEF attempts to extend this even further but is still based on formalizing mechanisms. Similar overall conclusions apply to the web application approach. Its basic generation potential is high as well, but from the usability point of view such a UI will in most cases not be the end of the development efforts.

6.4.3 Recommendations for ISCUE

The goal of giving a specific framework recommendation to ISCUE, that was formulated in the introduction, can be met by pointing out EMF’s advantages. It is a widely-adopted approach and framework, at least the EMF part. EMF and EEF provide model-to-model and model-to-code transformations that lead to a functional and powerful editor plugin. The Java code generated out of the Ecore and other models does itself introduce several levels of abstraction, manifested in providers, adapters, interfaces and implemented classes. Attempts to customize auch a complex architecture requires much more (Java) programming knowledge than expected. EMF as the much more mature framework in the Eclipse Modeling context defines its role as a basis for manual adjustments and customizations. EEF goes one step further by providing constructs that allow for more elaborate form building. Since EEF is not as mature as EMF, the documentation is in many cases not as reliable and verbose. Systematical introduction literature does exist for EMF, for EEF the wiki and the Eclipse forums are often the only information sources. The demonstrated customization examples in EEF are not all ready for productional implementation. The understanding of a software generator's output is absolutely essential for evaluating, customizing and debugging. In order to produce customized EMF/EEF editors that run stable, a solid knowledge of the Java programming language, the Ecore meta-model and the general Eclipse plugin development architecture is indispensable. The used EMF book [Ste09] provides a solid basis for understanding EMF, EEF in turn demands for intensive trial-and-error development due to its (still) lacking basic literature. However, Eclipse offers a lot of basic functionality that can directly be re-used: file handling, window management, undo and redo mechanisms and the deep Java integration. These are aspects that a manually-developed prototype as the web application has to be equipped with, in order to be really competitive. Table 6.2 shows the comparison; EMF/EEF can meet ten of the requirements, the web application only four.

Using a web-based application as the one achieved in this thesis is not feasible for meeting ISCUE's requirements. Stability, reliable mechanisms for deploying and an ideally seamless integration into the existing tool chain cannot be ensured with such a web application. Features that Eclipse (and thus EMF/EEF) provides by default, have to be considered explicitly in the browser context. File input and output are hardly achievable on client-side only. Undo stacks and the flexible window management infrastructure are prominent features that are crucial to data entry applications but have to be added by hand.

Summing up, the recommendation for a *PrIML* user interface that is supported by generative methods, is, to use the Eclipse frameworks EMF and EEF. The condition for a successful usage is to accept a much higher amount of effort than originally expected. An easily customizable GUI builder as for example Visual Studio, Qt and others provide it, does not exist and demands programatically changing and extending Java source code. The XSD model definitions provide enough information for generating a UI, that could be proven, but in order to be really usable and useful in comparison to just using the Eclipse XML perspective, it needs time and in both EMF's and EEF's contexts the willingness to accept high degrees of abstraction. In addition to that, manually changing and extending generated source code demands the awareness that a change in one class method may lead to errors in several others – even more drastically when customizing self-developed or self-generated source code.

7 Conclusions and Lessons Learned

The working hypothesis that led to this thesis' topic and goal was, that a model definition (in XML Schema) provides sufficient information for automatically generating a UI that enables users to CRUD data conforming to the model. Type definitions, relations and constraints should be the basis for ideally a complete or nearly complete generation of a UI that releases the end user from entering XML markup.

Evaluating MBUID methods on a practical use case and to pragmatically generate a UI application was the goal this thesis has set up. Instead of only gathering models, concepts and theories of MBUID approaches the *PrIML* vocabulary and the concrete necessity of a UI were the starting points of the considerations made.

First research led to the MBUID research field, that develops models, approaches and tools for over 20 years. Its existence emphasized the significance of concepts such as automating UI development with models and software generators. Formalizing UI components, abstracting out recurring tasks and patterns were notions that frequently occurred in literature and practical tool development reports. Goals and motivations were in most cases at least indirectly bound to the idea of increasing efficiency and deduplicating efforts. But when it comes to UIDLs and MBUID frameworks, an often-neglected aspect is the concrete transformation methods for code generation. Rendering the formal UI descriptions into functional user interfaces is usually left out and there are hardly any out-of-the-box frameworks building on top of UIDL models. Literature and reports on MBUID in general often come to the conclusion that the MBUID paradigm is not broadly accepted and adapted due to its high degree of formalization. Since usability and highly specific interfaces are not easily implementable by MBUID methods, manual design and development is often preferred in the first place. This common sense in MBUID criticism was used as the basis of the thesis' analysis – whether the limits of model-based UIs still hold true if a specific XML Schema is used as the demonstration case.

The working hypothesis stated in the introduction chapter has been proven since XML Schema definitions provide enough information for UI generation. The Eclipse plugin that EMF and EEF in combination generate and the browser-based application are examples for the ability of such UI generation out of XML Schema. The user does not need to enter XML markup code to CRUD schema-conforming instances. Forms with widgets are automatically generated and included as appropriate. Since design and development of UIs is highly user-specific and even slightly different interaction element choices can lead to drastic influences on usability, the customization possibilities are an essential necessity. Usability is an ever-present layer when designing and developing applications. The prototypes that have been developed in the thesis are examples for UI applications that do not focus on usability primarily. This became obvious when the (intermediate) results in EMF/EEF were put up for discussion with ISCUE's management and staff. Two aspects characterized the assessment of the UI: the process was claimed to be automated since the information necessary seemed to exist in the XML Schema already but at the same time customization and change requests were brought up. So the seemingly contrary demands for automation/formalization on the one hand and customization/specialization on the other hand were appreciable in the way the UI was perceived. In order to cover both aspects in the Eclipse development branch, the generations were used and some customization requests were implemented exemplarily.

As a light-weight environment for application development the browser-based solution demonstrates the potential of software components generation. Without introducing model-to-model transformations and without heavy abstraction layers it stresses the concept of generating source code in order to achieve consistency and still being able to directly comprehend cause and effect of these generative methods. Creating an architecture for the target application, developing the reference implementation for a defined XSD subset and implementing the generator methods on this basis were the steps taken in this approach. It directly demonstrates the needs for consistency over all UI components (models, views, controllers, XML bindings) and the challenges of MBUID paradigms such as appropriate widget choices. The desired effects of this sort of generation were the necessity to really experience what it takes to set up generators that interact with each other. Generating an application with self-developed processes demands for a continuous reflection about all kinds of application components and their communication and interdependencies. In some parts the development of the web application appeared to be a reverse engineering of the EMF and EEF paradigms, but whenever such perception arose, the awareness of software and UI generation even grew. The fact that model-based approaches often go back to similar mechanisms was demonstrated in such cases.

Methodically, the decision to compare two different prototypes instead of focusing on a single one, has in some cases led to trade-offs concerning the depth of research and implementation. Keeping the balance in terms of time and effort for both approaches was an additional challenge, complementing the anyway complex process of understanding, applying and evaluating model-based development concepts. However, being able to compare two ways of generating UIs out of XML Schema definitions (using a framework on the one hand and manually developing generators on the other hand) did provide benefits that made it worth the effort. The ability to reveal similarities in both approaches and the differentiation when it comes to the abstraction stack increased learning effects.

An essential lesson learned is, that generative methods can comfortably handle repetitive parts of development. Setting up scripts that automatically create class skeletons and consistent template bodies is a development pattern that can anyway be adapted – whether the development targets a 100% generation ratio or not. Generating UI components is only one specific use case of a widely adapted paradigm of automating development processes. But as stated above in some chapter contexts, formalizing the methods of UI development tends to result in similar-appealing sorts of interaction paradigms. This is appropriate and suffices when the primary goal is to hide XML markup from the end user, as it was in this thesis. But even then contra-intuitive widget choices, screen layout and positioning aspects often lead to customization demands. The subjectively perceived support that a UI provides the end user with, cannot be left out of the development considerations. The target audience has to be involved in the design decisions made for the UI and hence its eventual generation processes.

Summing up, the thesis is making its contribution to ISCUE's efforts to develop a UI for the *PrIML* vocabulary. The original expectation, that such a generated UI would immediately increase productivity and usability has to be put into perspective. A quite definite recommendation can be given to use the Eclipse Modeling Framework in combination with the Extended Editing Framework. However, the willingness to accept imperfection especially when it comes to EEF and the necessity to customize Java source code and/or the input schema models is essential. A web-based application as developed parallel to the Eclipse approach is in this state not feasible. But forming a demonstration and proof-of-concept case for moving state and software architecture to the client-side it is highly valuable for the understanding of web application design and generator methods producing such applications.

Bibliography

[Apa12] Apache. Apache Ant. http://ant.apache.org/ (Retrieved on June 25, 2012), 2012.

