# An Interpretative Approach to the Model-Driven Development of Web Applications

**Dissertation**

zur Erlangung des Grades eines

D o k t o r s  d e r  N a t u r w i s s e n s c h a f t e n

der Universität Dortmund
am Fachbereich Informatik
von

Stefan Haustein

Dortmund
2006

**Tag der mündlichen Prüfung:** 20.02.2006

**Dekan:** Prof. Dr. Bernhard Steffen

**Gutachterin/Gutachter:** Prof. Dr. Katharina Morik,
Prof. Dr. Ernst-Erich Doberkat

To Janine

**Acknowledgements**

I am grateful to my supervisor Katharina Morik for fruitful and helpful discussions and for providing the free and encouraging environment that made this work possible. I also thank my co-supervisor Ernst-Erich Doberkat for accepting the task of examining this thesis and for providing helpful feedback. I am grateful to Sonja Haustein, Michael Kroll, Ingo Mierswa, Klaus Unterstein, and Michael Wurst for reading this thesis or testing the Infolayer system and providing useful suggestions. Special thanks go to Jörg Pleumann for the inspiring cooperation.

**Abstract**

The increasing size and complexity of web applications has led to a situation where the traditional approach of creating and managing a set of plain HTML files is inappropriate in many cases. Consistency in structure, look and feel, and hyperlinks needs to be maintained, and support for different content formats may be required. The combination of XML Schema, XML and XSLT is able to improve this situation, but the expressive power of XML Schema is insufficient for application domains where more than a pure hierarchical structure is required.

In this work, we have chosen the XML toolchain as a guideline to construct an alternative basis for web information systems at a higher level of abstraction, namely UML class diagrams. We have identified a UML counterpart or implemented a substitute for each constituent of the XML processing chain, showing that it is possible to build a consistent UML-based system for model driven web applications.

Since our approach is based on model interpretation, a system prototype can be created by simply drawing a conceptual model in the form of a UML class diagram—a step that is required in the relevant development methodologies anyway. By making this first step immediately operational without any compilation or transformation steps, the gap between web development methodologies and actual system implementation has been narrowed significantly.

# Contents

# Chapter 1

# Introduction and Motivation

The goal of this work is to simplify the development and maintenance of information systems that are accessible via the World Wide Web for human users and application programs.

At the very beginning, the World Wide Web consisted of a set of simple servers, delivering text files annotated in the Hypertext Markup Language (HTML) [9] to web browsers, but development of web technologies did not stop there. Over the time, the web landscape became more and more complex. Originally, HTML was a hypertext format allowing to structure text into paragraphs, different levels of headings, and other logical document entities—but HTML did not provide any means to control physical format options such as the color or font size of text. Thus, browser vendors soon started to add their own proprietary markup elements, providing more control over the appearance of web pages. Meanwhile, official versions of HTML were supplemented by the Cascaded Style Sheet specification (CSS) [72], suiting the need of fine-grained control over the layout while re-establishing a clean separation of the layout from the content of a document.

Of course, the "document" structure of HTML is not directly suitable for all kinds of data one may want to present on the web. To provide a higher level of abstraction, and to better integrate content that does not fit in the static document category, the World Wide Web Consortium (W3C) [120] has developed the generic Extensible Markup Language (XML) [122]. The XML specification does not define any markup elements on its own, it just specifies the general document tree structure and syntactical aspects that allow

to distinguish comments, markup and text. In order to produce meaningful content, a Schema — declaring actual elements and their nesting rules — is required. Based on XML, various new content formats were developed. For instance, a binary version of XML (WBXML) [77] and specialized annotation languages such as the Wireless Markup Language (WML) [40] and Cell-HTML (cHTML) [65] were developed for mobile devices such as cell phones and PDAs.

With the growing size of web sites, the need of a clear and consistent structure increased. Sites often feature concepts with fixed properties and associations and a larger set of instances belonging to those concepts. When comparing pages for different instances of the same concept, one would expect the properties to appear in a consistent order and layout, enabling users to find information quickly. It also became increasingly important to integrate existing information sources. Methods to dynamically generate content from database sources such as ColdFusion [14], PHP [97], Active Server Pages [26], or Java Server Pages [82] were developed. Web Technologies are no longer limited to information services, but also used for purely commercial services such as shopping, hotel, or flight booking, including sophisticated authentication mechanisms. Collaborative applications allow users to add or alter information conveniently without leaving the browser interface. Also, web technologies are no longer limited to direct interaction with humans. The Remote Procedure Call mechanisms SOAP [24] and XML-RPC [119] are built on top of web Technologies such as the HTTP protocol [36] and XML. The W3C has started an initiative to build a "Semantic Web", that is a network which can directly be interpreted by computers, providing means for "semantic" services such as searches that can distinguish "potato chips" from "computer chips" and applications that automatically integrate flight and hotel booking from different providers for a journey.

## 1.1 The Problem

For web applications, most of the technologies described above can be combined in arbitrary ways, forming a complex landscape of partially overlapping concepts and technologies that have often evolved separately. The goal of this work is to provide a coherent framework that simplifies the development of web sites, especially where a strong inherent structure must be

preserved. We define a site with a strong structure as a site that features concepts with fixed properties and associations and a set of instances belonging to those concepts. An example for a concept may be a "person" or a "project", and one might want to see one instance of a concept on its own page, with all the pages being interlinked with each other. Properties of a person may be the name and address; an association may be "works in project". Instances may be the employees and projects of a particular company. Web sites featuring this kind of content have stronger consistency requirements than less structured sites. Since especially the maintenance aspect is currently a weak point in web development [78], we will keep a focus on this point. To avoid orphan links or inconsistent information, all side effects of changes must be considered. The addition of a new instance may have effects on linked pages. If an instance is deleted, all references must be deleted, too. The changed name of a project must be updated everywhere the project is referenced. If the strong structure can be formalized, it may actually provide the key to ensure both, layout consistency and structural integrity, automatically.

Of course, we do not try to find a general solution for all possible web-related problems. For sites also featuring different concepts, but no or only vague properties and associations or a hierarchical document-structure, Content Management Systems provide a sufficient solution. Templates for concepts ensure a consistent look and feel—further restrictions are neither needed nor possible due to the diversity of the stored content. Examples for this kind of systems are Messaging Boards, Blogs, News services, or document repositories.

### 1.1.1 Sample Scenario

Before going into details about current approaches, we will construct a sample scenario that fits into our definition of a strongly structured domain, and then sketch a naive approach based on existing web technology. The naive approach does not represent the state of the art, but it helps to illustrate the different categories of problems.

A good example for a strongly structured site is the representation of a university department. There are usually overview pages for the different

concepts of the domain such as research topics, projects, courses, publications and the staff members. The concept pages contain lists of instances, linking to further pages containing more detailed information about a particular topic, project, course, publication, or person.

### 1.1.2 A Naive Approach Illustrating the Problems

To be accessible to a regular web browser, the content must be available in the HTML format. A naive approach, but still commonly used, is to directly store all the information in HTML files. This approach relies only on the file system and a server capable of the Hypertext Transfer Protocol (HTTP). The main advantage of this approach is its simplicity. HTML pages can be created with a regular text editor, and the HTTP server simply maps web addresses to a location in the file system.

However, this approach has several significant drawbacks:

- The lack of a separation of layout information from the content leads to increased maintenance effort, especially when the layout of all pages has to be changed consistently.
- HTML does not support the constraints of the underlying model, and thus provides no help for maintaining structural integrity.
- HTML provides only document centric annotations. The semantics of the model is lost and thus cannot be used for searching hints or to support different content formats.

### 1.1.3 Separation of Concerns

The HTML pages contain three different types of information: The actual text or content, layout information such as font sizes and colours, and structural information, that is, the order or arrangement of components on the page. For multiple instances of the same concept, the layout and structural information should be identical, ensuring a consistent appearance, helping users to quickly find what they are looking for. The best way to ensure consistency is to store information identical for a set of pages only once, avoiding redundancies. HTML allows to factor out the layout information

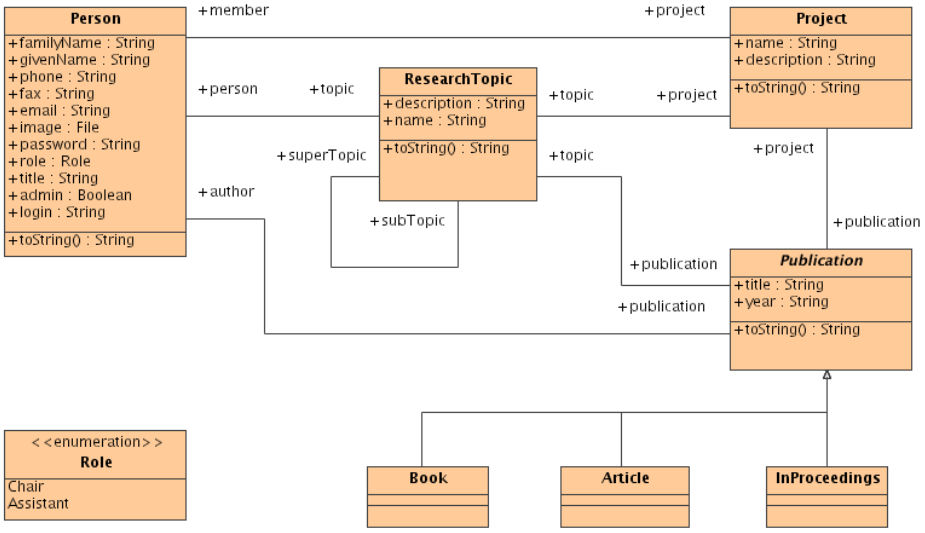


Figure 1.1: Partial Conceptual Model of a Web Presentation of an University Department (UML Class Diagram)

into a centralized cascaded style sheet (CSS), but the same is not fully possible for the structural information.

Intermangling of concerns leads to increased maintenance effort. The content is difficult to maintain when it contains a lot of embedded layout instructions. A consistent change in the layout is difficult because all pages must be changed accordingly.

### 1.1.4 Respecting Structural Constraints

Figure 1.1 shows a small part of a conceptual model matching our example domain in the form of a UML class diagram [93]. The model is actually a part of the model we are using for the web presentation of our own unit; some concepts for tasks like seminar management are omitted to improve readability. In the model, there is an association between persons and projects, denoted by a line connecting the concepts. Setting a link betwen a person $a$ and project $b$ means that person $a$ works in project $b$. To keep the web pages consistent with our model, it would be nice if we could limit "project" links from persons to instances of the concept "project". Unfortunately, in

HTML, links are untyped, as they are in the general Dexter hypertext model [47], a common abstraction of different hypertext systems. There is no way to limit a set of hyperlinks to instances of a certain target concept.

Another property of the association that cannot be represented in HTML is its bidirectionality. It is necessary to set both, a link from a person to a project and a link from a project to a person to express a project membership. Again, redundancy increases the probability of errors and inconsistencies. Ideally, a hypertext system would present a list of projects to choose from and maintain both link ends automatically when the content maintainer edits a link.

In plain HTML, consistency of the page structure with the underlying model including bidirectional link consistency needs to be maintained manually.

### 1.1.5 Content Suitable for Mobile Devices and Software Agents

Additional problems appear when the web presentation is needed in more than one target format. Nowadays, mobile devices such as cell phones or PDA are able to access the world wide web; but often they require the pages to be present in a specialized content format, the Wireless Markup Language (WML). WML is similar to HTML, but tailored towards the requirements and limitations of mobile devices. For WML, an automatic translation from HTML may be feasible to some extent since it is not required to generate information that is not present in the original HTML pages in the first place.

However, the department may also want to provide information in a way that can be interpreted by software agents, for instance to support the automatic compilation of different kinds of course directories. The *Semantic Web Research Community Ontology* (SWRC) [114] and the Resource Description format (RDF) [71] together define a set of semantic annotations featuring concepts such as organizations, projects, and research topics. Figure 1.2 shows a sample HTML page that is annotated with a set of those machine readable elements. Web spiders can read and interpret the annotations, generating directories with specialized search- and browsing capabilities based on high level concepts and properties instead of plain keywords[113].

Here, identical information is repeated in regular HTML markup (lower

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
 <head><title>Stefan Decker</title></head>
 <body> <!--
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:swrc=
"http://www.semanticweb.org/ontologies/swrc-onto-2000-09-10.daml#"
           xmlns:a=
"https://www.daml.org/actionitems/actionitems-20000905.rdfs#">