[App11] Apple Corp. Mac OS X Human Interface Guidelines. http://developer.apple.com/library/mac/#documentation/UserExperience/Conceptual/AppleHIGuidelines/Intro/Intro.html (Retrieved on June 25, 2012), 2011.

[Bay12] Bayer, Michael. Mako : Templates for Python. http://www.makotemplates.org/ (Retrieved on June 25, 2012), 2012.

[Big12] Bigot, Peter A. PyXB : Python XML Schema Bindings. http://pyxb.sourceforge.net/ (Retrieved on June 25, 2012), 2012.

[Bro10] Brown, Alex. Document Schema Definition Languages (DSDL). http://dsdl.org/ (Retrieved on June 25, 2012), 2010.

[Cal03] Calvary, Gaëlle and Coutaz, Joëlle and Thevenin, David and Limbourg, Quentin and Bouillon, Laurent and Vanderdonckt, Jean. A unifying reference framework for multi-target user interfaces. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.4512&rep=rep1&type=pdf (Retrieved on June 25, 2012), 2003.

[Cat12] Catlin, Hampton. SASS : Syntactically Awesome Stylesheets. http://sass-lang.com/ (Retrieved on June 25, 2012), 2012.

[Chr93] Christias, Panagiotis. UNIX man pages: diff(). http://unixhelp.ed.ac.uk/CGI/man-cgi?diff (Retrieved on June 25, 2012), 1993.

[Cla08] Clark, James. Trang : multi-format schema converter based on RELAX NG. http://www.thaiopensource.com/relaxng/trang.html (Retrieved on June 25, 2012), 2008.

[Cof11] CoffeeScript. CoffeeScript. http://jashkenas.github.com/coffee-script/ (Retrieved on June 25, 2012), 2011.

[Cos12] Costello, Roger L. XML Schemas: Best Practices. http://www.xfront.com/GlobalVersusLocal.html (Retrieved on June 25, 2012), 2012.

[Cro02] Crockford, Douglas. Introducing JSON. http://www.json.org (Retrieved on June 25, 2012), 2002.

[Cza04] Czarnecki, Krzysztof and Eisenecker, Ulrich W. *Generative Programming : Methods, Techniques, and Applications.* 2004.

[Del07] Delacre, Jean-Pierre. A Comparative Analysis of Transformation Engines for User Interface Development. `https://lilab.isys.ucl.ac.be/BCHI/publications/2007/Delacre-MSc2007.pdf` (Retrieved on June 25, 2012), 2007.

[Doc12] DocumentCloud. Backbone.js. `http://documentcloud.github.com/backbone/` (Retrieved on June 25, 2012), 2012.

[Dox12] Doxygen. Doxygen. `http://www.stack.nl/~dimitri/doxygen/` (Retrieved on June 25, 2012), 2012.

[Ecl06a] Eclipse. Generating an EMF Model using XML Schema (XSD). `http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/tutorials/xlibmod/xlibmod.html` (Retrieved on June 25, 2012), 2006.

[Ecl06b] Eclipse. Package org.eclipse.emf.ecore (EMF JavaDoc). `http://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/package-summary.html` (Retrieved on June 25, 2012), 2006.

[Ecl12a] Eclipse. Acceleo. `http://www.eclipse.org/acceleo/` (Retrieved on June 25, 2012), 2012.

[Ecl12b] Eclipse. EEF/Tutorials/First Generation. `http://wiki.eclipse.org/EEF/Tutorials/First_Generation` (Retrieved on June 25, 2012), 2012.

[Ecl12c] Eclipse. Xpand / Xtend Reference. `http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.xpand.doc/help/ch01.html` (Retrieved on June 25, 2012), 2012.

[ECM12] ECMAScript. ECMAScript : the language of the web. `http://www.ecmascript.org/` (Retrieved on June 25, 2012), 2012.

[Goo12] Google Inc. Android Design. `http://developer.android.com/design/index.html` (Retrieved on June 25, 2012), 2012.

[Gos05] Gossman, John. Introduction to Model/View/ViewModel pattern for building WPF apps. 2005.

[Gri01] Griffiths, Tony and Barclay, Peter J. and Paton, Norman W. and McKirdy, Jo and Kennedy, Jessie B. and Gray, Philip D. and Cooper, Richard and Goble, Carole A. and da Silva, Paulo Pinheiro. Teallach: a model-based user interface development environment for object databases. *Interacting with Computers*, 14(1):31–68, 2001.

[Han12] Hansson, David Heinemeier. Ruby on Rails. `http://rubyonrails.org/` (Retrieved on June 25, 2012), 2012.

[Hun00] Hunt, Andrew and Thomas, David. *The pragmatic programmer : from journeyman to master.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[ISC11] ISCUE. ISCUE : Embedded Projects. `http://www.iscue.com/` (Retrieved on June 25, 2012), 2011.

[ISC12] ISCUE. Explanations by ISCUE staff concerning PrIML design and developments, 2012.

[Jav10] Java Swing. Java Swing. `http://www.javaswing.org/` (Retrieved on June 25, 2012), 2010.

[Jav12] Java. Project JAXB. `http://jaxb.java.net/` (Retrieved on June 25, 2012), 2012.

[jQu12a] jQuery. jQuery : write less, do more. `http://www.jquery.com/` (Retrieved on June 25, 2012), 2012.

[jQu12b] jQuery. jQuery Templates Plugin. `http://api.jquery.com/category/plugins/templates/` (Retrieved on June 25, 2012), 2012.

[jQu12c] jQuery. jQuery UI. `http://jqueryui.com/` (Retrieved on June 25, 2012), 2012.

[Jso12] Jsonix. Jsonix. `http://confluence.highsource.org/display/JSNX/Jsonix` (Retrieved on June 25, 2012), 2012.

[Kay09] Kay, Michael. Saxon B. `http://sourceforge.net/projects/saxon/files/Saxon-B/` (Retrieved on June 25, 2012), 2009.

[Kla06] Klar, Michael and Klar, Susanne. *Einfach generieren : Generative Programmierung verständlich und praxisnah.* Hanser, München, 2006.

[KXF12] KXForms. KXForms. `http://www.lst.de/~cs/kode/kxforms.html` (Retrieved on June 25, 2012), 2012.

[Leh05] Lehmann, Christian. Einführung in modellbasierte XML-Sprachen für Benutzerschnittstellen : User Interface Engineering für mobile und web-basierte Anwendungen. `http://ebus.informatik.uni-leipzig.de/www/media/lehre/uiseminar05/ausarbeitung-lehmann.pdf` (Retrieved on June 25, 2012), 2005.

[Lid12] Liddell, Henry George and Scott, Robert. Meta - Greek word study tool. `http://www.perseus.tufts.edu/hopper/morph?l=meta%2F&la=greek#lexicon` (Retrieved on June 25, 2012), 2012.

[Mac11a] MacCaw, Alex. Asynchronous UIs - the future of web user interfaces. `http://alexmaccaw.co.uk/posts/async_ui` (Retrieved on June 25, 2012), 2011.

[Mac11b] MacCaw, Alex. *JavaScript Web Applications.* 2011.

[Mac12] MacCaw, Alex. Spine : build awesome JavaScript MVC applications. `http://spinejs.com/` (Retrieved on June 25, 2012), 2012.

[Mei11a] Meixner, Gerrit. Modellbasierte Entwicklung von Benutzungsschnittstellen. *Informatik-Spektrum*, 34(4):400–404, #aug# 2011.

[Mei11b] Meixner, Gerrit and Paternò, Fabio and Vanderdonckt, Jean. Past, Present, and Future of Model-Based User Interface Development. *i-com*, 10(3):2–11, 2011.

[moo12] mootools. mootools : a compact javascript framework. `http://mootools.net/` (Retrieved on June 25, 2012), 2012.

[Mye92] Myers, Brad A. and Rosson, Mary Beth. Survey on User Interface Programming. In Bauersfeld, Penny and Bennett, John and Lynch, Gene, editor, *CHI*, pages 195–202. ACM, 1992.

[Mye95] Myers, Brad A. User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.*, 2(1):64–103, 1995.

[Mye00] Myers, Brad and Hudson, Scott E. and Pausch, Randy. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7:3–28, March 2000.

[Nok12] Nokia Corp. Qt : cross-platform application and UI framework. `http://qt.nokia.com/` (Retrieved on June 25, 2012), 2012.

[O'G12] O'Grady, Stephen. The RedMonk Programming Language Rankings: February 2012. `http://redmonk.com/sogrady/2012/02/08/language-rankings-2-2012/` (Retrieved on June 25, 2012), 2012.

[OMG11a] OMG. *OMG Meta Object Facility (MOF) Core Specification : Version 2.4.1.* 2011.

[OMG11b] OMG. Unified Modeling Language^{TM} (UML®). `http://www.omg.org/spec/UML/index.htm` (Retrieved on June 25, 2012), 2011.

[OMG12] OMG. Object Management Group (OMG). `http://www.omg.org/` (Retrieved on June 25, 2012), 2012.

[Osm12] Osmani, Addy. Digesting JavaScript MVC – Pattern Abuse Or Evolution? `http://addyosmani.com/blog/digesting-javascript-mvc-pattern-abuse-or-evolution/` (Retrieved on June 25, 2012), 2012.

[Pin03] Pinheiro da Silva, Paulo and Paton, Norman W. User Interface Modeling in UMLi. *IEEE Software*, 20(4):62–69, 2003.

[Pro12] Prototype. prototype : JavaScript framework. `http://www.prototypejs.org/` (Retrieved on June 25, 2012), 2012.

[Ree78] Reenskaug, Trygve. MVC. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html (Retrieved on June 25, 2012), 1978.

[REL11] RELAX NG. RELAX NG home page. http://relaxng.org/ (Retrieved on June 25, 2012), 2011.

[Sch96] Schlungbaum, Egbert. Model-based User Interface Software Tool : current state of declarative models. 1996.

[Sch12] Schematron. Schematron : a language for making assertions about patterns found in XML documents. http://www.schematron.com/ (Retrieved on June 25, 2012), 2012.

[Sel12] Sellier, Alexis. less : The dynamic stylesheet language. http://lesscss.org/ (Retrieved on June 25, 2012), 2012.

[Sen12] Sencha. Ext JS 4.1 : JavaScript Framework for Rich Apps in Every Browser. http://www.sencha.com/products/extjs (Retrieved on June 25, 2012), 2012.

[Shn10] Shneiderman, Ben and Plaisant, Catherine. *Designing the User Interface - Strategies for Effective Human-Computer Interaction (5. ed.).* Addison-Wesley, 2010.

[Sta07] Stahl, Thomas and Völter, Markus and Efftinge, Sven and Haase, Arno. *Modellgetriebene Softwareentwicklung : Techniken, Engineering, Management.* dpunkt.verlag, Heidelberg, #may# 2007.

[Ste09] Steinberg, Dave and Budinsky, Frank and Paternostro, Marcelo and Merks, Ed. *EMF: Eclipse Modeling Framework.* Addison-Wesley, Boston, MA, 2. edition, 2009.

[Sty12] Stylus. Stylus : Expressive, dynamic, robust CSS. http://learnboost.github.com/stylus/ (Retrieved on June 25, 2012), 2012.

[Sze96] Szekely, Pedro A. Retrospective and Challenges for Model-Based Interface Development. In Bodart, François and Vanderdonckt, Jean, editor, *DSV-IS*, pages 1–27. Springer, 1996.

[Tra02] Traetteberg, Hallvard. Model-based user interface design. http://www.idi.ntnu.no/~hal/_media/research/thesis.pdf (Retrieved on June 25, 2012), 2002.

[Tra09] Traetteberg, Hallvard. Integrating Dialog Modeling and Domain Modeling : the Case of Diamodl and the Eclipse Modeling Framework, 2009.

[Twi11] Twitter Inc. Bootstrap, from Twitter. http://twitter.github.com/bootstrap/ (Retrieved on June 25, 2012), 2011.

[Usi07a] UsiXML. User Interface eXtensible Markup Language. http://www.usixml.org/ (Retrieved on June 25, 2012), 2007.

[Usi07b] UsiXML. UsiXML : Reference Manual. http://www.usixml.org/index.php?mod=download&file=usixml-doc/UsiXML_v1.8.0-Documentation.pdf (Retrieved on June 25, 2012), 2007.

[W3C95] W3C. Overview of SGML Resources . http://www.w3.org/MarkUp/SGML/ (Retrieved on June 25, 2012), 1995.

[W3C04a] W3C. RDF Primer : W3C Recommendation 10 February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/ (Retrieved on June 25, 2012), 2004.

[W3C04b] W3C. XML Schema Part 0: Primer Second Edition ; W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-0/ (Retrieved on June 25, 2012), 2004.

[W3C04c] W3C. XML Schema Part 1: Structures Second Edition ; W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-1/ (Retrieved on June 25, 2012), 2004.

[W3C04d] W3C. XML Schema Part 2: Datatypes Second Edition ; W3C Recommendation 28 October 2004. http://www.w3.org/TR/xmlschema-2/ (Retrieved on June 25, 2012), 2004.

[W3C05] W3C. Document Object Model (DOM). http://www.w3.org/DOM/ (Retrieved on June 25, 2012), 2005.

[W3C07] W3C. XSL Transformations (XSLT) Version 2.0 : W3C Recommendation 23 January 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/ (Retrieved on June 25, 2012), 2007.

[W3C08] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition) : W3C Recommendation 26 November 2008. http://www.w3.org/TR/xml/ (Retrieved on June 25, 2012), 2008.

[W3C11a] W3C. W3C - Model-Based User Interfaces Working Group Home Page. http://www.w3.org/2011/mbui/ (Retrieved on June 25, 2012), 2011.

[W3C11b] W3C. XMLHttpRequest Level 2 : W3C Working Draft 16 August 2011. http://www.w3.org/TR/2011/WD-XMLHttpRequest2-20110816/ (Retrieved on June 25, 2012), 2011.

[W3C12a] W3C. Cameleon Reference Framework Diagram. http://www.w3.org/2005/Incubator/model-based-ui/wiki/images/d/d9/Camelon-reference-framework.png (Retrieved on June 25, 2012), 2012.

[W3C12b] W3C. HTML5 : a vocabulary and associated APIs for HTML and XHTML. http://www.w3.org/TR/html5/ (Retrieved on June 25, 2012), 2012.

[W3C12c] W3C. W3C - HTML. http://www.w3.org/standards/techs/html#w3c_all (Retrieved on June 25, 2012), 2012.

[Wik12a] Wikipedia. Wikipedia - Desktop metaphor. http://en.wikipedia.org/wiki/Desktop_metaphor (Retrieved on June 25, 2012), 2012.

[Wik12b] Wikipedia. Wikipedia - XML schema language comparison. http://en.wikipedia.org/wiki/XML_schema_language_comparison (Retrieved on June 25, 2012), 2012.

[Zen12] Zend. Zend Framework. http://framework.zend.com/ (Retrieved on June 25, 2012), 2012.

A Detailed Generation Step Descriptions

This appendix contains complementary descriptions of the generation steps that characterize the two prototypes. In the case of the web application the methods are visualized and explained in more detail than in chapter 6. The screen cast demonstrating the web solution focuses on its usage since the generation of the web application is accomplished without much user interaction. Enabling the reader to comprehend the functionality and purpose of each method in the Python module is the goal of the web application detailed generation step description.

For the EMF/EEF editor the necessary wizard and configuration steps are explained. These are based on two tutorial guides that the EMF and EEF developer teams provide. The purpose is to especially enable ISCUE's staff to be able to reproduce the generations made in Eclipse.

A.1 Web Application

The processes for generating the web application components are written in the Python scripting language. The author developed a Python module containing all relevant methods for transforming PyXB Python classes into models, templates (i.e., views), controllers, JavaScript-XML bindings and the static HTML application file. This module is usable in two ways that are common practice in Python development: it is a) executable from the command line with arguments parsed with the `argparse` module and it is b) possible to `import` it (or only specific parts) from another Python script in order to use the contained methods programmatically.

As the module relies on PyXB class output in order to provide models, views and controllers, the PyXB generator script included in the PyXB package is to be executed first. The author of this thesis developed another Python script wrapping the PyXB generation step and the methods of the aforementioned Python generator module. It takes as a mandatory command line argument the name(s) of the XSD input file(s). Optionally the path to the PyXB generator script can be provided in case it is not included in the `PATH` environment variable.

The most important transformation and templating methods of the process are visualized as flow charts (in some cases slightly simplified and abstracted) in figures A.1 through A.7. They are described in the following chapters.

A.1.1 Environment Setup

The development environment used has several aspects. The generation scripts were written in Python, executed with Python v2.7.2. The PyXB library was installed in v1.1.3 and its installation requires the Python package `setuptools`. Templating is done with the package `mako`, v0.7.0 is used. The Python installer itself, setuptools (both for Win32 platforms), PyXB and mako are available on the thesis CD-ROM. Unixoid operating systems usually do provide Python by default. The exact Python version should not critically matter, but e.g. the package `argparse` is only available from Python 2.7 onwards.

The application has been run on Mozilla Firefox v12 and v13 and Google Chrome v19. The Firefox v13 installer is available on the CD-ROM, Google Chrome does not provide a stand-alone installer file.