    <swrc:AcademicStaff rdf:ID="person_stefan_decker">
      <swrc:name>Stefan Decker</swrc:name>
      <swrc:firstName>Stefan</swrc:firstName>
      <swrc:lastName>Decker</swrc:lastName>
      <swrc:email>stefan@db.stanford.edu</swrc:email>
      <swrc:phone>+1 650 723 1422</swrc:phone>
      <swrc:homepage
       >http://www-db.Stanford.EDU/~stefan/</swrc:homepage>
    </swrc:AcademicStaff>
  </rdf:RDF>   -->
  <p>
  <table border="0" width="75%">
   <tr>
    <td colspan="2"><b>Stefan Decker</b></td>
    <td rowspan="6"><img src="sde.gif" /></td>
   </tr><tr>
    <td><b>Email:</B></td>
    <td><a href="mailto:stefan@db.stanford.edu"
        >stefan@db.stanford.edu</a></td>
   </tr>
   <tr><td><b>Phone:</b></td><td>+1 650 - 723-1422</td></tr>
   <tr><td><b>Address (Work):</b></td>
         <td>Stanford University Gates Hall 4A, Room 425,
             Stanford, CA 94305-9040, USA</td></tr>
   </tr>
 </table>
 <!-- ... -->
 </body>
</html>
```

Figure 1.2: Excerpt from an SWRC-Annotated Web Page. Most of the information is duplicated in the machine readable upper part of the page and the traditional HTML code in the lower part, which may lead to maintenance and consistency issues.

part of figure 1.2) and the SWRC markup (upper part). In the sample page, the name, the email address, the phone number and the fax number are duplicated. Again, redundancy leads to additional effort, raising the probability of errors and inconsistencies. There are tools such as Ontomat [48] available to help users when annotating HTML pages, but this solves only a part of the problem. The burden of entering information twice (or even more often, depending on the desired content format support) and ensuring link consistency within a set of pages remains on the user [52, 53].

Thus, without a higher level of abstraction, the effort for creating the pages is multiplied with the number of supported formats.

## 1.2 The XML Toolchain

The recognition of the limits of HTML at the W3C led to the development of a more general format, the Extensible Markup Language (XML) [122]. HTML defines fixed document centric annotation elements such as "p" for paragraphs, and "h1" for top level headings. In contrast, the XML specification only defines a general tree syntax structure. The set of elements and nesting rules forming a concrete XML language must be defined separately in a Document Type Definition (DTD) [122] or XML Schema [123], defining an application specific markup language. Instead of being bound to the fixed set of HTML elements, custom elements such as "familyName" or "projects" can be used. XML is used internally by the W3C as the basis of all new content formats, and a new version of HTML named XHTML was reformulated in the terms of XML.

XML Schema is not only able to declare the elements that can be used in a conforming document, it also supports nesting rules and cardinality restrictions. XML Editors with support for XML Schema such as XOpus [125] are able to guide users to follow the restrictions of the schema, for instance by presenting lists of the elements allowed in a certain context, avoiding syntactical errors.

XML documents themselves cannot be displayed in current Web browsers. Web browsers do not "know" how to represent the new elements visually, so they are usually simply ignored. To transform XML documents to displayable HTML, the W3C has specified a declarative transformation lan-

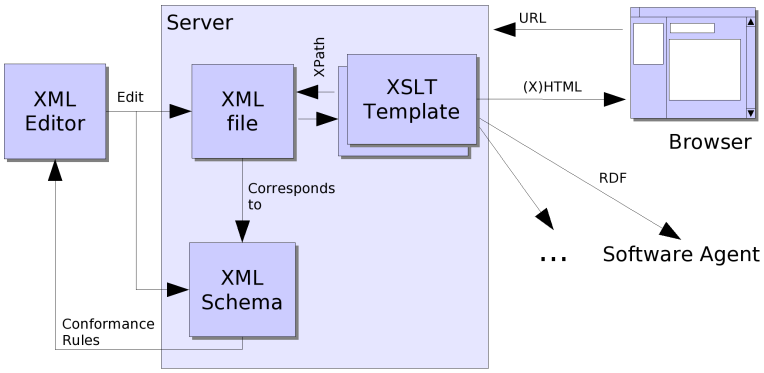


Figure 1.3: Simple XML Toolchain

guage, the *XML Stylesheet Language Transformations* (XSLT) [23]. XSLT is not only suitable for transformations from XML to HTML, but also for any kind of XML transformations. For that purpose, XSLT templates usually contain static fragments of the destination language, annotated with control elements, managing the insertion of content from the source format. XSLT utilizes the navigation language *XML Path Language* (XPath) [121] as a simple expression language for numerical and string operations, and to navigate the tree structure of the source document. XSLT is prooven to be computational complete [67], so it can be assumed that any desired destination format can be generated if the necessary information is contained in the source document. An XSLT style sheet can be applied at the client or server side, where the latter option avoids that every user needs an XSLT capable browser. Figure 1.3 depicts a possible chain of XML tools for the desired application.

Together, XML Schema and XSLT seem to provide the required means to solve the problems of the naive approach to building a Web site for an university department or other structured domains. XML Schema provides means to define domain specific elements such as “person” or “project” with specific sub-elements such as “name”, allowing a full separation of concerns. An XML editor is able to enforce consistency of an XML file with the rules of the schema. XSLT style sheets can be used to generate not only HTML but also different formats such as WML or even content suitable for the semantic web.

Actually, Manie et al. [76] suggest to use a generalized XML schema lan-

guage for semantic data modeling. For XML Schema itself, Murata et al. [86] provide a formal classification as a regular tree language, and Krumbein and Kudrass [115] provide a detailed analysis of mapping options from conceptual models to XML Schema. Unfortunately, Krumbein and Kundrass show that XML Schema is not able to model arbitrary associations between concepts without information loss. Hierarchical relationships in XML Schema cannot express cyclic associations such as the *person–publication–project* cycle in the university sample. The key references that can be expressed with XML Schema cannot guarantee a mutual reference between two element instances that take part in a bidirectional association: If a person is linked from a project, that does not ensure that the inverse link is set, too. Even if XML Schema would be extended to cover those cases, it does not seem very straight forward to use a language that was primarily designed to describe document tree structures for more general graphs, such as our domain model.

Another problem is that the XPath expression language that is used inside XSLT style sheets operates on XML instances only, but ignores valuable conceptual level information that is available in the corresponding XML Schema. XML queries are evaluated against XML documents based on element names and their syntactic nesting structure only, ignoring the element types and other conceptual level information [74].

## 1.3 Web Application Engineering

Problems associated with building Web applications were also recognized in the Software Engineering community, where a number of methodologies directly addressing hypermedia design have been developed. Early approaches such as the *Hypertext Design Method* (HDM) [42], the *Relationship Management Methodology* (RMM) [63], the *Object Oriented Hypertext Design Method* (OOHDM) [109], and the *Web Modeling Language* (WebML) [18] mainly focus on the development of new hypermedia-specific diagram types and the design process.

- The Hypertext Design Method (HDM) focuses on the introdcution of a new modeling language (HDL-HDM) that is especially tailored

towards hypermedia design. As design primitives, it uses entities with an hierachical inner structure and three different kinds of links. Typed links between different entities are called *application links*. The other link types are *perspective links* for different presentations of the same object (e.g in different languages), and *structural links* that are local to a single entity.

- The Relationship Management Methodology (RMM) builds upon HDM for its structural part, but shifts the focus more to the design process itself. The RMM process consists of seven steps: ER-design, slice-design, navigation design, conversion protocol, user interface design, and construction and testing. The slice design step determines how properties of entities are grouped logically. The navigation design specifies paths that are navigable, leading to a so-called Relationship Management Data model (RMDM).

- The Object Oriented Hypertext Design Method (OOHDM) is an object oriented extension of HDM. It introduces object oriented features such as inheritance, but also separetes several kinds of navigation diagrams from the underlying domain model.

- The Web Modeling Language (WebML) is a notation for specifying web sites at a conceptual level. WebML consists of a structural model for the data content, a compositional model for pages that compose the data model, a navigation model for the link topology, a presentation model for the layout and graphics and a personalization model for content delivery customization.

The methodologies have in common that they are based on a standard notation — such as E-R [20] or OMT [104] — for the conceptual design, and add their own notation for the following steps. More recent approaches such as Conallen [22] and Baumeister [8] feature standard extension mechanisms of the Unified Modeling Language (UML) [93] for this purpose. While Conallen mainly suggests a set of stereotypes for several web-specific artifacts such as frames or forms, Baumeister introduces a comprehensive navigational model along the lines of OOHDM, consisting of a navigation class model and a navigation structure model. He further uses object diagrams for a static presentation model and statecharts to describe the dynamic presentation.

Schattkowski and Lohmann point out that the overhead associated with heavyweight dedicated hypermedia development processes may be too expensive, in particular for smaller and medium projects [106]. Becoming familiar with the details of the process, working out the diagrams, and keeping them in a consistent state constitutes a significant amount of work — an effort that needs to be justified by savings in the overall development and maintnence costs.

The examples given in [109], [8], [59] suggest that navigation class diagrams are often nearly identical to the conceptual model, at least for smaller applications. With differences only in details, annotations in the class diagram itself would be better readable and would avoid the need for keeping a separate diagram consistent with the remainder of the specification.

To improve the effectiveness, Schattkowski and Lohmann propose a simplified process, focussing on formal parts. This way, they are able to tailor their process towards automated code generation, enabling rapid prototyping. In the remainder of this section, we will discuss different options for deriving an operational system from standard UML diagrams, including the code generation approach proposed by Schattkowski and Lohmann.

### 1.3.1 Code Generation

To address the lack of operability in existing methodologies, Schattowski and Lohman propose a rapid development approach, producing a working prototype of the application automatically [106]. Their ProGUM system is able to generate source code in the scripting language PHP [97], based on UML class diagrams, UML use case diagrams, and UML activity diagrams [73]. Source code is generated automatically for new parts of a model, changes in the model affecting existing code are highlighted by the system. Since PHP is computational complete, it can be assumed that it is possible to generate code for a web based maintenance interface that is able to fully support all constraints of the original model.

Requiring that the developer maintains code that once was generated seems difficult to avoid without a re-generation—possibly overwriting modifications. Here, a problem may be an inherent loss of abstraction. Let us assume that a relational database is generated from the model to hold instance information. Then, traversing a simple $n : m$ association involves

three tables: the table representing the source concept, a key match in the association table resulting from the database normalization process, and another lookup in the target table. General maintenance problems with scripting languages such as PHP [27] may be irrelevant if the code is purely generated, but not if the code must be actively maintained.

However, even in the simple case, when the code was not modified and can be re-generated, this approach is still not free of problems. Code generation introduces a delay between model change and model instance execution [103]. Generating code from models, compiling this code, shutting down the existing system, installing and configuring the new system, and starting it up can take from minutes to hours. Long turnaround times restrict the modeling options that can be explored.

### 1.3.2 Model to Model Transformation

An alternative to code generation is the transformation of the design model to an operational model. Here, in contrast to the transformation to executable code, the transformation result remains in a declarative form. Changes in a transformed model—such as an XML Schema generated from a class diagram—are simpler to port back to the source diagram automatically than arbritrary changes in generated source code. Source code generation leads to a distribution of model constraints to relational database tables and generated scripts, while a transformed model may be able to keep them together in one place, if sufficiently expressive. In their UWE approach, Kraus and Koch generate an XML Schema and XSLT stylesheets from an UML model. The instances conforming to the model are stored in an XML file [69]. Unfortunately, the problems discussed in section 1.2 apply: XML Schema is not able to represent cyclic and bidirectional associations and thus not sufficient to ensure consistency of the stored content with the underlying model.

### 1.3.3 Model Driven Architecture

In Model Driven Architecture (MDA) [111, 91], sufficient information to completely generate the target application must be included in a platform independent model (PIM). The PIM is then transformed into a platform

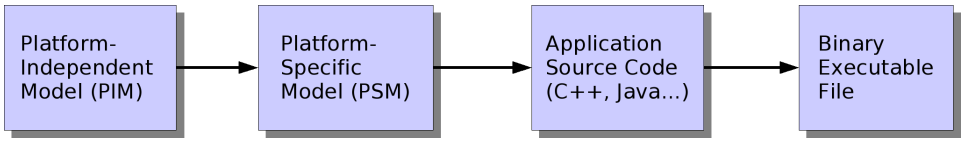


Figure 1.4: UML Model Driven Architecture (MDA)

specific model (PSM). The platform specific model includes specific implementation aspects of a target platform such as Java or .net. Since the platform independent model does not contain platform dependencies, it can be reused for different platforms. The platform specific models are used to generate the application source code. Finally, the source code is compiled, generating an executable object file. Figure 1.4 illustrates this process.

Melia et al. suggest to use the MDA in web software architectures [79]. Unfortunately, the automatic transformation from a PIM to a PSM is still an open issue [33], and the MDA approach combines some of the disadvantages of the compilation and transformation approaches by introducing another step of indirection. Heckel and Lohmann [57] criticize that the MDA does not pay enough attention to the functional evolution of a system. Every such evolutional step, for example a new requirement, triggers the whole transformation chain. Given that a large part of Web application development deals with the creative process of creating an appropriate user interface, involving the evaluation of small changes to HTML pages (or their equivalent in the model), time consuming transformations are likely to hamper the process.

In order to be able to fix problems in the model, information that makes it possible to trace them back needs to be preserved in all transformation steps. If the trace is not clear, or developers are more familiar with the language the application source code is generated in (e.g. Java or C++), they may be tempted to modify the generated code instead. Even if such a modification is meant only as a temporary fix, the introduction of a single inconsistency makes it no longer possible to run the generator chain completely. Especially when problems need to be fixed quickly under stress conditions, this may lead to increasing inconsistencies between the (modified) generated code and the model.

### 1.3.4 Direct Model Interpretation

In the knowledge management and rapid application development community, there are several approaches to directly interpret a conceptual model. Protégé-2000 [34, 44, 89] is a user interface for knowledge bases accessible via the *Open Knowledge Base Connection* interface (OKBC) [19]. Other systems such as Racer [46, 84] or FaCT[60] focus on efficient inference capabilities and finding the "right" tradeoff between expressiveness and runtime boundaries. Model interpretation approaches are not limited to AI Systems. Riehle [103], Mellor [80] and Frankel [41] mention UML virtual machines (VM) capable of interpreting arbitrary UML models.

Although those systems are not directly suitable for web applications, and there is no web engineering methodology based on the interpretation approach available yet, direct model interpretation seems to be capable of avoiding the problems of the code generation and model transformation approaches, and also to better address the short development cycles in web engineering [78].

## 1.4 Research Goal

The goal of this work is to provide conceptual and technical infrastructure that supports the direct interpretation of models as web based applications. The goal is not to provide yet another methodology, but to make existing ones more operational.

Web application development methodologies provide guidance to cleanly specify a web application. However, even for systems leading to a working prototype, significant maintenance problems remain. In the XML transformation case, the underlying tools are not sufficient to express the consistency constraints that may be identified in the original conceptual model. Code generation methods can result in more capable solutions, but lead to maintenance problems for the generated code itself. Model interpretation seems to be a viable alternative.

Since a conceptional model is a common property and starting point of all approaches, we will primarily focus on the technical advantages of a conceptual model:

- *The conceptual model is specific to a domain and not to a content format:* Alternative formats such as HTML or the Wireless Markup Language (WML)—designed for mobile devices—may be generated from the same content, stored in terms of the model.

- *The conceptual model is independent from the site layout:* When the content is not mixed with layout instructions, maintainers do not need to care about removing them by accident, introducing inconsistencies. People with different responsibilities can work on different entities.

- *The conceptual model provides constraints that can be exploited to ensure consistency:* The associations in the model constrain links to the endpoints of the associations. This makes it possible to reduce the selection options in a user interface to instances of the relevant concepts, guiding the user. Typed fields make it possible to provide further assistance, such as displaying a calendar when a date fields is edited. The consistency of bidirectional links can be ensured by the model.

While itself not sufficient for the desired purpose, the XML toolchain (Figure 1.3) consists of a coherent set of standardized technologies and avoids the problems of code generation. Hence, it may provide a us with a guideline for the architecture of the desired model runtime environment:

I. XML Schema needs to be replaced by a suitable conceptual modeling language.

II. The XML editor needs to be replaced by an editor for

   (a) the schema and

   (b) the content.

   The editor should respect the constraints of the (meta-) model. It should enable content maintainers to focus on their original task instead of being distracted by layout instructions or aspects that can easily handled by the system—such as ensuring bidirectional link consistency. In the XML toolchain, there is only one editor for both purposes, but that is not a necessity.

III. At least the XPath part of XSLT needs to be replaced by a query mechanism that provides full and high level access to the content of the system. The transformation mechanism should be able to access the stored information at an appropriate level of abstraction and not restrict the formats that can be generated.

There are some additional issues in real world web applications that are not considered in the idealized XML toolchain but are addressed by additional tools or mechanisms. With a more powerful modeling framework, we should consider whether those issues can be addressed in a way consistent with the remainder of the system:

IV. In the case of plain XML files, security concerns are covered only at file level by the web server. Although security is not the main focus of this work, it may make sense to address security at a more fine-grained level.

V. In many real-world systems, access to legacy databases will be a strong requirement. Tables can be converted to XML files on the fly or in batch jobs, making them accessible to XML tools. Similar means should be provided for our chosen high-level representation.

VI. The completeness of XSLT makes it possible to generate any desireable transformation of the content. However, XSLT does not provide means to manipulate the content permanently, except from a complete rewrite. It makes sense to examine to which extent modifying operations such as "delete users that did not log in for two years" or other actions, possibly triggered via the user interface, are feasible.

## 1.5 Outline

Designing a web system that directly interprets a conceptual model is a significant amount of interesting work. The next chapter will examine the first point, finding the right foundation, and discuss alternative conceptual modeling options for the underlying representation, namely Description Logics and UML class diagrams.

The following chapters will then show how an XML publishing toolchain can be replaced at a high level of abstraction, utilizing many of the UML 2.0 provisions for the Model Driven Architecture. The Infolayer, a system implemented to show the feasibility of concepts developed in this dissertation, is sketched in chapter 3. It is mainly based on the direct interpretation of a UML class diagram. Chapter 4 discusses in detail which UML modeling elements are supported and how they are interpreted by the system.

The Infolayer system supports the Object Constraint Language (OCL), a constraint language that is part of the UML. In the system, OCL is not only used as constraint language, but also to specify queries and to define query operations. The Infolayer OCL support is described in detail in chapter 5. For the support of the business logic of Infolayer applications, besides the OCL query methods, also operations with side effects may be required. For this purpose, a simple surface language for UML action semantics is supported by the system. The action semantics surface language is described in chapter 6.

The Infolayer counterpart to XSLT, an XML-based template mechanism for the generation of arbitrary XML code is described in chapter 7. Relying on OCL instead of XPath makes it possible to generate arbitrary XML code directly from the system state of an object oriented model, without first generating canonical XML and then applying XSLT.

Technical details of the default XML persistency mechanism of the Infolayer and alternative SQL database connections are described in chapter 8. Chapter 9 describes some case studies that go beyond the simple university example discussed here, including some third party extensions of the Infolayer system.

Finally, chapter 10 provides a summary of this work and an outlook to possible improvements and extensions.

# Chapter 2

# Formal Basis

For the interpretation of a conceptual model as a web application, an appropriate conceptual modeling framework must be chosen. Most recent web engineering methodologies presented in the previous chapter are based on UML, but UML is not the only option.

The W3C has recently recommended its own conceptual modeling language, the Web Ontology Language (OWL)[124], for the Semantic Web. The Semantic Web is an effort to make the content of web pages understandable for computers [118], like in the SWRC case presented in the previous chapter. The general idea is to use special semantic annotations such as "author", "name", or "ownedBy" instead of plain unstructured text. In the Semantic Web terminology, the formal agreement about a common vocabulary, available in a machine readable format, is called the domain ontology. Here, an ontology is a conceptual model with a special focus on the disambiguation for information interchange [45]. The ontology usually defines concepts, a specialization hierarchy, associations and attributes important for information interchange in a particular domain. For the semantic annotations corresponding to a given Ontology, an XML based format called *Resource Description Format* (RDF) [71] has been developed by the W3C. The Ontology Modeling Language is based on Description Logics, which can be seen as the state of the art in conceptual modeling in the Artificial Intelligence community.

In this chapter, we will first give a short overview of UML class diagrams and Description Logics and their general approaches to modeling, and then

compare several aspects that seem especially important for web applications:

- *Semantics:* Without precisely defined semantics, it is difficult to build a runtime system interpreting a model.
- *Query Capabilities:* To transform information stored in terms of the model to formats suitable for browsers and software agents, it should be possible to access any part of the content.
- *Extensibility and Integration:* Although only conceptual models are considered here, it should be easy to add functionality relevant in the web context—such as user registration or notification mechanisms—in a consistent way. Also, it should be possible to integrate existing resources such as relational databases.
- *Ease of Use:* For the development of small applications, it is desireable that developers can work with familiar tools instead of needing to learn new formalisms.

## 2.1 Description Logics

Before Description Logics (DL) was developed, the Artificial Intelligence community mainly used first order Predicate Logics and Frame systems for knowledge representation. The semantics of First Order Predicate Logic is well understood, but Predicate Logics is undecidable and larger rule based systems tend to become difficult to oversee and to manage. To avoid the latter problems, Frames systems consisting of graphical boxes, labels and arrows were suggested by Marvin Minsky [83], shifting the focus to intuitivity. However, while humans are easily able to get the meaning of boxes with labels and arrows, this task cannot be accomplished by computers if the symbols do not have a precise meaning, and critics pointed out that the boxes and arrow approach did not mean much even for humans when the English labels are not well-known words [56].

Description Logics have been developed based on KL-ONE [12] to combine the advantages of both approaches, the intiutivity of frame systems and the precise semantics of predicate logic, while also improving decidability by limiting expressability.

DL systems support the definition of concepts by simply naming them and specifying where they fit in the generalization/specialization hierarchy of existing concepts. New concepts can also be defined in terms of existing concepts using the operations of *concept conjunction*: the *and* operator can be used to specify that the new concept is a common specialization of a number of other concepts. New *roles* may be introduced to represent possible relationships that may hold between individuals in the domain being modelled, and concept definitions may include restrictions on the possible values, number of values, or type of values that a role may have for the concept being defined.

## 2.2 UML Class Diagrams

Outside the AI community, the modeling formalism of choice is the Unified Modeling Language (UML) [93]. The UML has been developed to unify several modeling approaches for object oriented systems and is maintained by the Object Management Group (OMG) [90].

For our purpose, the interesting part of UML are class diagrams. UML class diagrams can be seen as extended Entity-Relationsip diagrams [20]. UML classes correspond to entities, associations to relations. In addition to modeling classes, associations, and attributes, UML class diagrams cover the specification of operations and a specialization relationships between classes. Classes are depicted as boxes containing the class name and lists of attributes and operations. Associations are depicted as lines connecting the boxes. In contrast to description logics, association ends are not defined globally, but local to the declaring classes.

The use of UML is now widespread in industry and its rapid acceptance (even for the design of mission-critical applications) suggests that it provides an effective and scalable approach to conceptual modeling. Steven Cranefield has suggested to use UML class diagrams also for ontology modeling [28, 29]. Baclawski et al. even propose to add special modeling constructs for DL-like property-centric modeling [7].

The purely graphical UML diagrams are usually not detailed enough to provide all the relevant aspects of a specification. While it would be possible to express additional constraints in natural language, this approach often

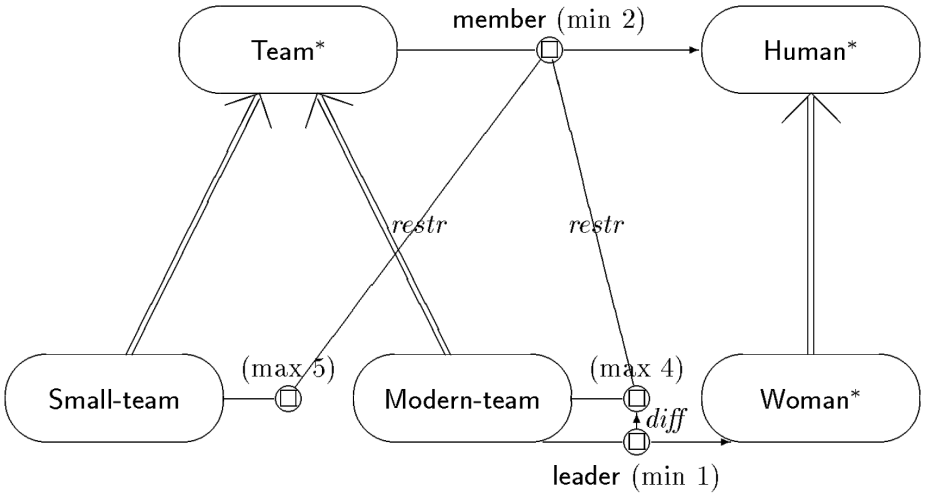


Figure 2.1: Description Logic Example from [88, page 104]

leads to ambiguities. In order to write unambiguous constraints, formal languages such as Z [112] can be used. The disadvantage of traditional formal languages is that they are difficult to use for persons without a mathematical background.

An associated constraint language has been developed to fill this gap. The Object Constraint Language (OCL) [93, chapter 6] it is a formal language that is easy to read and write. It can be used to assert arbitrary constraints on the possible instances of a model. OCL is a pure expression language, and OCL expressions are required to be without side effects.

Here, we will use the abbreviation "UML" as short form for UML class diagrams with OCL annotations, unless otherwise noted.

## 2.3 General Modeling Approach

Description Logic and UML class diagrams both use a graphical notation to describe concepts and relations between them. However, there are significant differences between the general approaches to modeling chosen in both frameworks.

In Description Logics, concepts are defined by a set of properties an instance

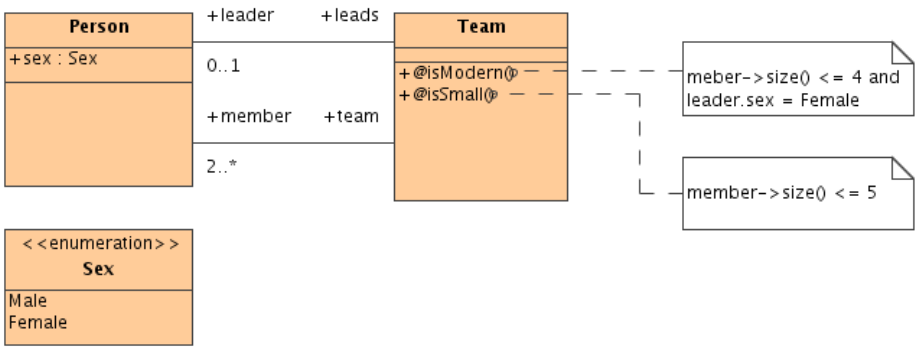


Figure 2.2: UML Adoption of the DL Team Sample

*must* have in order to be a member of this concept. Concepts may be atomic root concepts, or inherit requirements from other concepts. Figure 2.1, taken from [88], shows a small sample, modeling different types of teams in Description Logic. In the example, a team is defined as something that has at least two (human) members. A small team is a team which has at most 5 members. A modern team is a small team where the leader is a woman. The set of objects belonging to a concept is defined intensionally; the set is derived dynamically from the concept definition.

In object oriented modeling, concepts are described as the set of properties instances *may* have. In description logics, properties are globally defined, any property may be assigned to any instance as long as it is not explicitly excluded. Thus, it would be possible to assign a leader to a team, although the leader role is used only in the context of the modern team. Based on the member and leader information, the system could automatically infer that a certain instance of the concept *Team* is a actually a modern team. In UML, object properties are limited to those explicitly declared for a class and its super-classes. The class an object belongs to must be explicitly specified at the creation time of an object.

Because of those differences, it is not possible to build an identical UML Model for the above DL example. In object oriented systems, it is usually necessary to explicitly specify which properties are calculated, and which ones can be filled by the user of the system. Figure 2.2 shows a possible adoption to UML, where the different team types are represented as in-

ferred attributes of the class *Team* instead of separate concepts—like in the DL model. The comment boxes attached to the attributes contain OCL specifications of the values.

An advantage of the DL approach is that inferences are not limited to those explicitly specified. For instance, from the information that $T$ is a modern team and $L$ is the leader of the team, the system might be able to automatically infer that $L$ is a Woman [88, page 104]. From the information that $T$ is a small team and the leader of the team is a woman, the system could infer that the team is actually a modern team. The main disadvantage of this approach is that there are only global runtime and expressability boundaries, and finding the "right" trade-of between DL expressiveness and runtime limits has become its own branch of AI research.

## 2.4 Semantics

It makes very much sense that the semantics of a modeling formalism are well-defined, so there are no doubts about the "meaning" of a model. The users should be able to get a clear picture of what they are actually doing. Moreover, while humans are good at guessing or implying some kind of "common sense" meaning, for a runtime environment this is certainly not an option. Thus, the semantics of the modeling formalism should be well-defined.

The semantics of a formalism is usually described using a mapping to a *semantic domain*, a language for which the semantics are well understood [105]. For Description Logics, usually a well-defined mapping to a decidable subset of predicate logics exists.

The UML features a four-layered modeling approach. Layer 0 contains user objects such as "Team A". Layer 1 contains the user model, that is the set of classes, attributes, associations, etc., as defined by the user, describing the objects in layer 0. Layer 2 contains UML concepts such as class, association, and operation that are used in the user model. The concepts of layer 3 are defined in terms of layer 4, the meta object facility (MOF), which happens to be a subset of layer 3. Layer 4 is described in itself.

The semantics of the upper layers of UML were described in plain text and scattered over the whole standard documentation in UML 1. Several

proposals for defining precise semantics can be found in the literature, for instance a system model suggested by Ruth Breu et al [13] or mappings to the specification language Z [112, 35], or an integration of Fusion modeling techniques and Z [15].

The adopted OCL proposal [10] for version 2 of the UML standard follows the UML meta-model based approach, where the semantics of large parts of OCL and UML are described using a "core" subset of UML. The formal static semantics of the core parts are defined by a set-theoretical *object model*, based on work by M. Richters [102]. However, despite the significant progress made with the OCL 2.0 proposal, there are still areas in UML that lack a proper semantic definition, for example the precise meaning of "Aggregations" [58].

In his dissertation, Richters defines an object model as a tuple

$\mathcal{M} = (CLASS, ATT, OP, ASSOC, associates, roles, multipicities, \prec)$

The set of all Types $T$ consists of the set of types of the classes $CLASS$, the primitive types *Integer*, *Real*, *Boolean*, and *String*, and collection types representing sets, bags, sequences, and ordered sets of types. For naming model components, we assume a set of finite, non-empty names $\mathcal{N}$.

### Signature

**CLASS** $\subset \mathcal{N}$ is the set of classes, a finite set of names.

**ATT** is a set of signatures $a : t_c \rightarrow t$ where the attribute name $a$ is an element of $\mathcal{N}$, $t_c$ an element of $CLASS$ and $t \in T$ a type. Cardinality constraints other than one can be modeled by using set types.

**OP** Operations are defined as a set of signatures $w : t_c \times t_1 \times ... \times t_n \rightarrow t$, where the operation name $w$ is an element of $\mathcal{N}$, $t_c$ an element of $CLASS$, and $t_1 ... t_n \in T$ types.

**ASSOC** is a finite set of association names $ASSOC \subset \mathcal{N}$.

**associates** is a function $ASSOC \rightarrow CLASS^+$, mapping the association names to a tuple of two or more classes.

**roles** is a function $ASSOC \rightarrow \mathcal{N}^+$, mapping the association names to a tuple of two or more role names.

**multiplicities** is a function assigning each of the participating classes a multiplicity constraint which is a non-empty set of positive integers $\mathbb{N}$.

$\prec$ is a partial order on the set of classes $CLASS$. Pairs in $\prec$ describe the generalization relationship between two classes.

**Semantics** A system state if formally defined as

$\sigma(\mathcal{M}) = (\sigma_{CLASS}, \sigma_{ATT}, \sigma_{ASSOC})$

where finite sets $\sigma_{CLASS}(c)$ contain all objects of a class $c$ existing in the system state, functions $\sigma_{ATT}$ assign attribute values to each object, and the finite sets $\sigma_{CLASS}(c)$ contain links connecting objects.

Richters also defines precise semantics for OCL expressions. The object model does not cover the whole UML. Concepts that are not required in the UML meta model like visibility constraints are left out; the subset covers only concepts that are used in the UML specification to describe UML itself. For a detailed discussion of precise static UML and OCL semantics, please refer to [102].

## 2.5 Query Capabilities

Unfortunately, the query capabilities of the built in reasoning mechanism of Description Logics are limited. According to Alex Borgida, the constructors usually considered in the DL literature are exactly as expressive as the predicates defined by the subset of first order predicate calculus with monadic and dyadic predicates which allow only three variable symbols [11]. Even the full description logic $\mathcal{DL}$ cannot express the "conjunctive queries"—the least powerful query language considered in the relational database literature. For instance, if we add a role "father" to the team sample, it is not possible to formulate the following query in terms of Description Logics:

```
Team(T), Peson(P), Person(Q), Person(R),
  father(P, Q), father(Q, R), member(T, P), member(T, R)
```

Thus, one is very likely to need an additional query language when designing XML templates for a system based on Description Logics.

Although OCL was originally designed as constraint language, it can be used as a query language as well. For instance, an OCL equivalent of the above query taking advantage of path expressions is:

```
Team.allInstances()->select(t
   | t.member.father.father.team->includes(t))
```

Mandel and Cengarle have analyzed the completeness of OCL as a query language [75] in the sense defined by Ullman [116]. Union, difference and selection can be expressed directly as primitive OCL collection operations. Projection is only possible for just one attribute, and the cartesian product cannot be expressed, both due to the lack of tuple types. As a solution, Mandel and Cengarle have suggested the addition of tuple types to OCL. Accordingly, tuple types were included in the accepted proposal for the version 2.0 of OCL.

Richters points out that when allowing the implementation of query methods in OCL, the computational power of OCL is extended to the power of recursive functions for the price of the termination guarantee of expressions [102, page 112]. We will prove the computational completeness of OCL in chapter 5.

## 2.6 Extensibility and Integration

While the semantics of DL systems seem more mature than UML semantics, they are limited to a static world. It also seems difficult to integrate other techniques elegantly into the formalism, since there are no standardized "plugs" for extensions.

UML provides different means to describe the dynamic aspects of a system. OCL provides means for the precise description of the post conditions of method invocations. UML state chart diagrams can be used to describe the life cycle of objects. Moreover, the UML Action Semantics introduced with UML 1.5 [93, Chapter 2] allows the specification of arbitrary computations.

For custom extensions, the UML provides three different standard interfaces:

- *Stereotypes* can be used to further define an element in a UML diagram.

- *Tagged values* can be used to enrich all kinds of model elements with user defined annotations in the form of attribute–value pairs.
- *Operations* can be used to hide any kind of "blackbox" system, where OCL post conditions can be used to describe the invocation consequences that can be covered by the UML.

## 2.7 Ease of Use

DL based knowledge representation systems offer several inference services, helping the user to avoid building inconsistent or unsatisfiable models [16]:

- *Concept Satisfiability*: Does a concept $C$ have a non-empty set of possible instances? Concepts that are unsatisfiable are usually not intended in the design process.
- *Subsumption*: Is a given concept description more general or more specific than another, or can no such relation be established? This enables the system to show the user how a newly created concept relates to other existing ones.
- *Knowledge-Base Satisfiability*: Are the model and the set of recorded instances consistent with each other? This check can be used to validate a model using sample instance data.
- *Instance Checking*: Is $a$ an instance of concept $C$ in any model of the knowledge base?

All that DL services handle the overall knowledge base. It is a "global" model where each concept may have an effect on other concept definitions. UML does not offer this kind of inference services. In UML, most model elements are localized in scope. This feature of design locality helps the model designer to construct a more complex design in manageable "chunks", whose complexity is encapsulated within the scope of a class or a few classes.

Another important aspect is that UML is widely used in the industry and taught in Universities. Together with wide-spread use comes tool support; the UML is supported by a large number of sophisticated graphical tools such as Rational Rose, Magic Draw, Argo UML and others.

## 2.8 Conclusion

Both options, DL and UML class diagrams provide the necessary means to model a domain in terms of concepts and associations between the concepts. While DL provides mature semantics and a more intensional formulation of knowledge, UML generally seems to be more modular and also more popular, enabling people to build on existing knowledge and expectations, and to work with familiar tools. Taking the previous knowledge of users into account, UML will probably be simpler to work with for most users.

Concerning completeness of the formalism for the given task, both, DL and UML, provide means for adding and removing instances and links. However, the built in query capabilities of Description Logics seem insufficient for a sophisticated template mechanism, and there is currently no widely accepted query language available for Description Logics. While UML provides well-defined annotation and extension mechanisms that are fully supported by UML tools, this kind of standardized interfaces does not exist for Description Logics.

For those reasons, we have chosen UML class diagrams as the basis for our work on an interpretative approach to the model-driven development of web applications.

# Chapter 3

# The Infolayer System

In the introduction, we have identified a set of primary action points for building a system that resembles the coherence of the XML toolchain, but at a higher level of abstraction:

I. Replace XML Schema with a high-level conceptual modeling language.

II. Replace the XML editor with an editor that automatically respects the constraints of the chosen modeling language.

III. Replace XSLT with a language that is coherent with the remainder of the system.

We already have identified UML class diagrams as a modeling language suitable for our purpose in the previous chapter, addressing issue I. Here, we will address issues II and III.

In the XML toolchain, the XML editor serves two purposes, editing the schema and editing the system content. The schema editing part of issue II is sufficiently addressed by existing UML based CASE tools. In order to cover the remaining part of issue II (that is not covered by UML tools), we have developed a runtime environment that is able to interpret a class diagrams as a web applications. The runtime environment takes a class diagram in a standardized format (XML Metadata Interchange (XMI) [92]) as input. It uses the file to dynamically generate a browser interface that is based on HTML forms, taking the constraints of the model into account.

The generic HTML interface is described in section 3.1. The class diagram is also used to derive a simple default storage format that can be used to make the system state persistent.

In the XML toolchain, transformation rules to other XML formats directly accessible to clients can be specified in XSLT. In section 3.2, we address issue III with a similar transformation mechanism. The main difference of our approach is that we are using OCL to access the parts of the system state that are needed when applying the templates. The utilization of OCL helps us to keep the transformation rules coherent with the remainder of the system.

In the introduction, we have identified three additional issues that are not addressed in the XML toolchain, but are nevertheless important for real-life applications.

IV. Security Issues

V. Integration of legacy data

VI. Completeness

Those issues are considered briefly in sections 3.3–3.5, mainly pointing to separate chapters for a more detailed discussion.

All our proposed solutions have been implemented in the Infolayer system. By using this system in a number of real world applications (described in chapter 9), we were able to make sure that no important aspect of web applications was left unaddressed. Section 3.6 describes the architecture of this system. Section 3.7 shows the look and feel of an Infolayer sample application that results from the UML diagram presented in the introduction.

At the end of this chapter, we discuss means to keep the semantics of the implemented system consistent with the UML specifications.

## 3.1 Interpreting the Class Diagram as a Web Application

In the Infolayer system, a UML class diagram controls two main aspects of the web server: The HTML based default user interface, and a simple

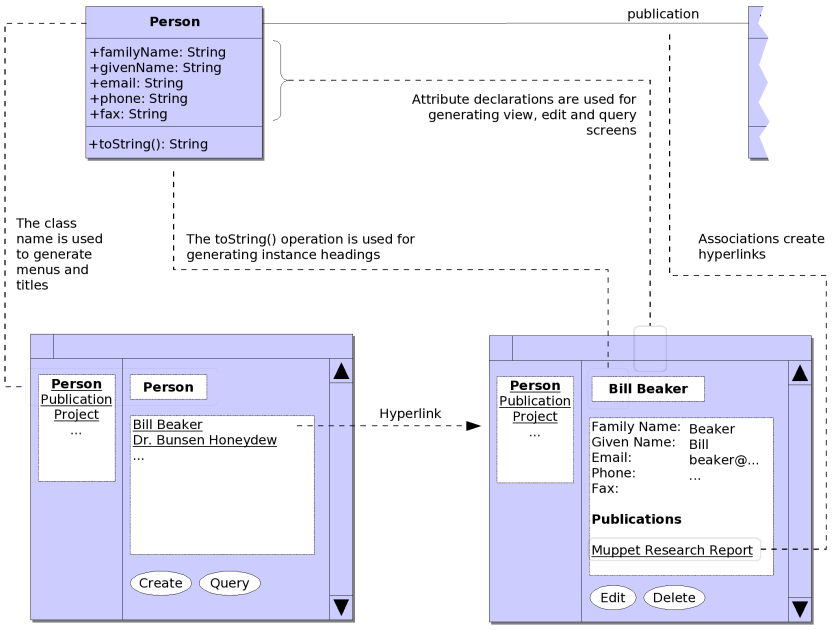


Figure 3.1: HTML Generation from the Class Diagram

XML-based persistent storage.

In addition to those primary aspects of the system, the class diagram is also used to derive a SOAP interface [24, 25], sketched in chapter 9, and a simple command line interface. SOAP is a remote procedure invocation protocol that enables third party applications to access the content of the Infolayer system in a machine readable manner. The command line interface is accessible via a telnet socket connection or an HTML form based interface and makes it possible to evaluate arbitrary OCL expressions for debugging and testing purposes.

While we focus on UML class diagrams here, the general approach of model interpretation is not limited to a single diagram type. For instance, an extension of the Infolayer system uses statechart diagrams to manage the state and life cycle of objects, including means to trigger state transitions via the web interface [100, 55].

### 3.1.1 HTML Generation

In the default case, the HTML pages generated by the Infolayer system are divided into two areas, a global navigation area to the left, taking about 1/5 of the screen, and a detail area. The navigation area is identical for all pages and contains a tree of class names, representing the classes and their generalization structure. The classes are annotated with a hyperlink, leading to a new page, the default page of the class.

A default class page has the address */auto.html?self=< classname >*. It shows

- the name of the class as the title,
- the Documentation of the class, extracted from the XMI file,
- the set of all instances of the class (excluding instances of subclasses), and
- a set of buttons corresponding to parameterless static operations available for the class. By default, a "create"-Button that triggers the creation of new instances and a "query"-button which leads to a query page are displayed.

The listed instances are hyperlinked with instance detail pages. An instance detail page shows a table of all attributes, associations and methods of an instance, including the corresponding values. For associations, the listed instances are again hyperlinks, leading to the detail page of the instance at the other end of the association. This way, it is possible to "click through" the whole system content conveniently.

All objects have stable URLs, so it is possible to create bookmarks for the generated pages. For modifying instances, a HTML page similar to the instance detail—but containing form elements for manipulating attributes and associations—is created. For primitive types, simple input fields are used. For associations, selection lists make it possible to pick instances conforming to the type of the association. Figure 3.1 illustrates the automatic interface generation.

### 3.1.2 Persistent Storage

By default, all instances managed by the Infolayer system are stored in two simple XML files. The primary file (instances.xml) stores a snapshot of all instances at a point of time. The secondary file (instances.chg) is used to store changes relative to the primary file. The current set of instances and properties is determined by the primary file plus the changes denoted by the secondary file. When the changes exceed a certain limit at system startup, the changes are incorporated into the main instance file, and the secondary file is discarded.

The XML files contain XML elements for all instances, where the element name corresponds to the class name. The unique id of an instance is stored in an *id* attribute. Elements corresponding to the properties of an instance are embedded in the instance XML element. Primitive types are stored as textual content of the property elements. References to objects are stored in an *idref* attribute. Binary content is stored in separate files. The persistent storage is covered in more detail in chapter 8.

This simple persistence mechanism is only a simple default option if no explicit link to a relational database is provided. Connections to relational databases are discussed in section 3.4.

## 3.2 Arbitrary XML Generation

The Infolayer system provides three different options to influence the look and feel and the generated XML code. The simplest option are Cascaded Style Sheets (CSS) [72]. CSS support is not an Infolayer specific feature, but a general feature of newer HTML versions. Additional formatting and visibility options can be influenced using Infolayer specific model annotations, as described in appendix C.

The most powerful customization mechanism provided by the Infolayer system is an XML Templates engine, enabling the generation of any XML based format. The templates consist of regular elements of the target format, enriched with three types of template elements:

- Conditionals, Loops and include elements, similar to those provided

by the XML transformation language XSLT [23], except that OCL is used as expression language instead of XPath [121].

- Elements providing views and editors for UML constructs such as properties or operations. Those elements provide simple means to generate complex constructs of the target language, such as components for editing associations or uploading binary content.
- Elements for special purposes such as search or login buttons.

Embedding the template elements is similar to other template mechanisms, such as XSLT. Before delivering pages to the client, the template elements are evaluated and replaced with code of the target language. The main difference to XSLT is that the Object Constraint Language (OCL) is used as a query language in the templates. While other languages would have been possible (and actually have been used in the system before), the OCL naturally fits with the UML-based modeling approach. The OCL capabilities of the Infolayer system are described in detail in chapter 5.

The templates follow an object-oriented schema: Every class can be assigned several templates, e.g. "view" and "edit", and the templates are inherited by subclasses. Thus, in the most simple case, the definition of a template for the base class "Object" can influence the appearance of the whole system. Actually, this mechanism is used to generate the HTML interface described above; the HTML interface is completely implemented using the template mechanism. Template inheritance makes it possible to provide a single generic default user interface for all classes, without constraining customization options.

Figure 3.2 compares the XML toolchain to the corresponding concepts of the Infolayer system. The template mechanism is described in detail in chapter 7.

## 3.3 Security Concerns

Since the Infolayer provides means for editing instances remotely, some kind of security mechanism is required in order to limit write access to authorized persons. In some case, even more fine grained access rights may be necessary,

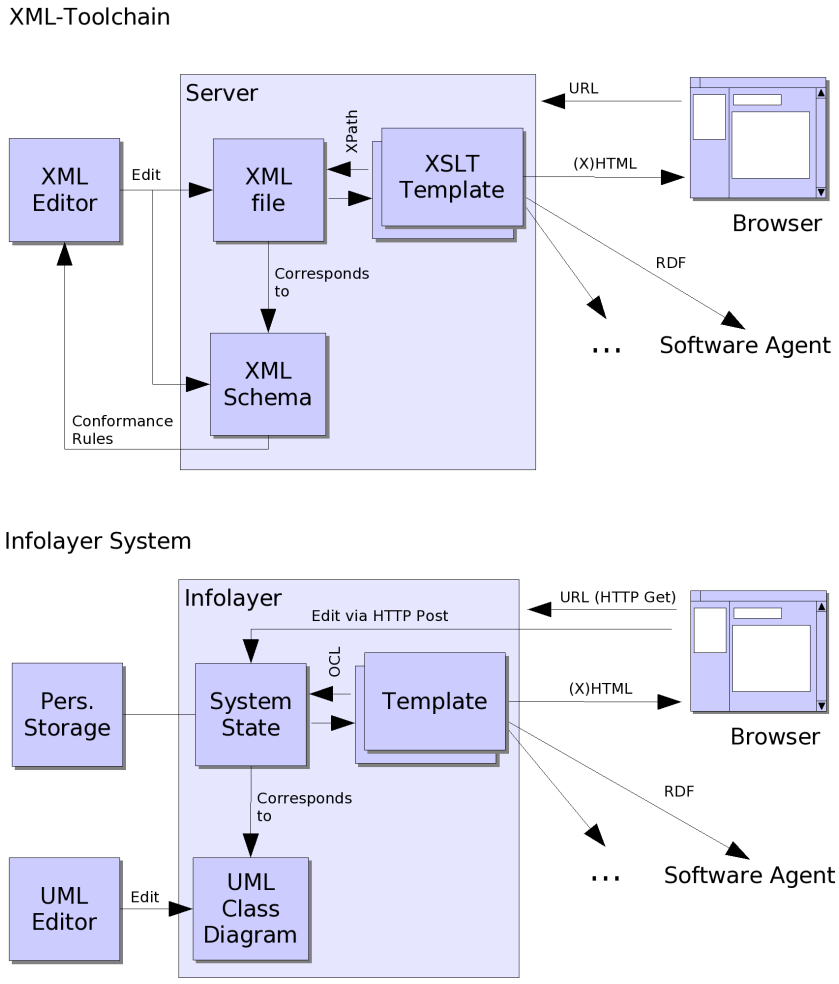


Figure 3.2: The XML Toolchain and the Corresponding Building Blocks of the Infolayer System

for example to limit read access, too, or if different groups of maintainers shall have access only to the content they are responsible for.

To address the different needs, the Infolayer system provides four different mechanisms to restrict access to the content stored in the system:

- General visibility of classes and properties, defined in the UML diagram
- User dependent access constraints for classes, properties and operations
- User dependent customization in XML Templates and
- URL based access restrictions

The default setting allows anybody to see all classes and properties that are marked public in the UML diagram, while for editing content, a user must be registered with the system.

Please note that all access restrictions are enforced by the user interface; operation implementations or page templates are not implicitly bound to the constraints.

### 3.3.1 UML Element Visibility

The simplest and least flexible mechanism for access control is the UML element visibility as defined in the UML model.

The element visibility is used to determine whether an element is displayed in the user interface automatically. If a class, attribute, association end, or query operation is marked *public* in the UML model, it will be displayed, otherwise not.

This default behavior can be overwritten by specifying explicit visibility rules, described in the following sections.

### 3.3.2 User and Dynamic Access Management

In order to determine user access rights, the Infolayer system needs a database of registered users and their properties. For this purpose, it uses a specialized class named *User*. By setting the tagged value *il-userclass* of the model or the class *Infolayer*, the Infolayer system can be instructed to consider an alternative class as the user class.

To restrict the access for classes, properties, and operations to certain users, the Infolayer system recognizes specialized model annotations. Both, the user class and the model annotations for access restrictions are described in more detail in the next chapter.

### 3.3.3 Customization in XML Templates and URL Based Rules

Customization and access limitations depending on the current user can also performed in the XML template mechanism. In contrast to the model annotations, which have a global scope, the XML templates enable different access restrictions in different situations.

The template based XML generation and URL based access rules are described in detail in chapter 7.

## 3.4 Connections to Legacy Databases

In addition to the default XML files, it is possible to use a number of other options such as CSV, BibTex or relational database tables as persistence mechanism. For this purpose, the model must be annotated with additional information describing the database connection. Moreover, for mapping associations, explicit rules must be provided since none of the supported storage mechanisms supports associations as first class entities. The alternative persistence options are described in detail in chapter 8.

## 3.5 Completeness

In the previous chapter, we have already mentioned that OCL, the expression language associated with UML, is computationally complete. We will handle OCL in more detail and prove this claim in chapter 5.

Since OCL is free of side effects, we cannot use OCL to manipulate the current content of the system, to create new instances, or to delete existing instances.

However, in contrast to OCL, existing database manipulation languages normally have capabilities beyond queries. For instance, Ozkaran asserts the need for more capabilities in his definition of completeness [95]:

> In language implementations, the following two operations are needed to assure relational completeness:
>
> 1. The ability to represent assignments, that is, the ability to create new relations to store the results of relational algebra operations that are also relations. [...]
> 2. The ability to compute transitive closures which enables recursion and/or nesting of relational algebra operations to express expressions of arbitrary complexity. [...]

While the ability to compute transitive closures is subsumed by Turing completeness, we are not able to formulate assignments or model manipulations in OCL. With assignments, OCL would no longer be free of side effects, so we cannot simply add an assignment operator to OCL. Without assignments, our system is suitable for pure information services, but not for web applications such as online shops that require program controlled changes of the system state.

Actually, the first point of Ozkarans notation of completeness can be split into manipulation of the data level (M0), that is the manipulation of data in existing relations or instances, and manipulation of the schema level (M1), that are manipulations that have an effect on the system tables, such as the creation of new tables or classes.

In order to be able to precisely specify the semantics of operations with side effects, UML contains an action semantics specification. In chapter 6,

we show how action semantics can be integrated into the Infolayer system to achieve program controlled data manipulation, extending the range of possible applications beyond the pure XML toolchain. In the next chapter, we will present the Infolayer meta-model that could be used for modifications at the schema level if we would extend the implementation to support write access to the user model.

## 3.6 Infolayer System Architecture

The architecture of the system, which is depicted in figure 3.3, consists of the following main components:

- The core of the Infolayer system is the UML Runtime Environment including the OCL and action semantics parser and interpreter. It holds a memory representation of the complete model and the instances that are currently being processed. The memory representation of the model is generated by the XMI loader. Instance data is requested from the persistence layer on demand. The UML capabilities of the Infolayer system are described in detail in chapter 4.

- The XMI loader loads the UML model from an XML Metadata Interachange (XMI) file and transforms it into the memory representation that is needed by the UML Runtime Environment. Using the standardized model interchange format XMI makes it possible to utilize existing CASE tools for conceptual modeling.

- The storage abstraction provides a unique interface to data that is stored persistently in different formats or at different locations. It makes the core of the system independent from the underlying data store. Currently, the following data sources are supported: XML files, JDBC/SQL, Bibtex (used for literature databases), and CSV (comma separated value) files.

- The Telnet server provides a simple command line interface to the Runtime Environment, mainly for testing and debugging purposes. OCL queries entered by the user are handed over to the core, where they are evaluated. The results are returned in textual form.

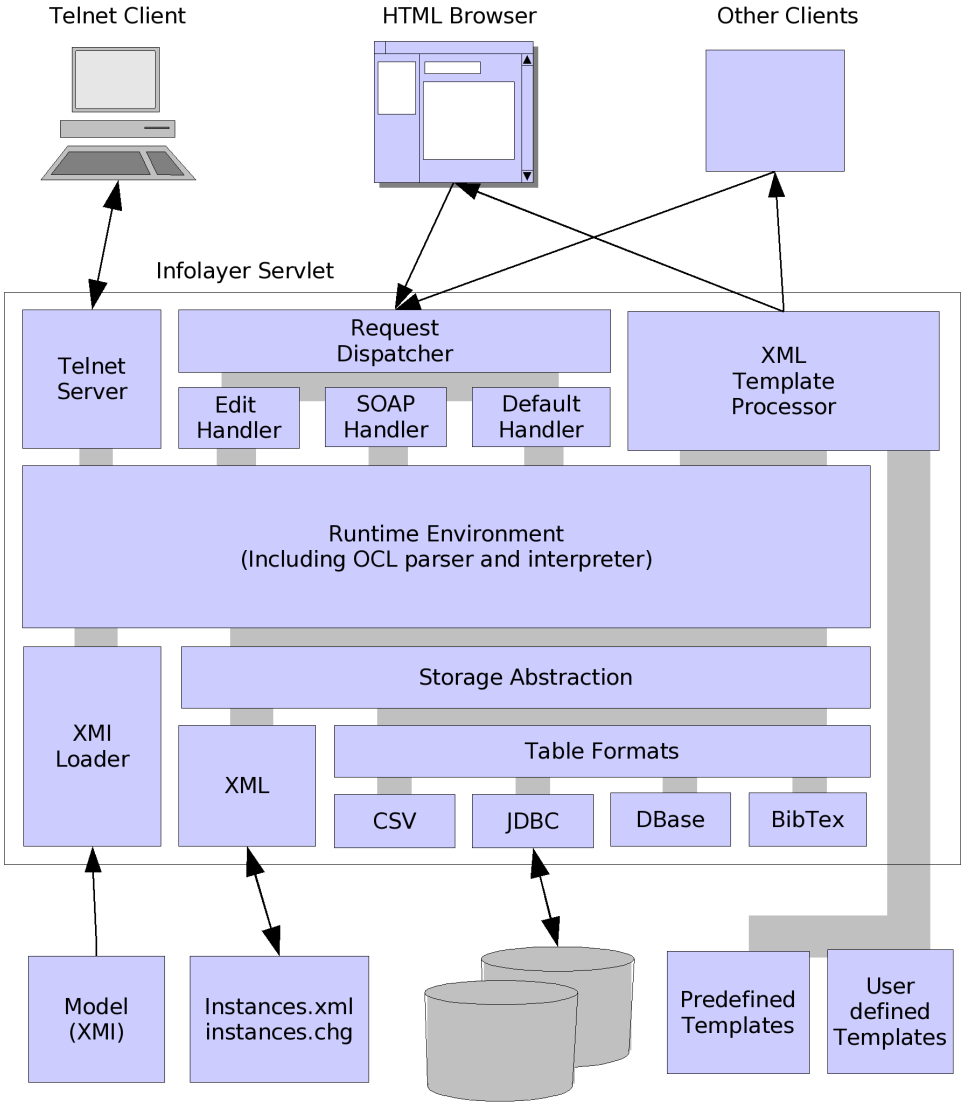


Figure 3.3: Architecture of the Infolayer System

- The HTTP request handler delegates incoming HTTP requests to a specialized processor where they are translated to requests to the system core and possibly a template name, depending on their type:
  - SOAP requests are processed immediately in cooperation with the core.
  - For a request originating from a browser, the processor performs all actions associated with the request, such as data manipulation, but the response is generated by a separate module, the template processor.
- The template processor merges predefined XML templates determining the layout of a page with the processing result of a HTTP request. It replaces instructions embedded in the template with data denoted by OCL expressions. This way, the layout of HTML and WML pages can be maintained seperately from the content. The template mechanism is sufficiently powerful to generate any desired output.

## 3.7 Sample Application

In chapter 1, we have introduced a simplified university department scenario to motivate this work. In the remaining chapters, we will use the simplified model with a few mock-up instances to illustrate different aspects of the system.

If the department sample is started locally, it can be explored by pointing a web browser to the URL *http://localhost:8080/department/*. Figure 3.4 shows two screen shots of the application, the upper one is generated using the generic Infolayer HTML interface, the lower one is created using the template mechanism. Figure 3.5 illustrates the OCL command line interface of the system.

A methodology or "best practice" for working with the Infolayer has evolved over time. It emcompasses these steps:

1. A domain model consisting of OCL-annotated UML class diagrams and possibly UML state machines is designed using one or more iterations of designing and testing a prototype, as described above.

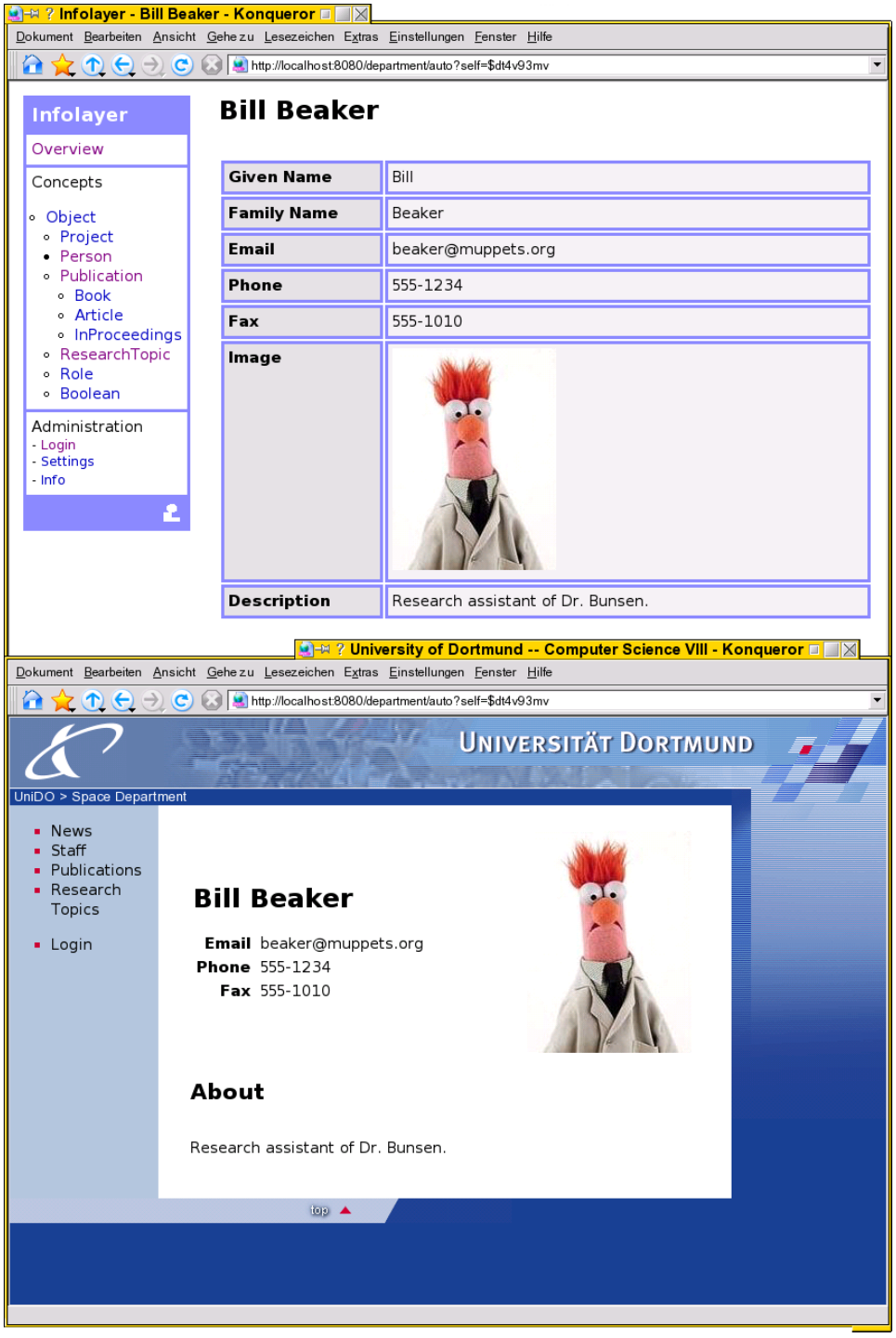


Figure 3.4: Generic and Customized Department Sample Screen Shot

```
> Person.allInstances()
type: Set(Person)
result: Set{Dr. Bunsen Honeydew, Bill Beaker}