Compilation of CoffeeScript is usually done offline via the node.js compiler, eventually complemented by other processes such as minification of the resulting JavaScript file(s) and bundling up all files in one. These steps are omitted for this prototype and the inline CoffeeScript compiler is used. It is remarkably influencing performance, even in small use cases as the one described here but since it is a proof of concept rather than a performance demonstration this is a trade-off being made in favor of development simplicity.

A.1.2 Preparation Methods

PyXB provides a class hierarchy for the simple and complex types the input XSD defines. These classes and its properties have to be inspected by the script and function as the parametrization of the templating mechanisms that output the model, view and controller instances for JavaScript and CoffeeScript respectively. The first step is retrieving Python dicts (*dict* is Python's abbreviated form for *dictionary*) from these PyXB classes. Simple types, complex types and XML Schema's own simple type definitions are processed separately.

Method `simpleType2Python(...)` Figure A.1 depicts the process of transforming simple type definitions to Python dicts. The first condition is whether the type is named (i.e., the XSD provides a `name` attribute for it) or anonymous (i.e., it is an unnamed child of an `xsd:element` or `xsd:attribute` node). Names from the XSD are reflected with namespace and local part, anonymous simple types have a formalized class name in the form `STD_ANON_`[number] where [number] is one of the continuously assigned numbers for anonymous simple types. Thus all anonymous types are unique throughout the complete namespace group that aggregates all namespaces involved.

Included in the resulting dict are enumeration lists, constraints concerning minimum and maximum values, assertions about (minimum, maximum or exact) length, regular expression patterns. These pieces of information allow for appropriate transformations when models, templates and controllers are generated.

Method `complexType2Python(...)` Similar to the method described above for transforming simple types into Python dicts exists a method accomplishing this task for complex type definitions. It also distinguishes between named and anonymous types, taking into account the more precise information available.

In terms of the content model there are three kinds of constructs inspected and reflected accordingly: a) XSD groups (sequence, choice, all), b) elements and c) attributes. All these are directly adapted and thus the complete structure is available in the resulting dict. For groups and elements the relevant properties are name, type (group

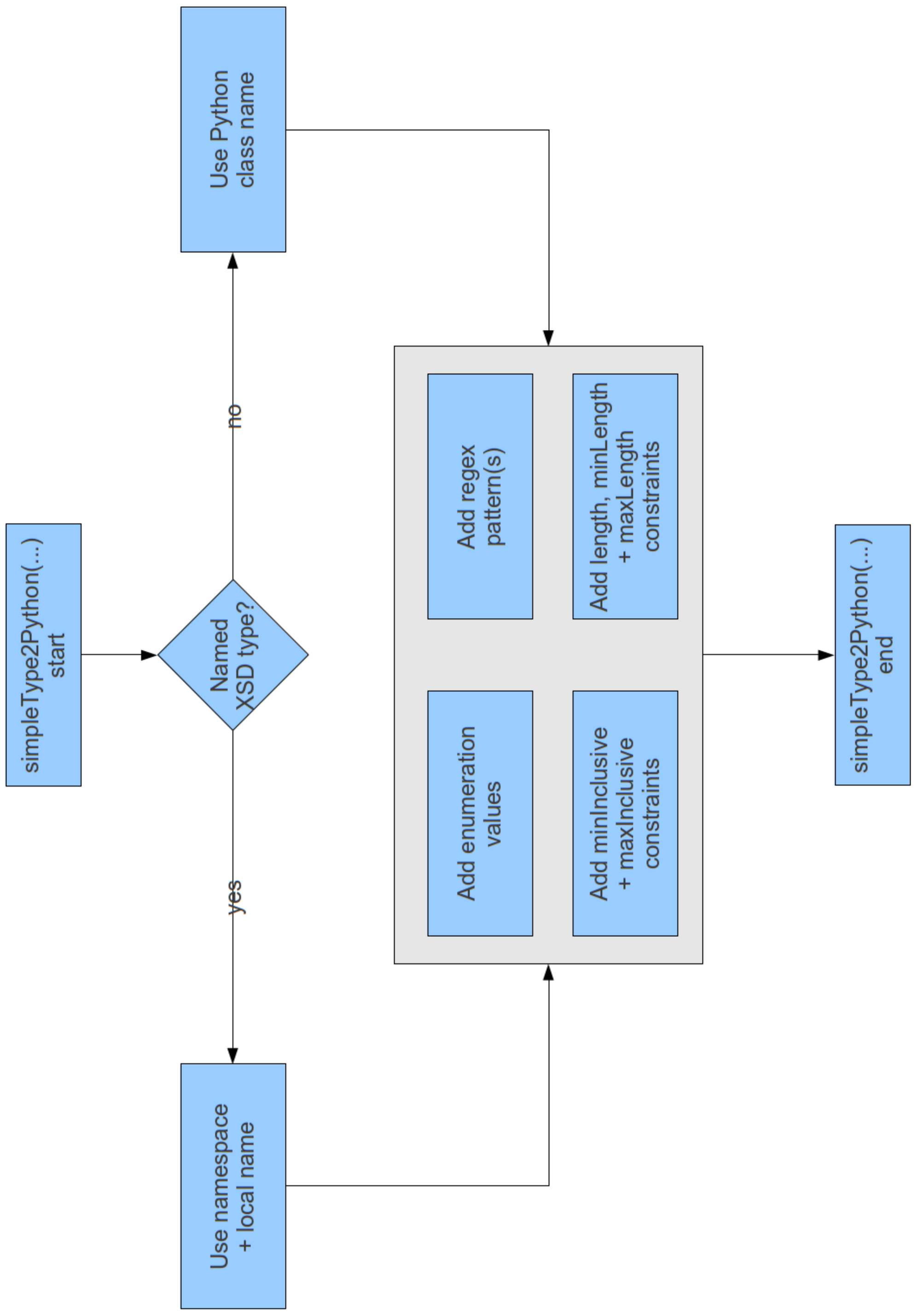

simpleType2Python(...) start
Named XSD type?
no
yes
Use Python class name
Use namespace + local name
Add regex pattern(s)
Add length, minLength + maxLength constraints
Add enumeration values
Add minInclusive + maxInclusive constraints
simpleType2Python(...) end

Figure A.1: Flow chart: method `simpleType2Python(...)`

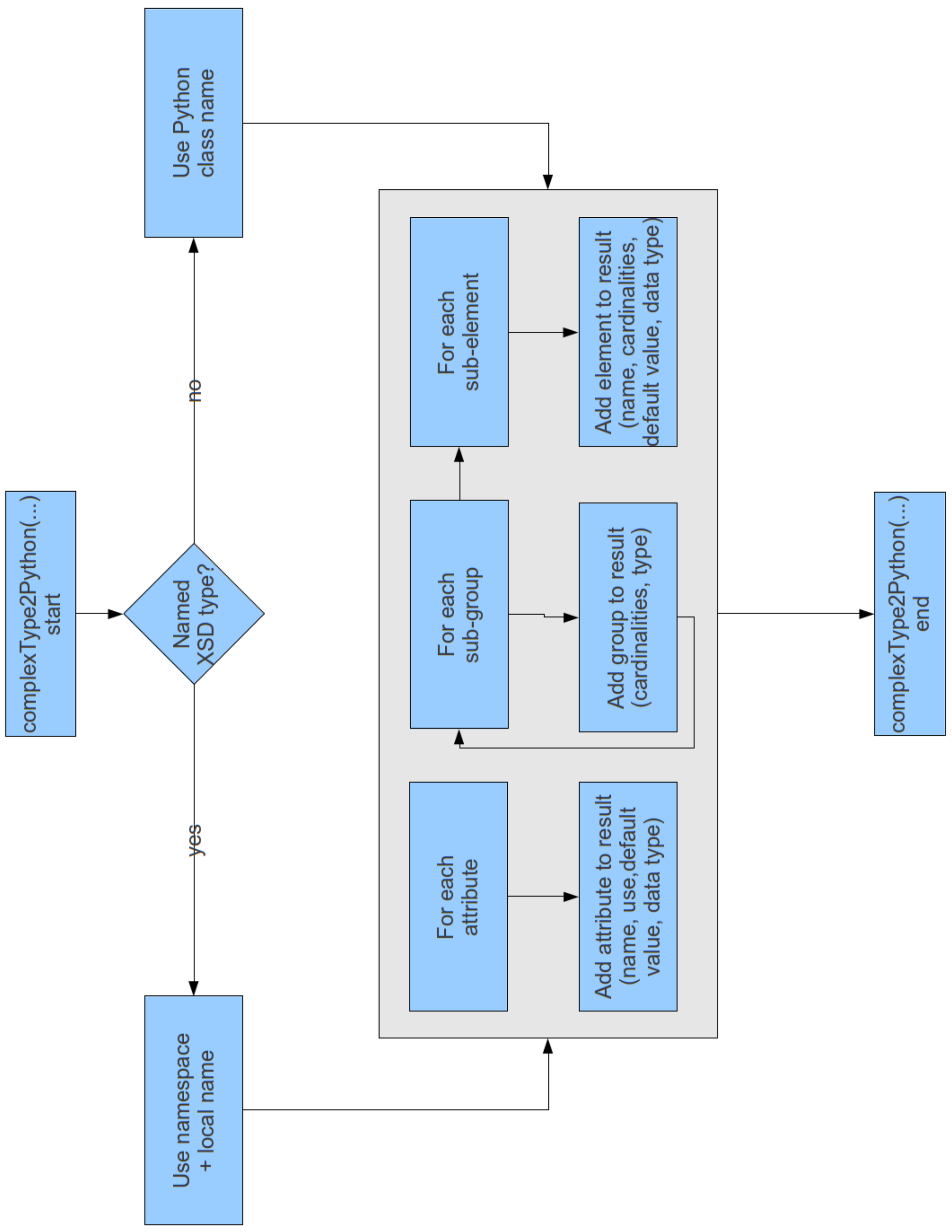

complexType2Python(...) start
Named XSD type?
no
yes
Use Python class name
Use namespace + local name
For each attribute
Add attribute to result (name, use,default value, data type)
For each sub-group
Add group to result (cardinalities, type)
For each sub-element
Add element to result (name, cardinalities, default value, data type)
complexType2Python(...) end

Figure A.2: Flow chart: method `complexType2Python(...)`

type for groups, data types for elements), cardinalities (`minOccurs` and `maxOccurs`). Elements can also have default values, so can attributes. For the latter there are no cardinalities, but attribute use assertions (required, optional or prohibited). A `fixed` attributes indicates that the value of this attribute is set by default and cannot be changed. These information are transported into the Python dict.

In addition to the directly copied information on cardinalities, the method computes "acummulated" `minOccurs` and `maxOccurs` values by mulitplying the values of all constructs from the respective component up to the content model root. These computed values are used by the method `getFlatElementsMapForComplexType(...)`.

Method `getFlatElementsMapForComplexType(...)` For generating models, views and controllers out of the XSD structures, the often very verbose and complex structure of complex type definition is flattened to a simplified Python dict. It contains the (in any case flat) list of attribute uses and a list of potential sub-elements retrieved by skipping groups when encountered in iteration, only taking elements into account. This takes as input the rsult of the nested Python dict created by method `complexType2Python(...)` (which is scripted explicitly mainly for debugging and logging reasons in addition to a better over-all maintainablity enabled by such a two-fold process). The acummulated cardinalities mentioned above are the only cardinality values existing in the resulting dict. Iterating over the elements is directly possible and XSD groups are abstracted out. This leads to information loss in some cases, when it comes to preserving the order of sub-elements in the parent element. This simplification is a trade-off made in favor of comprehensibility of the resulting UI components.

A.1.3 Templating Methods for MVC Components

The instances of MVC's three component types models, views and controllers are generated with the Python templating library mako. It uses the same syntax for expression contexts (${...}) as jQuery templates, that makes escaping necessary when generating templates. Expression contexts for later jQuery evaluation have to be transported into the result that mako puts out. Formally, a templating language (mako) is used to produce text string output at generation time that itself contains templating syntax for another templating engine (jQuery tmpl) that will be evaluated at run time.

Basically two different sorts of generation output is required for the generated components: JavaScript (for an object literal holding the template strings, for the `Backbone.RelationalModel` classes and the Jsonix mappings) and CoffeeScript for the `Spine.Controllers`. An important aspect to be taken into account when designing the mako templates is the fact that the CoffeeScript syntax is whitespace-sensitive and indentation carries information about expression blocks. This does not apply to JavaScript but a clear and appropriate source code indentation is important for debugging and comprehensibility issues there as well.

Method `getModelForComplexType(...)` Models for the generated web application are implemented as sub-classes of the `Backbone.RelationalModel` class. It origins from the *Backbone-Relational* extension of the web development MVC framework *BackboneJS*. It extends `Backbone.Models` by providing support for one-to-one, one-to-many

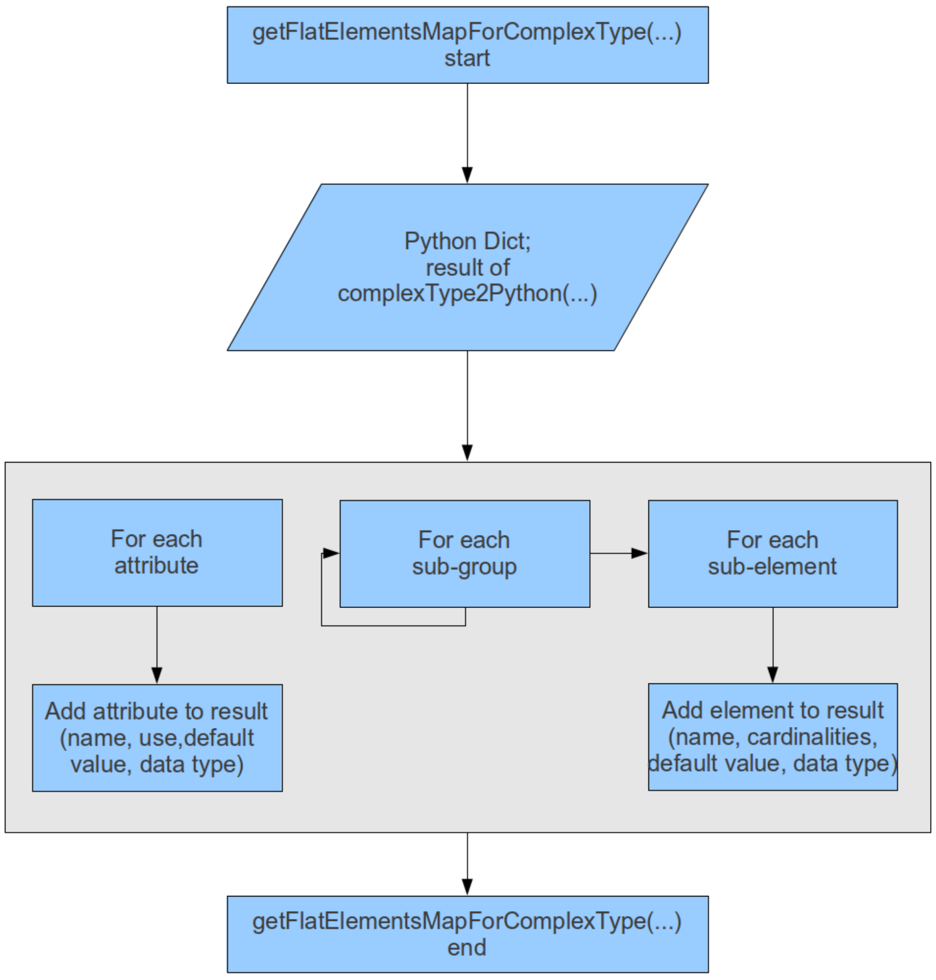


Figure A.3: Flow chart: method `getFlatElementsMapForComplexType(...)`

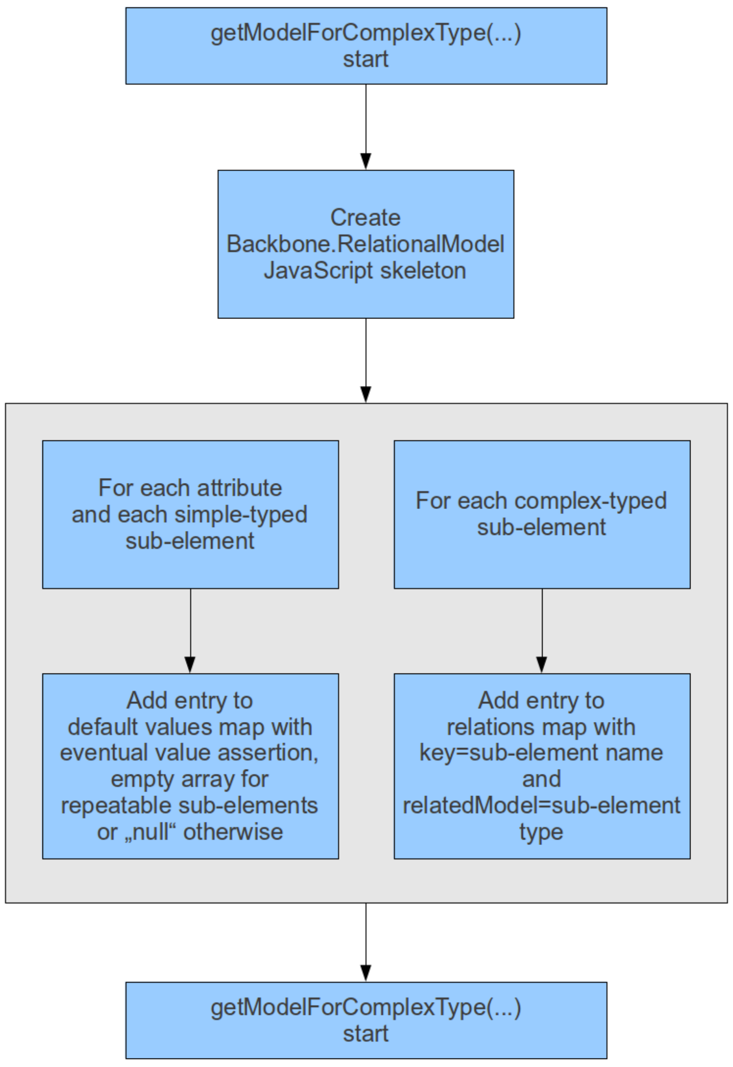

getModelForComplexType(...) start
Create Backbone.RelationalModel JavaScript skeleton
For each attribute and each simple-typed sub-element
For each complex-typed sub-element
Add entry to default values map with eventual value assertion, empty array for repeatable sub-elements or „null“ otherwise
Add entry to relations map with key=sub-element name and relatedModel=sub-element type
getModelForComplexType(...) start

Figure A.4: Flow chart: method `getModelForComplexType(...)`

and many-to-many relations between model instances. Including the expanded related instances into the JSON serialization automatically belongs to the features this class brings. The relations entry in such a `Backbone.RelationalModel` holds information about all out-going relations. In the method that is described, all complex-typed sub-elements are represented as entries in this relations component of the model definition. The `key` is the name for addressing the related instances, `relatedModel` references the model of which the related instance are. Property `includeInJSON` is used to assert whether to include complete instances as properties in the serialization, only specific fields or nothing at all. Defining the relation type, `type` can take one of the values `Backbone.HasOne` and `Backbone.HasMany`. In this context, `Backbone.HasMany` is always used due to easier maintainablity when setting/adding concrete relation values. Potential reverse relations that can be asserted that lead to automatic addition of the respective reverse direction are not used in this application context since there is no corresponding feature in the XSD and no such direct need arose in the development phase.

All simple-typed sub-elements and all attributes get an entry in the `defaults` class component. When default values de facto exist, they are set on the respective element/attribute, otherwise an empty array is asserted for repeatable elements/attributes, or null if none of these cases apply.

The model class source code is generated as JavaScript code. *BackboneJS* does support CoffeeScript but the *Backbone-Relational* extension does not completely conform to the paradigms CoffeeScript introduces (e.g., native class support emulation). This is the reason for choosing JavaScript directly as the target language for model class definitions.

Method `getTemplateForComplexType(...)` Templates take the role of MVC views, in this case jQuery templates. For each attribute of the input type dict a jQuery template call invoking the template for the data type of the respective attribute is generated. This jQuery template call is wrapped in a jQuery template conditional statement checking for the existence of a value for this data field. Nota bene: this jQuery template conditional and template call will be executed at runtime, only the mako templates are directly evaluated in the Python script.

For each simple-typed sub-element of the type jQuery template calls are generated analogously to the ones generated for attributes in all cases where an element has a value for `minOccurs` > 1 (i.e., it is required). Such jQuery template calls are included for `minOccurs` times. This ensures the existence of form elements for at least the required number of such sub-elements. Such expansion is an element of user guidance that can help ensure the entry of valid data for elements. Buttons for adding instances of complex-typed sub-elements complement the template calls.

Method `getControllerForComplexType(...)` Controllers, as the third kind of component in the MVC paradigm, are generated as CoffeeScript classes extending `Spine.Controller`. *Spine*, similar to *BackboneJS* (note the similarity in terms of nomenclature metaphors of both frameworks), provides constructs for defining MVC components. *Spine* is in most cases more light-weight, *BackboneJS* does not directly implement a

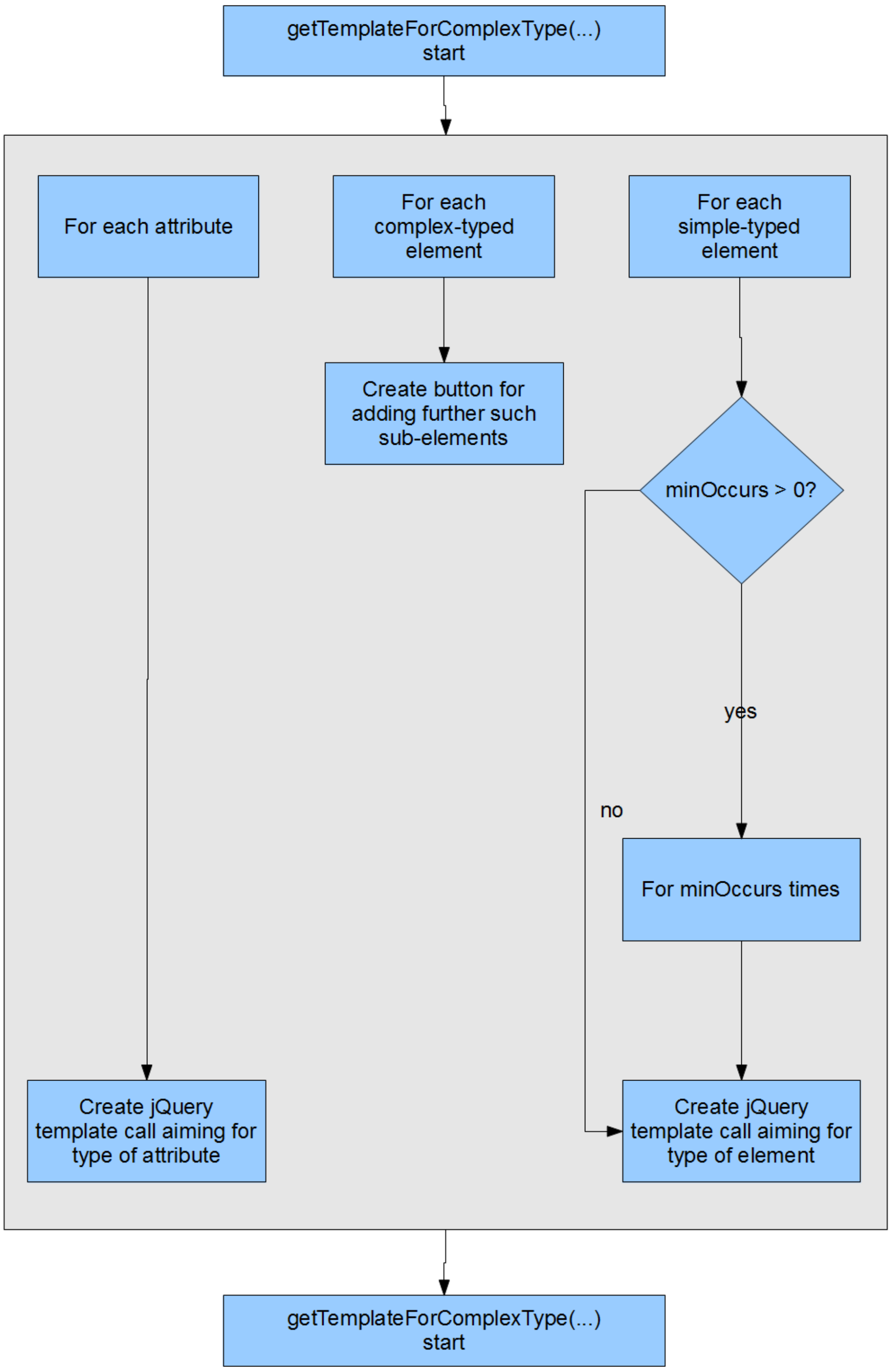

getTemplateForComplexType(...) start
For each attribute
For each complex-typed element
For each simple-typed element
Create button for adding further such sub-elements
minOccurs > 0?
yes
no
For minOccurs times
Create jQuery template call aiming for type of attribute
Create jQuery template call aiming for type of element
getTemplateForComplexType(...) start

Figure A.5: Flow chart: method `getTemplateForComplexType(...)`

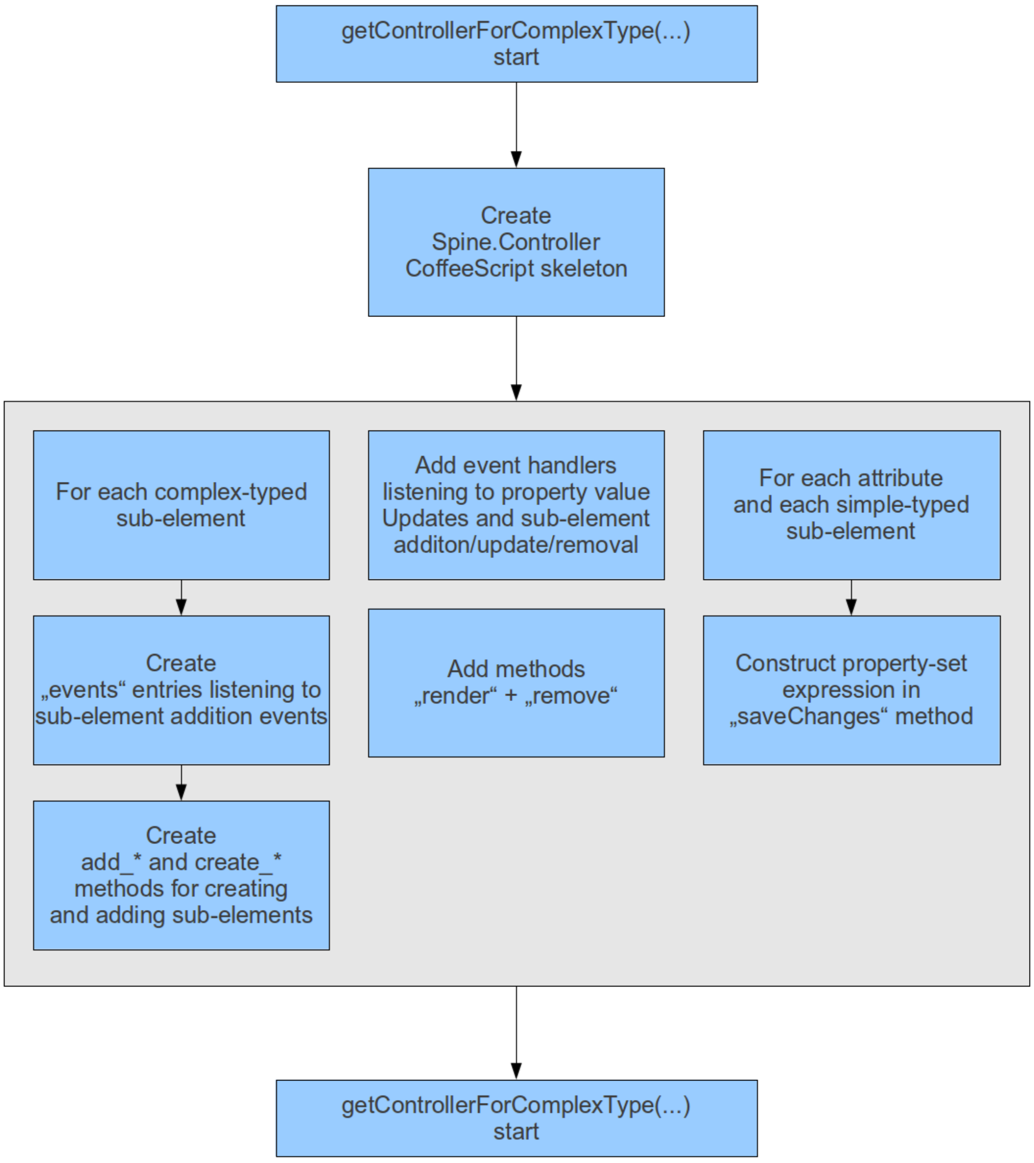


Figure A.6: Flow chart: method `getControllerForComplexType(...)`

Controller class but includes controller-like functionality in its views, that it explicitly holds in JavaScript (or CoffeeScript) objects, contrary to *Spine*.

The basic parts of `Spine.Controller`s that are generated by this method are a) the `events` map declaring events to listen to, `add_*` and `create_*` methods for adding (complex) sub-elements and their respective controllers, the constructor function, methods for rendering and removing and the `save_changes` method for setting the attribute values.

Method `getAppController(...)` In addition to the type-specific controllers that are responsible for their attached items there needs to exist an application controller being the entry point into the application. It instantiates the top-level element controller(s).

For every possible top-level element the generator designates code fragments, all of which are commented by default. The application is only functional when one of these fragments is un-commented.

The application controller gets a `doOutputSource` method that is triggered whenever the static "get data" button is clicked. Its purpose is to gather the data from the editor's model instance (in fact, the application controller's item), convert it to JSON, invoke the Jsonix marshaller with this JSON data, beautify the resulting XML data with the `vkbeautify` script and output the pretty-printed XML data to the right-hand textarea on the applications interface.

Necessary for marshalling specific JSON data structures are mappings conforming to the Jsonix library. These are generated along with models, views and controllers.

A.1.4 Additional Generation Methods

Method `getJsonixMappingForComplexType(...)` Jsonix provides a script evaluating JavaScript objects as the basis for JSON-to-XML mappings. This method generates such a mapping object for each complex XSD type. Figure A.7 depicts the process. First of all, a declaration for each type has to be generated in order to make types referencable. This declaration only sets an object and attaches a name property to it.

After the declarations generation all complex types are iterated again and for each type all sub-elements and attributes are asserted with their respective names and data types. Since Jsonix does not fully support XML Schema's simple data types, only Boolean is specifically resolved as such, all other simple types are resolved to `xsd:string`s. This does not limit the serialization functionality since validation is not part of Jsonix' functionality anyway.

The last piece of information needed by Jsonix' marshalling and unmarshalling processes is a list of possible top-level elements. These are iterated and a name-type relation is appended for each.

Method `getHTMLStaticFile(...)` The static part of the application is generated by this method. It concatenates strings for the HTML header, the controllers and the HTML body. The script includes in the header are

- jQuery,
- jQuery templates plugin,

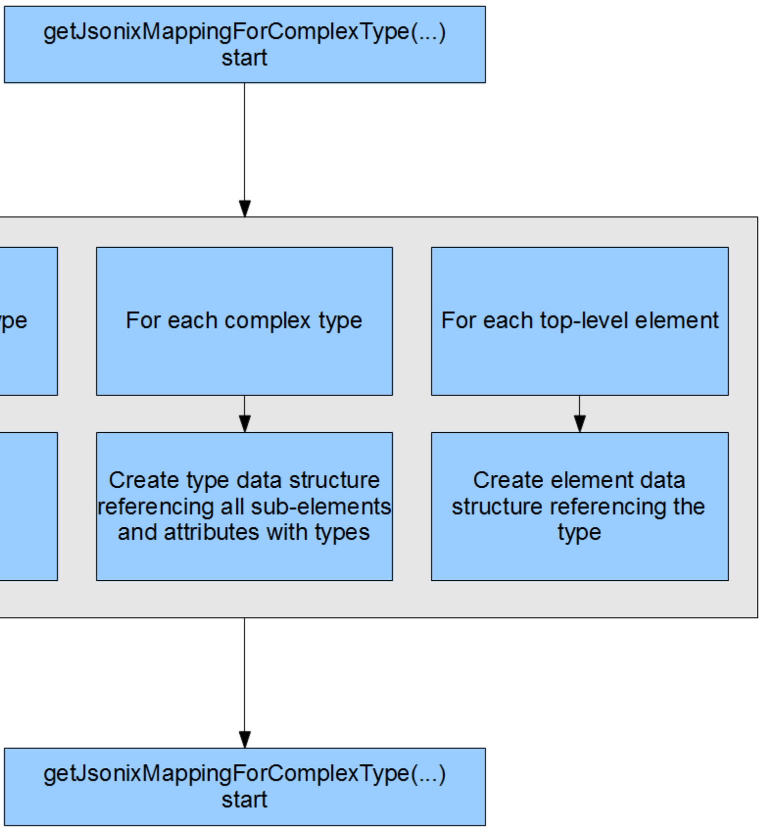

getJsonixMappingForComplexType(...) start
For each complex type
Create type declaration
For each complex type
Create type data structure referencing all sub-elements and attributes with types
For each top-level element
Create element data structure referencing the type
getJsonixMappingForComplexType(...) start

Figure A.7: Flow chart: method `getJsonixMappingForComplexType(...)`

- the inline CoffeeScript compiler script,
- Spine,
- UnderscoreJS (required by BackboneJS),
- BackboneJS,
- Backbone-Relational,
- the Jsonix library,
- the application's Jsonix mapping file,
- vkbeautify as the used XML beautifying script,
- the application's models and
- the application's templates.

The content of the HTML file is only depending on the controllers for the application, which are directly inserted into a CoffeeScript `<script>` element. All other content is statically put out into the file.

A.2 Eclipse-Based

A.2.1 Environment Setup

The Eclipse IDE was used in version 3.7 *Indigo* with modelling components on both Ubuntu Linux (32 bit) and Windows 7 (32 bit). The archive files for these releases can be found on the thesis' CD-ROM in the `eclipse` folder.

EMF and EEF were bundled with these releases, EMF in version 2.7.0 and EEF in version 1.0.2.

A.2.2 Eclipse Modeling Framework (EMF)

Prior to any of the EMF-specific generation steps it is useful[23] to disable the creation of Ecore *FeatureMaps*. These are incorrectly handled by EEF in the later generation steps, so their creation is suppressed in the first place. EMF creates *FeatureMaps* for repeatable (i.e., with `maxOccurs` > 1) XSD content groups and mixed complex types. These have to be complemented in the original XML Schema with the attribute `ecore:FeatureMap=""` in order to prevent EMF from building *FeatureMap*s for them.

The tutorial [Ecl06a] was used for these steps. It explains the prerequisites that are necessary when generating an EMF plugin for Eclipse. The plugin packages

- `org.eclipse.emf.common`,
- `org.eclipse.emf.ecore`,
- `org.eclipse.emf.mapping` and
- `org.eclipse.emf.codegen`

have to be installed. This can be checked through the plugin overview Eclipse provides under *Help / About Eclipse / Installation Details / Plugins*.

Using the XSD-to-EMF wizard starts with creating a new project. The type of the project has to be *EMF Project*, listed below the *Eclipse Modeling Framework* group in

[23]It proved to be even necessary to do so in case of the combination of EMF and EEF in the *PrIML* use case.

the wizard for project creation. Entering a unique project name and proceeding leads to the choice of which model type to use for the EMF plugin. Ecore, Rose, UML and XML Schema are possible, for this use case XML Schema is chosen. The next step requires the path to the XML Schema file that should be used. It can either be resolved from the workspace if it already exists in it or from any other path in the file system. In case the desired XML Schema definition is separated into more than one XSD file, it is useful to choose the *master* XSD file that (directly or indirectly) imports the other XSDs. (The *PrIML* vocabulary for example is separated into seven (eight when taking PrIOF into account as well) files which are all (most of them directly, some indirectly) referenced (via `xsd:import` statements) from the PrIML_Schema.xsd file.)

While loading the respective XSD file(s) the wizard is blocked from user interaction. When it finishes loading the input field *Generator model file name* has been filled in. The default value is the XSD local file name with an extension set to `.genmodel`. The option *Create XML Schema to Ecore Map* has to be checked in order to generate the mapping needed for serialization and de-serialization.

The following overview lists the Java packages that have been built for the namespaces resolved from the input XSD(s). All packages that shall be included in the plugin have to be checked. The same applies to the list below showing referenced namespaces. Clicking the *Finish* button creates a project with the name entered at the beginning of the wizard. It contains a Java Runtime Environment (JRE), a model folder containing a `.ecore`, `.genmodel` and `.xsd2ecore` model file for each of the packages created from the input XSD(s) and a source folder (`src`) holding the generated Java source code files for the elements, simple types and complex types.

Opening the `.ecore` model can be done with an XML editor or with the *Sample Ecore Model Editor* for a tree view reflecting the Ecore model derived from the XML Schema model input. Renaming of types and attributes can be applied there, data types or constraints of attributes can be customized. All facets that Ecore provides are reflected in the property view describing the model components.

The `.genmodel` holds meta data about the generation steps that follow this model-to-model (i.e., XSD to Ecore) and the basic model-to-code transformation (Ecore to Java classes). A possible configuration that can be taken in the `.genmodel` is setting the *Create Child* option to `false` for features that shall not be shown in the EMF (and especially later to-come EEF) tree view. Right-clicking the `.genmodel`'s tree view root element provides five *Generate ...* options. Clicking *Generate All* invokes all possible EMF generation steps in the logical order they require. This creates three more projects in the workspace, next to the basic EMF project just generated: `[project name].edit`, `[project name].editor` and `[project name].tests` (where [project name] is a wildcard for the originally chosen EMF generation project name).

Forming the edition basis for the generated classes, the `.edit` project contains Item-Providers for all types in the model. ItemProviders enable other classes and processes to view and edit instances of the classes the input model introduces. They provide content and labels and handle notifications in case of changes/updates (for ItemProviders cf. [Ste09], pp. 46ff.).

The `.editor` project is the working plugin executable in the Eclipse workbench. It can be executed by right-clicking it and choosing *Run As / Eclipse Application.* This causes a second Eclipse instance (the workbench) to open in which the editor plugin is running.

A.2.3 Extended Editing Framework (EEF)

In the Eclipse Modeling Framework Technology (EMFT) project exist several other sub-projects complementing EMF Core. One highly relevant to the approach taken in this thesis is EEF with its generation steps on top of EMF plugins.

The tutorial [Ecl12b] was used for the following generations. It demands to apply the steps of the installation guide. The respective update sites for stable and nightly builds of EEF are linked there; EEF needs to be installed if not already available through Eclipse's modelling component set (see chapter A.2.1). When EEF is properly installed, the tutorial steps can be applied to an EMF plugin set as the result of chapter A.2.2.

First a destination folder for EEF's models needs to be set up. The tutorial recommends to call it *models*, attempting to keep consistency to the EMF model folder, it can be called *model* (singular form). It is placed in the `.edit` plugin project. Right-clicking on the EMF `.genmodel` file opens the context menu with the menu item *EEF / Initialize EEF models*. Applying this step leads to a wizard asking for the just created destination folder for the EEF models. Confirmation causes the creation of a `.components` and a `.eefgen` model.

The next step is creating a new source folder in the `.edit` project. Note that a usual folder (*Right-click / New / Folder*) does not suffice, choose *Right-click / New / Source Folder* instead and the name has to be *src-gen*, which is a common practice in software component generation. Before generating the source code, the EEF runtime package (`org.eclipse.emf.eef.runtime`) has to be added to `.edit`'s plugin.xml file. Open it with the *Plugin Manifest Editor* and choose the *Runtime* tab. After adding the package open its properties and check the *Reexport this dependency* in order to make the EEF runtime package available to further plugins, namely `.editor`. After these configurations right-click the generated `.eefgen` model and choose the *EEF / Generate EEF architecture* option.

Due to a reported bug in the used EEF version[24] there are incorrect references to a non-existing Java class `EStringToStringMapEntry` in every *.edit/src-gen/*/components /DocumentRootPropertiesEditionComponent.java* file. Changing the references to the implementation class `EStringToStringMapEntryImpl` is the recommended solution and it indeed solves the errors caused by the otherwise false references. The adjusted methods can be marked with `@generated NOT` tags in order to preserve the change when regenerations are applied.

When the generation is finished, the plugin.xml file in the `.edit` project has to be opened with the Eclipse manifest editor. On the runtime tab, the generated **.providers* classes have to be added. After that, re-opening the same plugin.xml file with the XML or text editor is needed in order to paste the content of the generated *_properties.plugin. xml file(s) into the plugin.xml.

The last manual step in EEF's enhancement is overwriting some existing class methods and one property in the **Editor.java* class in the `.editor` project. See the original

[24]cf. http://www.eclipse.org/forums/index.php/mv/msg/206326/660537/

tutorial on the thesis CD-ROM for the exact methods and property to replace.

After these generation, dependency configuration and manual copy-paste steps the `.editor` plugin is enhanced with form-based property views and data entry dialog windows. All further customizations are optional, some of the ones attempted in the context of this thesis are described in chapter 6.3.2.

B Thesis CD-ROM Contents

The CD-ROM accompanying this thesis contains resources and material used and/or produced in the context of the thesis. See the following list for detailed information:

- The thesis PDF
- Source code
  - *PrIML* XML Schema collection
  - *PrIML* subset isolated for better development maintainability
  - Web Application Generator
    - Self-developed, commented Python module `pyxb_to_mvc_web_app` and helper script `generate.py` together with library files
    - Generated application for *PrIML* subset
  - EMF/EEF
    - Workspace copy containing EMF editor
    - Workspace copy containing EEF editor
    - Workspace copy containing customized EEF editor
    - Saved tutorial used for EEF generation and setup
- Eclipse Modeling Components package installer (Windows 32-bit)
- Python development environment used
  - Python v2.7.2
  - Python package `pyxb` v1.1.3
  - Python package `setuptools` v0.6c11
  - Python package `mako` v0.7.0
- Mozilla Firefox v13 installer (Windows 32-bit)
- *PrIML* XSDs and visualizations as .png files
- Screen casts of generation processes, some customizations and basic editor usage