> select(title = 'Dr.')
type: Set(Person)
result: Set{Dr. Bunsen Honeydew}
```

Figure 3.5: OCL Command Line Interface Session

2. The system's layout is tailored to personal taste or a given corporate design using one very simple template that provides a basic frame and a main navigation structure. This is usually the point at which the system can actually be utilized by its users, that is, the Web application's database can be filled with content.

3. The page layout for individual classes is successively improved. If required, the system's whole navigational structure is changed according to the specific needs dictated by the application.

4. The model itself can be modified, too, as long as these changes only introduce elements (classes, attributes, associations, constraints) into the system that are consistent with existing instances. We hope to loosen this restriction using refactoring facilities in the near future.

## 3.8 Summary

In this chapter, we have outlined how the goals raised in the introduction are met by the Infolayer system. We have shown how a UML class diagram can be directly interpreted as a web application, implicitly solving the editor problem. We have sketched a language for generating arbitrary XML content that builds upon both, the UML constraint language OCL, as well as XSLT. Here, building upon OCL ensures consistency with the remainder of the system. For both, UML and XSLT, we can leverage existing developer knowledge in many cases. By considering issues such as security and completeness, we go beyond the capabilities of the XML toolchain without compromising the overall integration and consistency of the system. The

ongoing formalization of UML provides a precise reference for comparing intended and actual behaviour of the system.

The following chapters discuss the solutions for the various problems in more detail. The next chapter illustrates which elements of UML class diagrams have been selected to show the feasibility of the system. A separate chapter is devoted to the OCL and its implementation because of its central role in various parts of the Infolayer system.

# Chapter 4

# UML Class Diagram Support

In the previous chapter, we have outlined the general architecture of a runtime environment that makes it possible to execute UML class diagrams as web applications. In this chapter, we discuss the interpretation of the elements of UML class diagrams in the system in more detail.

Although it seems technically feasible to support all the concepts of UML class diagrams, for the limited scope of a university project, an adequate subset must be selected. The first section of this chapter describes the selected subset and its interpretation in the web application context.

In the second section, we present some additional predefined classifiers which are not part of the UML standard. They are used in the Infolayer system to address common requirements of web applications such as dealing with binary files and user management.

In the third section, we introduce a set of Infolayer specific model annotations that can be used to limit access to elements of the model to specific users or general rules in the form of OCL expressions.

Some capabilities of the Infolayer that are not directly covered by the UML specification can be configured using a standard UML extension mechanism: all model elements can be annotated with special name-value pairs, so-called *tagged values*. The Infolayer makes use of tagged values for various purposes, such as customization of the presentation, access permissions and other features like database connections. All tags used in the infolayer system start with the prefix “il-” .

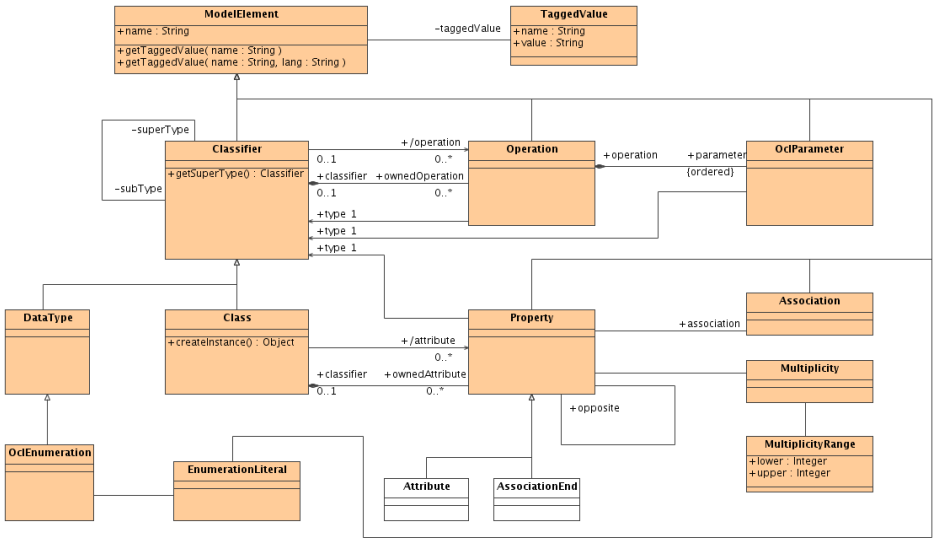


Figure 4.1: Infolayer Meta-Model

## 4.1 Supported Elements of UML Class Diagrams

The following elements of UML class diagrams are understood by the Infolayer system:

- Classes
- Properties (Attributes and Associations including Compositions)
- Operations
- Generalization / specialization relations between classifiers
- Data Types
- OCL expressions

Those elements form the subset of the UML 2.0 proposal for class diagrams depicted in figure 4.1. The selection matches the UML core defined in [102]. Properties and operations support visibility to the extent that public elements are shown in the user interface by default, whereas private properties are not.

For specifying the UML model, any UML tool that generates valid XMI 1.1 can be used.

Relevant features and model elements that are (currently) not supported are:

- Multiple Inheritance
- Interfaces
- Association Classes
- N-Ary Association
- Classes with overlapping sets of instances

Those elements were not required in the applications we used to evaluate whether the system meets its requirements and thus left out. However, for other applications it may make sense to add them. Even if unimplemented concepts can be emulated with implemented elements, it was not the goal to identify a general minimum working set.

### 4.1.1 Classes

Classes are—like in object oriented databases—used to describe the entities managed by the system. Classes may be derived from other classes, inheriting their properties, and building a class hierarchy. Classes that are marked abstract cannot be instantiated. In the Infolayer system, each instance of a class has a stable unique alphanumerical ID, identifying the instance.

IIn addition to the standard UML capabilities, the Infolayer supports tagged values for user interface customization, localization, and user access management. The annotations for customization and localization are described in appendix C; the annotations for access management are described in section 4.3.

### 4.1.2 Primitive Data Types

Data types are simple types where the instances do not have an "identity"—it is not possible to create two different instances of the type *Integer* with

| Type | Domain |
|---|---|
| Boolean | Supported values are *true* and *false* |
| Integer | 64 bit signed integer numbers |
| Real | 64 bit real numbers |
| String | Unicode text strings |

Table 4.1: Predefined Data Types that can be used in Attributes

the value 5. Data types supported by the Infolayer system are the OCL types *Boolean*, *Integer*, *Real*, and *String*. In contrast to UML, the domain of the numeric types is limited to simplify the implementation of the system. Table *fig:datatypes* shows an overview of the built in UML data types and their (restricted) domains in the Infolayer system, roughly corresponding to primitive data types in programming languages.

### 4.1.3 Attributes

Attributes are typed properties of classes, where the type is usually a simple data type. Attributes with a class type are regarded as compositions, as specified in the UML standard. A default value of an attribute is specified by providing an OCL expression; that expression is interpreted whenever a new instance is created. Attributes must have an instance scope, attributes with a classifier scope are not supported.

### 4.1.4 Associations

Associations in UML can be seen as a tuple of attributes representing the association ends at each connected class, where the "attribute values" are not independent, but synchronized with the other ends. In contrast to regular attributes, the type of the association ends cannot be a primitive type,. Most information about associations relevant for the information layer, such as the local name and the cardinality, is accessible via the association ends. Like for classes and attributes, additional information regarding access permissions and appearance can be set using tagged values. If the UML CASE tool does not allow to set tagged values for the association ends, it is possible to set tagged values named *il-endname-tagname* for the association itself. Those values are assigned to *il-tagname* at the association end *endname*.

For compositions, the associated elements are deleted automatically if the owning element is deleted.

If the field "ordering" is set to the value *ordered*, an ordering that cannot be determined from the instances implicitly is preserved. For the Infolayer, that means that the ordering in which the links were set is not changed. For all other values, the associated instances are ordered alphabetically in the user interface.

Like for attributes, default values are supported, and the scope of the association must be instances.

Please note that attributes and associations are only displayed in the user interface by default if they are marked public. Many UML tools set the default access rights for attributes and association ends to private.

### 4.1.5 Operations

It is possible to add operations to the classes. For example, an operation *getName()* may return the concatenation of *givenName* and *familyName* fields. The implementation of the method must be provided as OCL expression in the tagged value "il-implementation". For the given example, the OCL expression would be

```
givenName.concat(' ').concat(familyName)
```

Each class contains a predefined method *toString()*, which is called when a string representation of an object is needed, for instance in titles or overviews. The default implementation displays the *name* or *title* attribute of the object if available. Otherwise, the unique ID of the object is returned. For persons, an appropriate implementation would correspond to the *getName()* implementation shown above.

For operations without parameters, action buttons are displayed in the user interface, which allow the user to trigger a method invocation if he or she has the required permissions. The permissions and other customization options for operations are discussed in detail in the following chapters.

In addition to user defined operations, the Infolayer provides a set of predefined OCL operations. The Infolayer OCL support is discussed in more

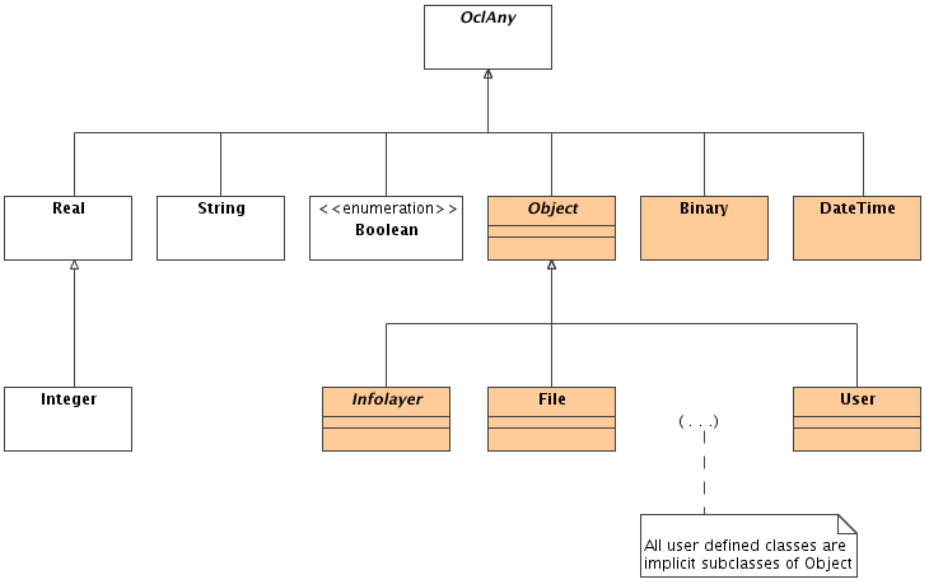


Figure 4.2: Predefined Classifiers in the Infolayer System

detail in the next chapter. Methods with side effects are handled in chapter 6.

In contrast to attributes and association ends, operations may be defined with a class scope.

## 4.2 Additional Predefined Classifiers

In addition to the predefined UML types presented in section 4.1.2, the Infolayer contains the predefined data types *Binary* and *DateTime* and the classes *Object*, *Infolayer*, *User* and *File*, providing some basic functionality for web applications. Figure 4.2 shows an overview of all predefined types available at the user model level, including the abstract OCL base type *OclAny*, which will be handled in the next chapter. Infolayer-specific types are marked with a dark background in the diagram. The data type DateTime is needed to store points of times such as publication dates or dates of changes. Support for binary data is needed to be able to store binary data such as PDF documents or images in the system.

The class Object was added to be able to track changes in the system in a central place and to simplify object identity by adding an explicit unique ID value, that is also used to identify links in the persistent storage mechanism. The class File type combines binary data with a file name that serves as MIME type indicator, allowing the browser to automatically select the right form of presentation. The class Infolayer provides access to information about the system state such as the current time and user, represented by the class User.

The generation of custom classes with a name identical to the predefined ones will lead to conflicts or ambiguities in the system and is not recommended. However, it is possible to add an explicit representation of the predefined classes to the UML model, in order to set tagged values or additional properties for this class. For instance, by adding properties to *Object*, it is possible to add attributes and operations to all other classes contained in the model.

### 4.2.1 Predefined Data Types

The Infolayer-specific types *DateTime* and *Binary* can be used to store date and time values and arbitrary binary data.

### 4.2.2 Object

The class Object is the implicit abstract base class of all other classes.

The Infolayer supports a set of methods for objects in addition to the built in OCL methods inherited from *OclAny*, presented in the following chapter. The method *getId()* returns the unique identification string of the object. The callback methods *onChange()*, *onCreate()*, and *onDelete()* allow to react on changes of the object if they are overloaded with an own implementation. The default implementation of those methods is empty.

An important method of the class *Object* with a class scope is *createInstance()*, which can be used to create new instances. Since this method has an obvious side effect, it cannot be used in OCL constraints and query operations.

| Property | Type | Mandatory | Description |
|---|---|---|---|
| password | String | yes | Stores the encrypted password of the user |
| login | String | no | Stores the login name of the user. If not present, the *toString()* method is used to determine the login string. |
| admin | Boolean | no | Determines whether a user has administrative rights. By default, administrators can manage users and have access to the Infolayer command line interface. If not present, the user with the name "admin" is the only user with administrative rights. |

Table 4.2: User Class Properties Interpreted by the Infolayer System

### 4.2.3 Infolayer

The Infolayer class contains the static methods *getCurrentUser()* and *getCurrentDateTime()* to access general information about the current state of the Infolayer:

**getCurrentUser(): User** : Returns the current user, corresponding to the current browser session.

**getCurrentDateTime(): DateTime:** Returns the current system date and time.

The Infolayer class supports the tagged value *il-userclass* to instruct the Infolayer system to use a different class for users than the default *User* class. The built in user class and the requirements for custom user classes are described in the next section.

### 4.2.4 User

The User class is generated automatically by the system if no explicit user class is set using the *il-userclass* tagged value of the *Infolayer* class. A custom user class must contain at least an attribute named *password* of the

type String, which is used internally to store the passwords of the users in an encrypted form. The additional attribute *login* is optional. If present, it is used to store the login name of the user. If not present, the *toString()* method is used to determine the login string.

The Infolayer makes a distinction between regular users and administrators. By default, regular users can edit all the content of the system, except from *User* instances belonging to other users; only administrators can create new users or change existing users. Furthermore, the OCL command line interface of the Infolayer system is restricted to users with administrative rights.

In addition to the predefined properties of the user class, it is possible to add attributes, association ends and operations that can be used for customized access control such as user group membership or similar.

### 4.2.5 File

The File class is designed for storing binary content along with a file name. It consists of the following attributes:

**name: String** The name of this file object, including the file extension.

**data: Binary** The binary content of the stored file.

**counter: Integer** A download counter for the file.

The tagged value "il-mimetype" may be used in attributes of type File in order to restrict uploaded files to the given mime type(s).

## 4.3 Dynamic Access Management

In the Infolayer system, OCL expressions can be used to limit or extend access to classes, attributes, associations, and operations. For this purpose, OCL expressions can be attached to the corresponding elements of the UML model using specialized tagged values, such as *il-permission-read* for read access and *il-permission-write* for write access. Table 4.3 shows an overview of all tagged values that can be used for access control.

| Tag | Attached to | Description |
|---|---|---|
| il-permission-read | class | If the expression evaluates to *true* for the instance denoted by *self*, the current user may view this instance. The default value is *true* |
| | property | Determines whether the attached attribute or association end is visible in the user interface. If not present, the expression attached to the class is used to determine the read access rights. |
| | query operation | Determines whether the attached query operation may be executed. If not present, the permission specified for the class applies. Please note that this property has no influence on non-query operations |
| il-permission-write | class | If the expression evaluates to *true* for the instance denoted by *self*, the current user may modify or delete this instance. The default value is: *not Infolayer.getCurrentUser().oclIsUndefined* |
| | property | Determines whether the attached attribute or association can be edited in the user interface. If not present, the expression attached to the class is used to determine the read access rights. |
| | non-query operation | Determines whether the attached non-query method may be executed. |
| il-permission-query | class | Determines whether queries may be performed for the attached class. In contrast to most other permission expressions, self points to the class and not an instance. |
| | property | Determines whether queries may be performed for the attached attribute or association end. In contrast to most other permission expressions, self points to the class and not an instance. |
| il-permission-create | class | If the expression evaluates to true, the current user may create new instances of the attached class. In contrast to the other permission expressions, *self* points to the class. The default value is: *not Infolayer.getCurrentUser().oclIsUndefined()* |

Table 4.3: Tagged Values Controlling Access Permissions

The OCL expressions contained in those tags are evaluated at runtime, when the attached element is accessed. If the evaluation result is true, an access permission is granted, otherwise not. While any boolean OCL expression can be used to define the access rights, in most cases it will make sense to let the permissions depend on the current user. In Infolayer OCL expressions, the current user can be determined using the method *Infolayer.currentUser()*.

For instance, the following expression can be used to limit access for a class or property to scientists in the example model:

```
let user = Infolayer.getCurrentUser() in
(not user.oclIsUndefined()) and (user.job = Job::Scientist)
```

Of course, the access permissions can also be based on other values such as the current time, but in most cases they will also depend on the current user.

Some of the tagged values can be attached to both, classes and properties. If both options are given, only the expression at property level is considered and the expression at class level is ignored. Permissions are inherited along the class hierarchy.

## 4.4 Summary

In this chapter, we have described a subset of UML class diagrams that can be used to build models that can be directly interpreted as web applications. We have clarified the interpretation of those elements in this context where necessary. Our subset is similar to the subset defined in [102]. However, in contrast to Richters' work, our subset was selected to show the feasibility of the concept, and not to identify a small core subset for a specific purpose. Thus, it may make sense to extend this set to reach more convenience and a higher level of abstraction. As an example, association classes could be supported to make it possible to assign properties to associations.

In addition to describing the UML coverage, we have also introduced a small set of pre-defined classifiers that model common aspects of "real world" web applications such as users, access permissions, and binary files.

In the next chapter, we will discuss the OCL implementation of the Infolayer system, which is the foundation for several aspects of the system, such as the implementation of query operations and the XML templates.

# Chapter 5

# OCL Support

OCL is a side effect free textual expression language that was added to UML in order to be able to express formal constraints of the model that cannot be expressed with other, mostly graphical, elements of the UML. The original purpose of the Object Constraint Language (OCL) [93, chapter 6] is to specify invariants on classes and types in a class diagram. Without OCL, those constraints would need to be written down in plain text, quickly leading to ambiguities.

However, OCL is also quite suitable as a general purpose query language [30]. Although there are specialized query languages for object oriented systems such as the Object Query Language (OQL) [17], it does not seem to make much sense to add an additional language to an UML based system if the OCL is sufficiently qualified for the desired tasks.

Thus, the OCL is utilized everywhere in the Infolayer system where the ability to express sophisticated queries is necessary or useful. In particular, OCL is used as an expression language in XML templates and for defining query operations in the class diagram. The XML templates are a central element of the Infolayer system that is not directly covered by elements of the UML. They are mainly used for building web-based user interfaces. The XML templates are discussed in detail in chapter 7. In chapter 8, we will show how delayed evaluation and referential transparency of OCL expressions can be used to optimize generated SQL queries.

An detailed overview of the Infolayer OCL support, focussing on deviations

from the official standard, is contained in appendix B. As for class diagrams, we needed to leave out some parts that were not essential for our purposes.

In this chapter, we will describe OCL-related aspects of the Infolayer system that are not covered by the UML specification. Section 5.1 shows how query operations can be (technically) implemented in OCL in the Infolayer system using tagged values. Section 5.2 describes how the meta model can be accessed from OCL in the Infolayer system, following the power type approach proposed by Stefan Flake [39]. Section 5.3 shows the Turing completeness of OCL.

## 5.1 Implementing Query Operations and Derived Properties in OCL

One of the main purposes of the OCL in the Infolayer system is to provide operational specifications of query operations. Query operations are operations without side effects, denoted by the *isQuery* attribute in the class diagram.

In order to specify the implementation of a query operation in the Infolayer system, the operation must be annotated with the tagged value *il-implementation*, holding the OCL code for the implementation of the method. Similarly, calculated properties can be implemented using the tagged value *il-getter*.

Like for other tagged values that are interpreted by the Infolayer, the specifications may alternatively be embedded in the documentation of the operation, marked by *@il-implementation* or *@il-getter*.

The implementation of operations that are not necessarily free of side effects and writeable properties is discussed in the next chapter.

## 5.2 Access to the Model (M1-Level)

The Infolayer system does not only provide access to an object diagram conforming to a model, but also access to the user model itself. Access to the model is required to be able to implement functionality of the XML template

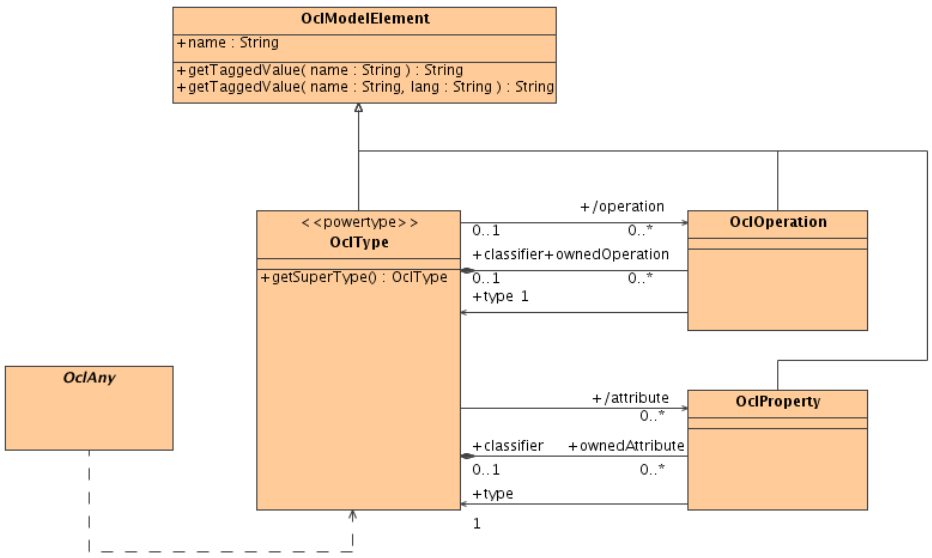


Figure 5.1: OCL Access to the User Model (M1) in the Infolayer System

mechanism (chapter 7) in OCL, for instance displaying all properties of a class. Unfortunately, the OCL 2.0 introspection capabilities are very limited and not sufficient for this purpose.

In OCL 2, the type of any object can be determined at runtime using the *oclType()* method, which returns an instance of the type *OclType*. In contrast to earlier Versions of the OCL, *OclType* is now an enumeration. It provides methods to query information about the given type, such as *name()* to query the name or *supertypes()*, returning the set of all super types. However, the information available is limited to the name of the classifier, the set of all instances, and sub and super types. Earlier versions of the OCL specification allowed to obtain the operation and property names, but those operations are not part of OCL 2.0.

To overcome this limitation and to provide better access to the user model (M1), the Infolayer provides access to the classes of the user model as instances, following a UML powertype approach proposed by Stefan Flake [39]. A powertype is a type whose instances are classes of the user model. In the Infolayer system, the type *OclType* is implemented as power type for *OclAny*. Thus, all subclasses of *OclAny* are accessible as instances of *OclType*.

The operation *oclType()*, which is defined in *OclAny* and thus available for all types, returns the type of the instance as an instance of the type *OclType*. *OclType* corresponds to the M2 element *Classifier* and provides methods and properties making it possible to access to the user model (M1), as depicted in figure 5.1. The diagram is a simplified version of the Infolayer metamodel presented in the previous chapter, however, the names of the corresponding M2 elements are prefixed with "Ocl" for consistency with other parts of the OCL specification. A full list of methods and properties is provided in the appendix.

In particular, access to the user model makes it possible to specify generic rules for XML generation using the XML template mechanism presented in chapter 7.

Please note that currently the Infolayer system provides only read access to the user model.

## 5.3 Turing-Completeness

Mandel and Cengarle have proven that OCL is not Turing complete [75]. However, Richters claims that the expressiveness of OCL may be extended to recursive functions by allowing the implementation of query operations in OCL [102]. As described in the section 5.1, this option is available in the Infolayer system using the tagged value *il-implementation*. In the remainder of this section, we prove Richters' claim.

One option to demonstrate Turing completeness is to show that it is possible to implement $\mu$-recursive functions [107]. However, this technique requires unlimited integers, and in the Infolayer system integers are limited to 64 bit. Moreover, it would be nice to use the completeness of OCL to show that the XML template mechanism presented in chapter 7 is able to generate any computable XML output. This seems simpler with the tape output of a Turing Machine. Hence, we will implement a Turing Machine in OCL in order to show the Turing completeness of our extended OCL.

A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \#, F)$, where

- $Q$ is s a finite set of states,

- $\Sigma$ is a finite set of symbols, the input alphabet,
- $\Gamma = \Sigma \setminus \#$ is a finite set of symbols, the tape alphabet,
- $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R, N\}$ is the transition function,
- $\#$ is a symbol called blank,
- $q_0 \in Q$ is the initial state,
- $F \subset Q$ is a set of final states.

A Turing Machine has an endless tape with unlimited storage positions for the input alphabet. Any portion of the tape that has not been visited by the read/write head is initially filled with blank (# ) symbols. In each step, the Turing Machine reads the symbol at the current position of the read/write head. Then the transition function $\delta$ is used to determine the new state of the machine, the symbol to be written at the current position and the movement of the read/write head, depending on the current symbol and state. A Turing Machine terminates when a final state is reached.

Since the sets of states and symbols are finite, they can be mapped to integers in order to simplify the OCL implementation:

- The states can be encoded as integers, where negative integers mean halting states.
- The tape and input alphabet can be mapped to positive integers.
- Zero can be used as blank symbol.

The transition function $\delta$ can be modeled as a set $\Delta$ of 5-tuples representing single transitions $(q, \sigma, q', \gamma, m)$ where

- $q$ is the current state,
- $\sigma$ the current symbol,
- $q'$ is the new state,
- $\gamma$ the symbol to be written, and

```
def: let
  step (delta: Set(Sequence(Integer)),
        state: Integer,
        tape:  Sequence(Integer),
        pos:   Integer
       ): Sequence(Sequence(Integer))

  = if pos < 1 then
      step(delta, state, tape->prepend(0), pos+1)
    else if pos > tape->size() then
      step(delta, state, tape->append(0), pos)
    else if state < 0 then
      Sequence{tape, Sequence{state, pos}}
    else
      apply (delta, tape, pos,
        delta->any(transition|transition->at(1) = state
        and transition->at(2) = tape->at(pos)))
    endif endif endif


def: let
  apply (delta: Set(Sequence(Integer)),
         tape: Sequence(Integer),
         pos: Integer,
         transition: Sequence(Integer)
        ): Sequence(Sequence(Integer)) =

  = step (delta,
      transition->at(3),
      tape->subSequence(1, pos - 1)
        ->append(transition->at(4))
        ->union(tape->subSequence(pos + 1, tape->size())),
      pos + transition->at(5))
```

Figure 5.2: A Simple OCL Implementation of a Turing Machine. The function *step*() analyzes the current state and tape position, the function *apply*() applies the corresponding transition and moves to the next step.

- $m \in \{-1, 0, 1\}$ denotes the head movement $\{L, N, R\}$.

The OCL functions *step*(*delta, state, tape, pos*) and *apply*(*delta, tape, pos, transition*) shown in figure 5.2 realize a corresponding Turing Machine. The first parameter *delta* represents the transition function $\delta$, encoded as described above, using the OCL type *Set(Sequence(Integer))*. The remaining parameters represent the current configuration of the Turing Machine. The second parameter *state* is the current state $q$ of the machine. The third parameter is the tape, encoded as an integer sequence. The fourth parameter *pos* is the current position *pos* of the read/write head.

The machine is started by calling *step*() with the parameters $\Delta, q_0, t_0, p_0$, where $t_0$ denotes the initial tape configuration encoded as *Sequence(Integer)* and $p_0$ denotes the initial head position.

In order to prove that the step function actually implements a Turing machine, we need to show that

1. the tape is extended as needed and thus potentially infinite,
2. the machine terminates if (and only if) a halting state is reached,
3. the machine reads the symbol at the current position of the read/write head and determines the correct transition according to the delta function, current tape symbol and state, and that
4. the new state and head position are set according to the transition, and the machine continues with 1.

**Claim 1** *The tape is extended as needed, so although the integer sequence is finite, the machine will never reach its end.*

**Proof 1** *The read/write head position* pos *is outside the scope of the current tape representation when it is smaller than one (the first index position of an OCL integer sequence), or if it is larger than* tape->size*(). The OCL expression*

```
if pos < 1 then
  step(delta, state, tape->prepend(0), pos+1)
```

*makes sure that the* step*() function is reapplied with an extended tape where a blank (0) is inserted at the beginning and the position adjusted accordingly if the read/write head position is smaller than one.*

*The OCL expression*

```
else if pos > tape->size() then
  step(delta, state, tape->append(0), pos)
```

*makes sure that the* step*() function is reapplied with an extended tape where the blank (0) is appended at the end of the tape.*

**Claim 2** *The machine terminates if and only if a halting state is reached.*

**Proof 2** *A halting state is represented by a negative value of* state*. The OCL expression*

```
else if state < 0 then
  Sequence{tape, Sequence{state, pos}}
```

*returns the configuration of the turing machine if* state *contains a negative value. Since all results of* step*() and* apply*() are immediately returned without further processing, this ensures that the OCL implementation terminates and yields the current configuration of the machine.*

**Claim 3** *The machine reads the symbol at the current position of the read/write head and determines the correct transition according to the delta function, current tape symbol and state.*

**Proof 3** *The symbol at the current tape position pos can be read with the OCL expression* $tape- > at(pos)$*. To determine the correct transition, the sequence in delta must be found where the first element matches the current state state and the second element matches the current tape symbol. This is performed by the OCL expression*

```
delta->any(transition|transition->at(1) = state
        and transition->at(2) = tape->at(pos), delta, tape, pos)
```

**Claim 4** *The new state and head position are set according to the transition.*

**Proof 4** *In order to actually perform a transition, we define a help function* apply *that applies a transition, represented as a 5-element integer sequence. The apply function takes as arguments the representation of $\delta$, the tape, and the transition to be performed, where the transition is determined as described in the previous step. The corresponding call to* apply*() is:*

```
apply (delta, tape, pos,
  delta->any(transition|transition->at(1) = state
  and transition->at(2) = tape->at(pos), delta, tape, pos))
```

*The result of the* apply*() function is immediately returned as the result of the* step *function without further processing.*

*The* apply*() function uses its arguments to recursively call* step() with the configuration of the machine that results from applying the transition to the current configuration, keeping the machine running until a halting state is detected in *step*().

The first two elements of the transition are the old state and tape symbol that were used in the previous step to look up the right transition.

Element three of the transition ($transition- > at(3)$) determines the *state parameter of the* step*() function, denoting the new state of the machine.*

*The fourth element of* transition *is the new tape symbol that needs to be written at the current head position. The OCL expression*

```
tape->subSequence(1, pos-1)
      ->append(transition->at(4))
      ->union(tape_->subSequence(pos_ + 1, tape_->size()))
```

*generates the new tape representation by appending the new tape symbol and remainder of the tape to the part of the tape preceding the current head position. The new tape representation is delivered to the* tape *parameter of the* step*() function.*

*The fifth element of the transition represents the head movement. Since a movement to the left is represented as -1, a movement ot the right as 1,*

*and no movement as 0, the new head position can determined by adding the integer representation of the head movement to the head position:* $pos + transition- > at(5)$.

*The result of the* step*() function is immediately returned as the result of the* apply*() function.*

The fact that the results of the recursive calls in the machine implementation are returned immediately without further processing could be used in the OCL interpreter to return the results directly to the original caller, avoiding unnecessary usage of stack space (*tail recursion*). Without this optimization, the number of steps the turing machine can perform is limited by the stack space available. With the optimization, the only limit would be the memory that is needed for the tape representation.

## 5.4 Summary

In this chapter, we have provided an overview of the meta-model access capabilities of the Infolayer OCL implementation, following the powertype approach, as suggested by Stefan Flake [39]. We have proven the assertion of Mark Richters that OCL is Turing complete by providing a Turing machine implemented in pure OCL.

The decision to use OCL as a query language inside the system proved to be very helpful. A previous version of the Infolayer used OQL instead, so we are in a position to draw a comparison here: OQL is basically a slight syntactic adaption of SQL to the object-oriented world. Queries that encompass multiple associations with a cardinality greater than one tend to become lengthy and unreadable, because they result in nested `select` statements. The implicit `collect()` in OCL expressions supports much shorter and more intuitive queries. As a result, we propose to use OCL as a general query language for OODBMS.

In the next chapter, we go beyond queries and show how UML action semantics and OCL can be combined to a consistent language that permits side effects without tainting OCL or leaving the UML framework.

# Chapter 6

# Actions

In several cases, it is desirable to be able to specify functionality that is not free of side effects in a web application. For instance, in the sample application, one may want to implement a "vote" button, making it possible for the users to cast a vote for their favorite paper.

As discussed in the previous chapter, OCL is well suited for implementing query operations, but the language does not provide means for the implementation of any functionality that modifies the current system state. Of course one could simply implement an interface to an existing programming language—such as Java—to circumvent this problem. Actually, the Infolayer features this kind of interface (see appendix D), but this means leaving the UML framework. For implementing simple operations like incrementing a counter, falling back to a regular programming language seems to be a overly complex solution and introduces additional dependencies and integration issues. Also, general purpose languages usually do not support UML concepts such as associations as first class entities, introducing a step of indirection when accessing the system state [110].

For modeling operations with side-effects, so-called actions, UML 1.5 contains the specification of action semantics [93, chapter 2]. Action semantics is a framework for the formal description of programming languages [85]. One motivation for including action semantics in the UML is to enable language-independent code generation [81]. An implementation of the action semantics specification seems to be a natural choice for adding non-query operations to the Infolayer system without leaving the UML framework. The

only problem is that the action semantics specification does not contain a surface language, but stays on the level of abstract syntax. The specification refers to several possible action surface languages, but all of them rely on a syntax that differs significantly from the OCL syntax. Yet, since the action semantics covers a superset of the OCL, using two completely different syntaxes seems confusing and inappropriate. Thus, instead of using one of the action languages referenced in the specification, we have created syntax constructs for actions that make it possible to embed OCL expressions for read access to the current system state.

Section 6.1 of this chapter provides a brief overview of the UML action semantics. Section 6.2 discusses the relation between OCL and UML action semantics. Section 6.3 describes our action surface language based on OCL Queries (ASOQ). Finally, section 6.4 shows how ASOQ implementations can be attached to operations, properties, and predefined user interface callback methods in the Infolayer system.

## 6.1 UML 2.0 Action Semantics

In the previous chapter, we have seen that the OCL is not sufficient to describe operations with side effects in an operational way. In earlier versions of the UML standard, textual descriptions were used to capture the behavior of operations. However, the OMG has recognized the need to be able to define the behavior precisely in a standardized form. A precise specification of behavior makes it possible to share semantics of actions and operations between modelers and tools, leading to significant advantages such as a higher level of abstraction, support for formal proofs, model-based simulation, and code generation [81]. Hence, the OMG accepted a proposal to formalize different categories of actions [1] for inclusion in UML 1.5 and 2.0:

**Control Actions** are used to model loop and branch structures.

**Read and Write Actions** are used to access the values of object properties and to create new instances or to delete existing instances.

**Computation Actions** transform a set of input values to produce a set of output values without side effects on other parts of the system.

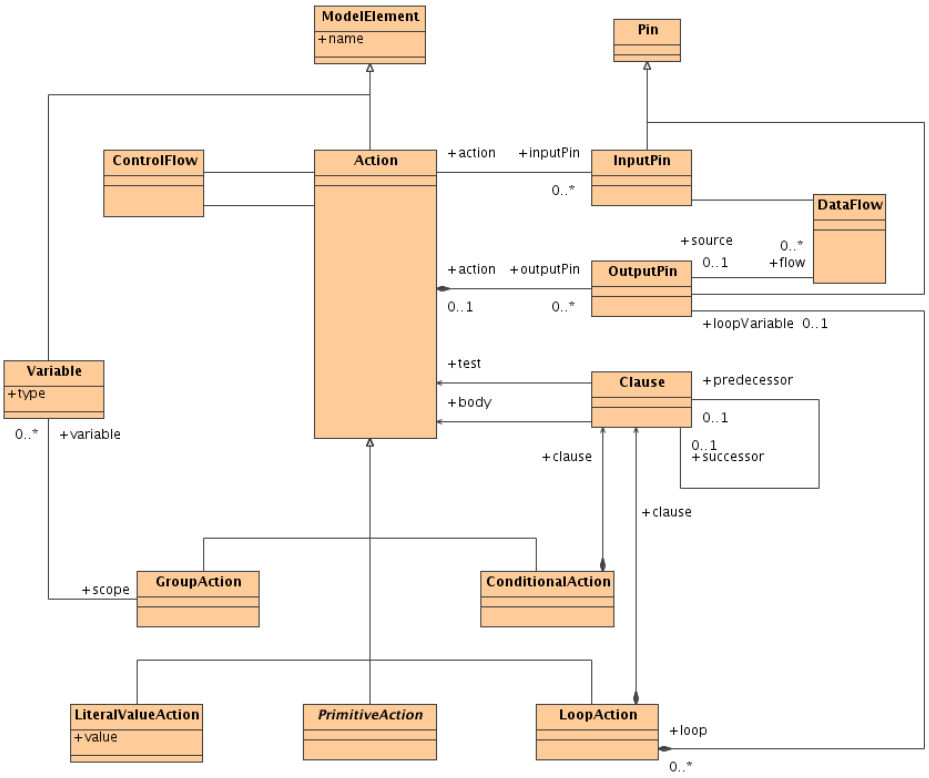


Figure 6.1: Control Action Overview

**Collection Actions** permit the (parall) application of an action to a set of data elements.

**Messaging Actions** trigger asynchronous or synchronous actions such as state machine transitions or method invocations with a return value.

**Compositional Actions** model iterations and conditionals.

**Jump Actions** allow deviations from the main path of control flow in iterations similar to the *break* statement known from many programming languages.

Following the general spirit of UML, those actions are modeled as UML classes. Figure 6.1 shows an overview of the classes describing control structures in the Action Semantic framework. The other categories are described in a similar way. Each action has a set of input and output *pins*. Output

pins can be connected to one or more input pins of other actions, creating sequential dependencies. Alternatively to the pin mechanism, the class *ControlFlow* can be used to explicitly enforce an order of execution. The class diagrams of the action semantics are roughly similar to the grammar specifying the abstract syntax of a programming language. Naturally, object diagrams could be used to capture the equivalent of the abstract syntax tree of a program, altough even for simple expressions the diagrams become extremely verbose. What is missing in this system is a concrete syntax, which is called surface language in the action semantics specification.

## 6.2 Action Semantics and the Object Constraint Language

The UML action semantics recognizes the need for an action semantics surface language, but does not recommend a specific one. The specification contains a set of informal mappings to action languages used in different UML tools, namely the Action Specification Language (ASL) [66], the BridgePoint Action Language (AL) [101], the Kabira action semantics (Kabira AS) [64], and the action language subset of the Specification and Description Language (SDL), an international standard widely used in the telecommunications industry [117].

All those languages rely on a syntax that is incoherent with the existing UML expression language, the Object Constraint Language (OCL). Actually, large parts of the action semantics specification duplicates functionality that is already covered by the OCL, such as

- Navigation and read access to properties
- Computation
- Calls to query operations
- Collection operations

The great overlap of the model access constructs defined in the AS and OCL specifications suggests that using two completely different syntaxes

may be inappropriate and confusing; one would expect a surface language that leverages existing OCL knowledge and infrastructure. Naturally, due to the side effect free nature of OCL, OCL cannot cover actions such as write actions or calls to non-query operations, but using OCL for the parts covered would mean a significant improvement over the current situation.

It seems quite straight-forward to build an action surface language that is based on OCL expressions, without tainting OCL itself with side effects. Anneke Kleppe and Jos Warner suggest an action clause as an extension to OCL that would fit nicely with the declarative nature of OCL [68], but for the desired purpose a fully operational solution is required. Since the OCL is a subset of the AS, there are two options for building an action surface language based on OCL:

1. Map all OCL constructs to actions, then add new syntax constructs for actions that are required, but not covered.
2. Embed OCL expressions in new syntax constructs for actions.

The first option requires a complete mapping of the abstract OCL syntax to actions. This would mean to give up declarative semantics in OCL, or to have two flavors of OCL with different specifications that would need to be aligned carefully.

The second option can be implemented by referring to the existing OCL surface language, without modifying it, maintaining a clean syntactical separation between plain queries and actions that may influence the system state. This approach keeps the interface between the languages minimal and makes it possible to keep both languages relatively separate, not tainting OCL by introducing side effects to OCL itself. Since this approach seemed the more promising one, we have implemented it in our Infolayer system.

## 6.3 ASOQ

The Infolayer system is basically an UML runtime environment that interprets class diagrams and state charts as a Web application. It can be seen

as an implementation of a variant of MDA that does not compile PIMs, but interprets them instead. In this approach, the transformation from the PIM to the PSM is handled implicitly by a model-driven runtime (MDR) environment. Where MDA potentially transforms object-oriented concepts to non object-oriented ones (as in the case of the relational database), our MDR implements selected parts of the UML metamodel and interprets them for a given application domain.

In the system, OCL is used to implement user defined operations without side effects. To implement operations that are not free of side effects, we have implemented a language termed *Action Semantics Surface Language based on OCL Queries* (ASOQ), following the ideas outlined in the previous section [54].

Syntax constructs needed to be created only for the functionality that is not already present in OCL, namely composite actions, write actions, and messaging actions. We tried to align the new constructs with existing OCL syntax:

- The OCL *if–then–else* structure includes *endif* as a specific end marker, where other languages such as C and PASCAL use an explicit block structure, marked by curly brackets or special keywords such as *begin* and *end*. For consistency, implicit blocks and end markers specific to the control structure are used in ASOQ also for loops (while–do–enddo).

- '=' is used mainly as comparison operator in OCL; when used in assignments it may be paired with a colon and a type declaration. Thus, using ':=' for property and variable assignments seems consistent with OCL.

- OCL uses dots ('.') to separate parts of path expressions. In ASOQ, we will use the exclamation mark ('!') to indicate an operation with side effects at the end of a path expression.

- The OCL *let...in* declaration block and the *if–then–else* structure can be reused in the ASOQ with an identical syntax to build group actions and conditional actions.

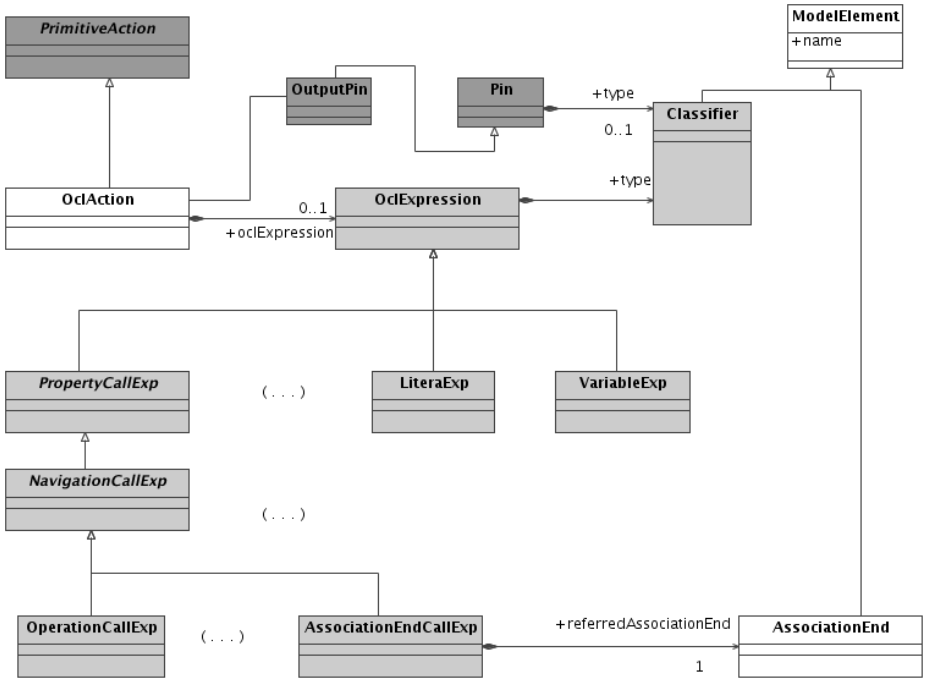


Figure 6.2: OclAction and Related Classes. Classes Contained in the Action Semantics Specification are Dark Gray, Classes from the OCL Specification are light Gray. The new Class *OclAction* and UML Core Classifiers are White

Before we can define the details of the syntax, we need a construct to integrate arbitrary OCL expressions in the action semantics framework. For this purpose, we create a class *OclAction* that inherits from *PrimitiveAction*, but has a link to an *OclExpression* object, defined in the OCL specification. The new class makes it possible to embed OCL expressions in chains of actions. It can have at most one *result* output pin, which delivers the result of the evaluation of the expression. The upper part of figure 6.2 shows the *OclAction* class. The *isReadOnly* property of the *OclAction* is always true, since OCL expressions cannot have any side effects. Variable references can be handed from actions to the OCL parser by pre-initializing the OCL environment accordingly. Full read access to the System state is already available in OCL.

The OCL specification includes a mapping from the concrete syntax to the abstract syntax in the form of a full attribute grammar. The full attribute grammar contains not only the syntax definitions, but also a specification of the resulting abstract syntax tree. For consistency with the OCL specification, we use the same formalism to specify the ASOQ syntax. This enables us to include some OCL constructs by reference, too.

The following subsections contain the syntax specification of the ASOQ building blocks that complement the OCL to a action semantics surface language:

- A simple block structure including variables and statements
- The OclAction allowing to embed OCL constructs
- An if-then-else conditional structure
- A while loop structure
- Non-query method invocations
- Variable assignments
- Property assignments

In the next section, we will show in detail how the full attribute grammar yields a completely operational action graph in addition to defining the syntax. For this purpose, we start with property assignments, before proceeding

to introducing the top level ASOQ block stucture. In the following sections, we will focus only on special aspects of the corresponding constructs.

### 6.3.1 Property Assignments

Property assignments are not included in the OCL because they change the current state of the system. In ASOQ, we introduce the binary infix operator ':=' to represent this feature. The left operand of the assignment operator denotes the property or association end that is the target of the assignment. The right operand is an expression that determines the value to be assigned. For the assignment target, we can simply refer to *AssociationEndCallCS*, the OCL grammar rule for identifying association ends. For the value expression, we use the the new construct *AsoqExpressionCS* that is defined later in this chapter. The main difference to its OCL counterpart is that it may have side effects on the system state.

For the syntax specification itself, the full attribute tree grammar uses an extended Backus-Naur [87] form:

**PropertyAssignmentCS** ::= AssociationEndCallCS ':=' AsoqExpresisonCS

In addition to the syntax, the full attribute grammar also specifies the environments of the sub-expressions and the construction rules for the syntax tree in a declarative form.

The environment attribute *.env* of a grammar rule defines the evaluation context for sub-expressions. In the case of an property assignment, the environment for the sub-expressions does not change; it is simply forwarded without change:

```
AssociationEndCallCS.env = PropertyAssignmentCS.env
AsoqExpressionCS.env = PropertyAssignmentCS.env
```

The environment part is more interesting for operations such as variable declarations, where the environment is actually modified.

For UML action semantics, the construction of the abstract syntax tree translates to the construction of an operational set of connected *Action* objects. Here, the corresponding action object is a *CreateLinkAction*:

```
PropertyAssignmentCS.ast : CreateLinkAction
```

The *value* and *endData* properties of the *CreateLinkAction* still need to be connected to the corresponding properties of the operands. For the *value*, this means a simple connection to the output pin of the left hand ASOQ expression. Unfortunately, the *InputPin* and *DataFlow* objects — needed to connect actions — make the specification relatively verbose, despite its simplicity:

```
PropertyAssignmentCS.ast.isReplaceAll = true
PropertyAssignmentCS.ast.value = InputPin
PropertyAssignmentCS.ast.value.flow = DataFlow
PropertyAssignmentCS.ast.value.flow.source =
  AsoqExpressionCS.outputPin
```

For the *endData*, linking is a bit more complicated. Actually, the assignment target consists of two parts: A reference to the association end that will be assigned and an expression that determines the target instance of the assignment.

In the action semantics specification, the association end and target instance are referenced in the *end* and *value* properties of a *LinkEndData* object. The abstract syntax tree node of the *AssociationEndCallCS* holds this information in the properties *referredAssociationEnd* and *expression*. The construction of the *LinkEndData* object and connecting the referenced association end is relatively simple, since the same type is used in the action semantics specification and the OCL specification:

```
PropertyAssignmentCS.ast.endData : LinkEndData
PropertyAssignmentCS.ast.endData : LinkEndData
PropertyAssignmentCS.ast.endData.end =
  AssociationEndCallCS.ast.referredAssociationEnd
```

For the target object we need to construct an intermediate *OclAction* instance in addition to the *InputPin* and *DataFlow* objects connecting that are needed to connect actions. The *OclAction* wraps the OCL expression denoting the target object in an action that can be connected to the input pin of the link end data.

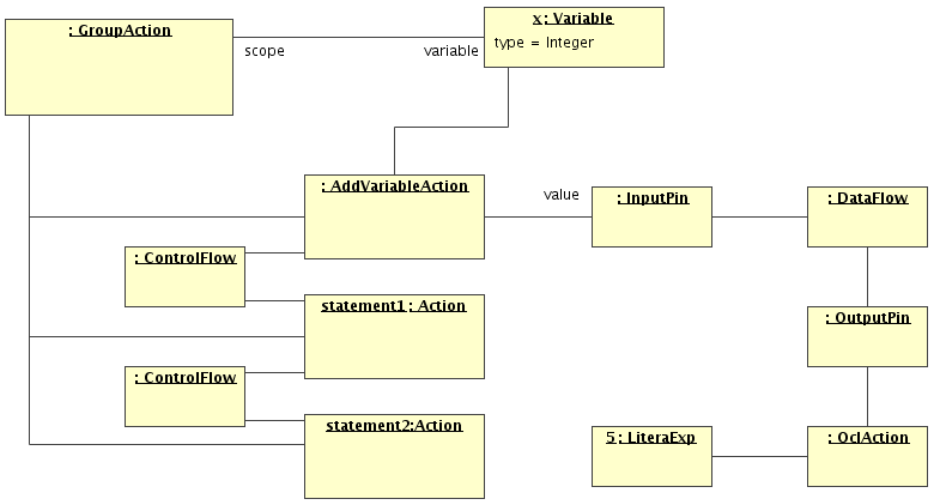


Figure 6.3: Mapping of a *Block* to a *GroupAction* object

```
PropertyAssignmentCS.ast.endData.value : InputPin
PropertyAssignmentCS.ast.endData.value.flow = DataFlow
PropertyAssignmentCS.ast.endData.value.flow.source =
  OutputPin
PropertyAssignmentCS.ast.endData.value.flow.source
  .action = OclAction
PropertyAssignmentCS.ast.endData.value.flow.source
  .action.expression = AssociationEndCallCS.source
```

### 6.3.2 Blocks, Variables, and Statements

The top level concept of ASOQ is a block, consisting of a list of variable declarations, followed by a sequence of statements. Figure 6.3 shows an action object diagram for the following block:

```
let
  x = 5
in
-- Statement 1;
-- Statement 2
```

In the following annotated syntax definition, *SimpleNameCS* and *OclExpressionCS* from the OCL Grammar are referenced. *OclActionCS* simply

wraps the *OclExpression* abstract syntax tree in an *OclAction* object. The OCL variable declaration syntax cannot be used without modification here, since it does not permit to initialize a variable with a newly created object. The construction of the rules is similar to the previous section; most space is spent for the correct chaining of actions.

**BlockCS** ::= StatementListCS

```
StatementListCS.env = BlockCS.env
BlockCS.ast : GroupAction
BlockCS.ast.subaction = StatementListCS.ast
```

**BlockCS** ::= LetBlockCS

```
LetBlockCS.env = BlockCS.env
BlockCS.ast : GroupAction
BlockCS.ast.subaction = LetBlockCS.ast
```

**LetBlockCS** ::= 'let' AsoqVariableDeclarationCS LetBlockSubCS

```
LetBlockSubCS.env =
  LetBlockCS.env.nestedEnvironment().addElement(
    AsoqVariableDeclarationCS.variable.name,
    AsoqVariableDeclarationCS.variable,
    false)
LetBlockCS.ast = LetBlockSubCS.ast->prepend(
  AsoqVariableDeclarationCS.ast)
LetBlockCS.ast->first().consequent : ControlFlow
LetBlockCS.ast->first().consequent.sucessor =
  LetBlockSubCS.ast->first()
```

**LetBlockSubCS** ::= ',' AsoqVariableDeclarationCS LetBlockSubCS[2]

```
LetBlockSubCS[2].env =
  LetBlockSubCS.env.nestedEnvironment().addElement(
    AsoqVariableDeclarationCS.variable.name,
    AsoqVariableDeclarationCS.variable,
    false)
LetBlockSubCS.ast = LetBlockSubCS[2].ast->prepend(
  AsoqVariableDeclarationCS.ast)
LetBlockSubCS.ast->first().consequent : ControlFlow
LetBlockSubCS.ast->first().consequent.sucessor =
  LetBlockSubCS[2]->first()
```

**LetBlockSubCS** ::= 'in' StatementListCS

```
StatementList.env =
  LetBlockSubCS.env.nestedEnvironment().addElement(
    AsoqVariableDeclarationCS.variable.name,
    AsoqVariableDeclarationCS.variable,
    false)
LetBlockSubCS.ast = StatementListCS.ast->prepend(
  AsoqVariableDeclarationCS.ast)
LetBlockSubCS.ast->first().consequent : ControlFlow
LetBlockSubCS.ast->first().consequent.sucessor =
  StatementList->first()
```

**AsoqVariableDeclarationCS** ::= simpleNameCS : typeCS = AsoqExpressionCS

```
AsoqExpression.env = AsoqVariableDeclarationCS.
VariableCS.ast : WriteVariableAction
VariableCS.ast.variable : Variable
VariableCS.ast.variable.name : String = simpleNameCS
VariableCS.ast.variable.in : InputPin
VariableCS.ast.variable.in.flow : DataFlow
VariableCS.ast.variable.in.flow.source : OutputPin
VariableCS.ast.variable.in.flow.source.action =
  AsoqExpressionCS
```

**StatementListCS** ::= StatementCS

```
Statement.env = StatementListCS.env
StatementListCS.ast = Sequence{StatementCS.ast}
```

**StatementListCS** ::= StatementCS ';' StatementListCS[2]

```
StatementListCS.ast =
  StatementListCS[2].ast->prepend(StatementCS.ast)
StatementCS.ast.consequent : ControlFlow
StatementCS.ast.consequent.successor =
  StatementListCS[2].ast->first
```

**StatementCS** ::= BlockCS

```
BlockCS.env = StatementCS.env
StatementCS.ast = Block.ast
```

**StatementCS** ::= IfStatementCS

```
IfStatementCS.env = StatementCS.env
StatementCS.ast = IfStatementCS.ast
```

**StatementCS** ::= WhileStatementCS

```
WhileStatementCS.env = StatementCS.env
StatementCS.ast = WhileStatementCS.ast
```

**StatementCS** ::= AssignmentCS

```
AssignmentCS.env = StatementCS.env
StatementCS.ast = AssignmentCS.ast
```

**StatementCS** ::= AsoqExpressionCS

In order to avoid ambiguities with OCL *let* and *if* constructs, all other statement rules must take precedence to this one.

```
AsoqStatementCS.env = StatementCS.env
StatementCS.ast = AsoqExpressionCS.ast
```

### 6.3.3 OclAction

A simple rule simplifies the integration of OCL expressions in the action syntax by creating an *OclAction* instance. The created *OclAction* can be directly connected to input pins in the abstract syntax specifications of other rules:

**OclActionCS** ::= OclExpressionCS

```
OclExpressionCS.env = OclActionCS.env
OclActionCS.ast : OclAction
OclActionCS.ast.expression = OclExpressionCS.ast
```

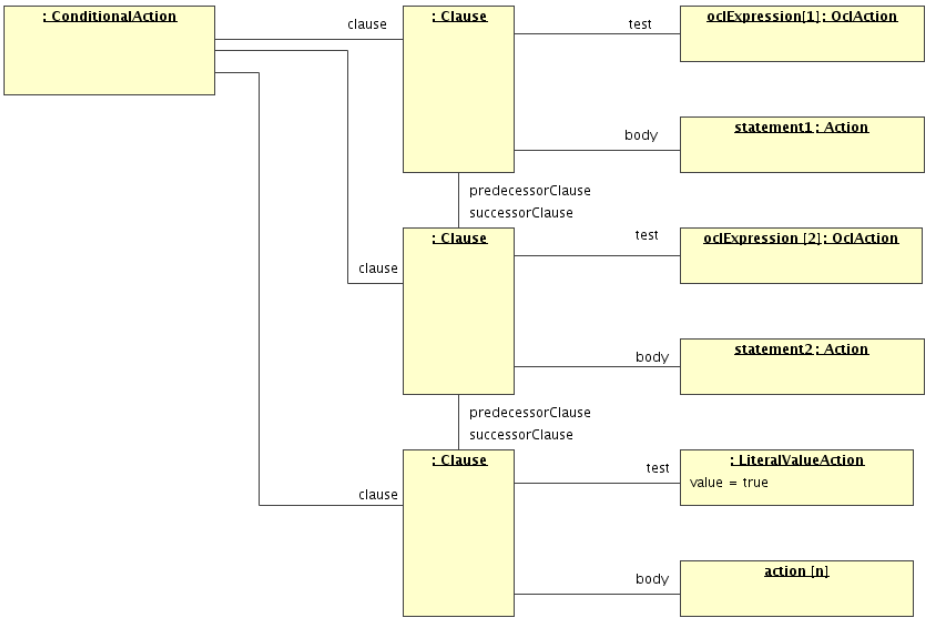


Figure 6.4: Mapping of an if Statement to a *GroupAction* Object

### 6.3.4 if-then-else

The OCL already contains an *if-then-else* control structure, but at ASOQ statement level it is not necessary that a value is returned, thus the *else* part may be omitted. To avoid deep nesting for multi-way decisions, while still keeping the syntax consistent with OCL, we add an *elseif* construct. Other languages such as C or PASCAL do not need this because they use an explicit general block structure—instead of implicitly opening a block that must be terminated with *endif*.

An example code snippet, matching the multi-way decision sample of the action semantics specification, looks as follows:

```
if factor = 1 then
  -- Some action 1
elseif factor = 2 then
  -- Some action 2
else
  -- Some action 3
endif
```

The resulting action semantic object diagram is depicted in figure 6.4.

**IfStatementCS** ::= 'if' ClauseCS 'endif'

```
ClauseCS.env = IfStatementCS.env
IfStatementCS.ast : ConditionalAction
IfStatementCS.ast.clause = Set{ClauseCS.ast}
```

**IfStatementCS** ::= 'if' ClauseCS ElseCS 'endif'

```
IfStatementCS.ast : ConditionalAction
IfStatementCS.ast.clause =
  ElseifCS.ast->prepend(ClauseCS)
ClasuseCS.ast.successorClause = ElseifCS->first()
```

**ClauseCS** ::= OclActionCS 'then' BlockCS

```
ClauseCS.ast : Clause
ClauseCS.ast.test = OclActionCS.ast
ClauseCS.ast.body = BlockCS.ast
```

**ElseCS** ::= 'else' DefaultClauseCS

```
ElseCS.ast = Sequence{DefautlClauseCS.ast}
```

**ElseCS** ::= ElseifCS 'else' DefaultClauseCS

```
ElseCS.ast = ElseifCS.ast->append(DefautlClauseCS.ast)
DefaultClauseCS.ast.predecessorClause =
  ElseifCS.any(successorClause->isEmpty)
```

**DefaultClauseCS** ::= BlockCS

```
DefaultClauseCS.ast : Clause
DefaultClauseCS.ast.test : LiteralActionCS
DefaultClauseCS.ast.test.value = true
DefaultClauseCS.ast.body = BlockCS.ast
```

**ElseifCS** ::= 'elseif' ClauseCS

```
ClauseCS.env = ElseifCS
ElseifClauseCS.ast = Sequence{ClauseCS}
```

**ElseifCS** ::= 'elseif' ClauseCS Elseif[2]CS

```
ClauseCS.env = ElseifCS.env
ElseIfCS[2].env = ElseifCS.env
ElseifCS.ast = Elseif[2]CS.ast->prepend(ClauseCS.ast)
ClauseCS.ast.successorClause = ElseifCS[2].ast->first()
```

### 6.3.5 while

A while loop is fortunately quite straightforward to construct:

**WhileStatementCS** ::= 'while' OclActionCS 'do' BlockCS 'enddo'

```
WhileStatementCS.ast : LoopAction
WhileStatementCS.ast.clause : Clause
WhileStatementCS.ast.clause.test = OclExpressionCS.ast
WhileStatementCS.ast.clause.body = BlockCS.ast
```

### 6.3.6 Method Invocations and ASOQ Expressions

Non-query Operations with and without parameters can be invoked just like query operations are invoked in OCL. All parameters must be OCL expressions, it is not possible to nest invocation actions. An optional path to the operation may be provided as an OCL expression. If so, the operation invocation must be separated from the path expression with an exclamation mark. An actual ASOQ code snippet may look as follows:

```
Publication.allIstances()->one
  (title='Cold Fusion')!vote()
```

Figure 6.5 shows the relevant parts of the action semantics used in the following syntax definition.

**AsoqExpressionCS** ::= simpleNameCS ArgumentsCS

```
ArgumentCS.env = AsoqExpressionCS.env
AsoqExpressionCS.ast : CallOperationAction
AsoqExpressionCS.ast.argument : Sequence(InputPin)
```

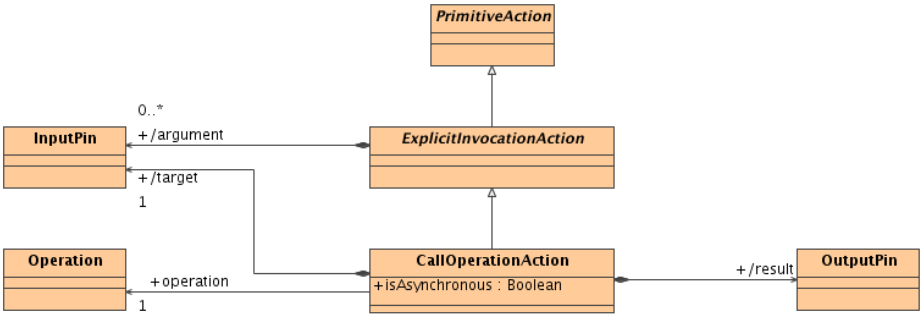


Figure 6.5: Messaging Actions Referenced in ASOQ

```
AsoqExpressionCS.ast.argument->size() =
  ArgumentCS.ast->size()
AsoqExpressionCS.ast.argument->forAll
  (i|i.flow : Dataflow)
Sequence{1..ArgumentCS->size()}->forAll(i|
  AsoqExpressionCS.ast.argument->at(i).flow.source =
    ArgumentCS->at(i).result)
AsoqExpressionCS.ast.operation =
  env.lookUpImplicitNqOperation(
    simpleNameCS.ast,
    ArgumentCS.ast->collect(result.type))
```

**AsoqExpressionCS** ::= OclActionCS '!' simpleNameCS ArgumentsCS

```
ArgumentCS.env = AsoqExpressionCS.env
AsoqExpressionCS.ast : CallOperationAction
AsoqExpressionCS.ast.target = OclAction
AsoqExpressionCS.ast.target : InputPin
AsoqEspressionCS.ast.target.flow : DataFlow
AsoqExpressionCS.ast.target.flow.source =
  OclActionCS.result
AsoqExpressionCS.ast.operation =
  OclActionCS.result.type.lookUpNqOperation(
    simpleNameCS.ast,
    ArgumentCS.ast->collect(result.type))
AsoqExpressionCS.ast.argument : Sequence(InputPin)
AsoqExpressionCS.ast.argument->size() =
  ArgumentCS.ast->size()
```

```
AsoqExpressionCS.ast.argument
  ->forAll(i|i.flow : Dataflow)
Sequence{1..ArgumentCS->size()}->forAll(i|
  AsoqExpressionCS.ast.argument->at(i).flow.source =
    ArgumentCS->at(i).result)
```

**AsoqExpressionCS** ::= OclActionCS

```
OclActionCS.env = AsoqExpressionCS.env
AsoqExpressionCS.ast = OclActionCS.ast
```

**ArgumentsCS** ::= '(' ')'

```
ArgumentCS.ast : Sequence(OclAction)
ArgumentCS.ast = Sequence{}
```

**ArgumentListCS** ::= OclActionCS

```
OclActionCS.env = ArgumentListCS.env
ArgumentListCS.ast = Sequence{OclActionCS.ast}
```

**ArgumentListCS** ::= OclActionCS ',' ArgumentListCS[2]

```
OclActionCS.env = ArgumentListCS.env
ArgumentListCS[2].env = ArgumentListCS.env
ArgumentListCS.ast =
  ArgumentListCS[2].ast->prepend(OclActionCS.ast)
```

### 6.3.7 Variable Assignments

The action semantics classes modeling assignments are depicted in figure 6.6. Variable assignments are simple to handle. It is only required to look up the name in the environment and then construct an according *WriteVariableAction*:

**VariableAssignmentCS** ::= SimpleNameCS ':=' AsoqExpressionCS

```
SimpleNameCS.env = VariableAssignmentCS.env
AsoqExpressionCS.env = VariableAssignmentCS.env
VariableAssignmentCS.ast : AddVariableValueAction
```

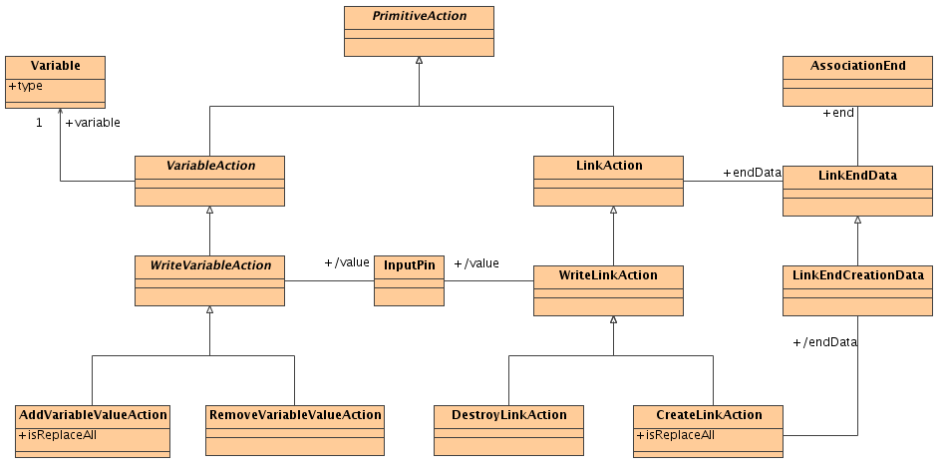


Figure 6.6: Read/Write Actions Referenced in ASOQ

```
VariableAssignmentCS.ast.isReplaceAll = true
VariableAssignmentCS.ast.variable =
  env.lookUpASVariable(SimpleNameCS)
VariableAssignmentCS.ast.value = InputPin
VariableAssignmentCS.ast.value.flow = DataFlow
VariableAssignmentCS.ast.value.flow.source =
  AsoqExpressionCS.outputPin
```

## 6.4 ASOQ Utilization in the Infolayer System

The main purpose of the ASOQ capabilities of the Infolayer system is to provide operational definitions of operations and writeable derived attributes. The operations may either be explicitly invoked via the user interface by pressing a button, or may be called from other operations with side effects. In addition to general user defined operations, there are a few predefined callback methods that are called by the user interface for notifications about changes.

### 6.4.1 Operation and Property Implementation

Operations and virtual properties can be implemented similar to query operations and derived properties, using the same tags, except that it is per-

mitted to use the ASOQ extensions instead of plain OCL.

In the user interface, methods with and without side effects are displayed as buttons. The only difference is that query buttons trigger a HTTP GET request, whereas non-query buttons trigger a HTTP POST request.

As for query methods, the implementation can be attached to the operation using the tagged value *il-implementation*. However, for non-query methods, the ASOQ syntax must be used. As an alternative to the tagged value, the implementation can be included in the documentation of the method, which often provides a larger input field in UML tools. If so, it must be prefixed with the marker *@il-implementation*.

In addition to operations, the Infolayer supports the implementation of derived and "virtual" attributes and association ends using ASOQ expressions.

The implementation of derived properties can be attached to the model using the tagged value *il-getter*, as illustrated in the previous chapter. Virtual properties are derived properties that are writeable. The following tagged values are supported:

**il-setter** The contained ASOQ expression is responsible for storing the new value of the property, accessible via the local variable *value*.

**il-adder** Add *value* to the existing values of the property.

**il-remover** Remove *value* from the existing values of the property.

At least, either the setter or the adder and the remover must be defined. The alternatives are provided to avoid unnecessary conversions at the persistent storage layer.

### 6.4.2 Predefined Callback Methods

The predefined class *Object* contains the callback operations *onChange()*, *onCreate()*, and *onDelete()* which are invoked automatically by the Infolayer system. The implementation of those operations may have side effects:

**onChange** is invoked on an object when the object was changed in the user interface.

**onCreate** is invoked when a new instance of a class was creates via the user interface.

**onDelete** is invoked when an object is deleted via the user interface.

Those methods can be overwritten with customized behavior. For instance, to set a "lastChangeBy" attribute for an object by implementing the onChange method can be implemented as follows:

```
lastChangeBy :=  Infolayer.getCurrentUser();
```

## 6.5 Summary

In this chapter we have shown how OCL expressions can be cleanly integrated into the UML action semantics framework by introducing a specialized OclAction. We have created an action semantics surface language that embeds OCL expressions without needing modifications of the OCL grammar or the OCL semantics. The syntax is kept consistent with the OCL syntax. By not duplicating parts of the action semantics framework, the additional constructs can be limited to a small set. The resulting ASOQ language provides a clean solution to the surface language and functionality overlap problem of the current action semantics specification. It is not limited to the Infolayer system, it may also be used where similar problems arise, such as the YATL transformations system [96].

In addition to defining ASOQ, this chapter illustrates how the gained functionality can be utilized in the Infolayer system, using the same tagged value mechanism as for non-query operations.

With ASOQ, the Infolayer system allows assignments and operation calls with side effects, such as the creation of new instances, as announced in chapter 3. Theoretically, modifications of the M1-model would also be possible with this mechanism, since UML uses the same modeling constructs for both levels, making ASOQ also suitable for refactoring. However, currently the M1-model is not writeable. Yet, this is only a limitation of the underlying implementation, and not a general limitation of Infolayer approach.

In the next chapter, we will consider simple access permission and customization options available in the Infolayer system.

# Chapter 7

# Transformations

The default HTML presentation of the Infolayer system illustrated in chapter 3 and the simple customization options discussed in the previous chapter make it possible to set up a working prototype quickly, given a UML class diagram. However, those mechanisms are neither sufficient to generate a sophisticated customized HTML presentation nor able to generate other XML formats—such as the Wireless Markup Language (WML) [40] or the Resource Description Framework (RDF) [71]. If we want to build a processing chain for object oriented systems with the capability to generate arbitrary XML (requirement III in the introduction), we still need a replacement for the last link in the chain, the XML Stylesheet Language Transformations (XSLT) [23].

XSLT templates transform a source XML document to a target document of any desirable format [67]. XSLT is itself an XML-based language and consists of XML elements for writing to the target document such as `<xslt:text>` or `<xslt:element>`, as well as control structures such as conditionals (`<xslt:if>`, `<xslt:choose>`) and loops (`<xslt:for-all>`). It is also possible to write literally included XML fragments to the target document. To access the source document that is to be transformed, XSLT makes use of another W3C standard, the XML path language (XPath). XPath is an expression language with the main purpose of being able to address any part of an XML document. It also provides simple numeric, boolean and string operations. In XSLT, XPath expressions are supported in `test` or `select` attributes, providing the conditions for conditionals or

source selectors for write-elements.

Since the transformation source for XSLT is an XML file, XSLT cannot be applied to the state of an object oriented system directly; it is required that the system state is translated to some kind of generic intermediate XML tree before. The intermediate XML format may be a standardized XML format for the serialization of objects or a custom format, such as in the UWE web engineering approach [69].

The problem with this requirement is that for the serialized form, query operations may become more complicated. As already discussed in chapter 1, XML does not directly support arbitrary associations [115], hence links may only be accessible via key reference matches, and XPath remains at a pure syntactical level [74]. Expecting a developer to obtain detailed knowledge of the serialization process does not seem very user friendly.

There are several alternative approaches for XML generation. Mechanisms such as Active Server Pages (ASP) [26], Java Server Pages (JSP) [82], PHP [97] or ColdFusion [14] are based on XML templates with embedded programming language fragments. The programming language fragments are used for control structures and to fill the templates with content from query operations. Unfortunately, the programming language fragments are platform dependent and hence not suited to specify XML generation in a platform independent manner.

To solve the XML generation problem in a platform-independent manner, it seems to be a natural choice to build on XSLT, but to use OCL as expression language instead of XPath, providing direct access to the system state without an intermediate XML representation. Although OCL was originally proposed as a pure constraint language, its ability to navigate the model and form collections makes it perfectly suitable as query language [30]. Thus, OCL can directly be used to replace XPath expressions that return simple values, such as the `test` attribute in conditionals and the `select` attribute in content generation elements.

For instance, the following XML fragment shows a possible serialization of a set of Book objects from the department sample presented in chapter 1:

```
<list>
  <Book>
```

```
    <title>Mastering Quantum Physics in 24 Hours</title>
    <price>49.50</price>
  </Book>
  <Book>
    <title>Building a Particle Accelerator</title>
    <price>16.70</price>
  </Book>
</list>
```

The XPath expression contained in the XSLT element

```
<xsl:value-of select="sum(//Book/@price)" />
```

could simply be replaced by the OCL expression

```
Book->allInstances().price->sum()
```

A corresponding language, mainly derived from XSLT, eliminates the need of an intermediate XML format—while still able to utilize huge parts of XSLT knowledge.

The replacement of XPath by OCL is not the only difference between XSLT and the Infolayer template mechanism. The processing model was adopted to better fit the object oriented architecture of the Infolayer model. Since Infolayer templates are always applied on the server side, more specific rules how different template files can interact and a schema for URL resolution can be provided. The most significant change in addition to using OCL was the addition of several server sided elements that make it possible to interact with the server, for instance to modify the content of the system. A minor difference to XSLT is that hyphens in identifiers were replaced by a "camel syntax" (for technical reasons, but also improving consistency with most other XML standards); for instance the XSLT element `<value-of>` was renamed to `<valueOf>` and `<for-all>` was renamed to `<forAll>`. The `test` and `select` attributes have both been renamed to `expr`.

The first section of this chapter describes the general architecture of the Infolayer template mechanism and provides an overview of the URL resolution mechanism that acts as a replacement for the source fragment selection in

XSLT. In the second section, we discuss the general XML template elements that are derived from XSLT counterparts.

Since the Infolayer templates are always applied on the server side, it was possible to design additional high level interaction elements that are described in separate sections. Interaction details of the XML code generated by those elements with the server are described in section 7.5. Finally, the completeness of the template mechanism is discussed in section7.6.

## 7.1 Template Language Architecture

Template files consist of literal XML content (e.g. regular XHTML) and special template elements. The template elements are distinguished from "regular" XML elements using dedicated XML namespaces.

The namespace mechanism was developed by the W3C in order to make it possible to mix different XML languages in a single XML file without ambiguities.

The namespace for the general Infolayer template elements is `http://infolayer.org/templates`. Additional specialized XHTML template elements reside in the namespace `http://infolayer.org/templates/html`. The XHTML template namespace also contains the basic template elements for convenience, of which some are overwritten with functionality adopted to the special needs for generating XHTML output.

Here, the XML namespace prefix "`t`" will be used for all Infolayer template elements in order to avoid confusion with XML elements of the target language. A corresponding namespace declaration is always required, even if it is omitted in some code fragments below. It is not required that the namespace prefix is actually set to "`t`", any other prefix can be used if declared properly.

A simple HTML template file may look as follows:

```
<html xmlns:t="http://infolayer.org/templates/html">
  <head><title>Books</title></head>
  <body>The sum of all book prices is:
    <t:valueOf expr="Book.allInstances().price->sum()" />
```

```
  </body>
</html>
```

### 7.1.1 Template Processing Model

XSLT style sheets contain a number of template rules. Each rule contains an XPath selector, denoting the part of the source file the rule applies to. The XSLT processor determines the best match and applies the corresponding template. In the template, `<apply-templates>` can be used to apply the template selection process on the currently processed node.

When applying a template to the state of an object oriented system, the notaion of a "current node" in the source XML tree can be replaced by a "current object". An OCL expression could be used as selector expression. However, this kind of centralized dispatching approach, where a set of template rules for different classes or expressions is listed in a single template file, does not seem to fit well with the object oriented paradigm.

Instead, from an object oriented perspective, one expects an object to know how to transform itself. Thus, templates should be attached to a class in some way. Moreover, one will want to reuse parts of the functionality for different templates. For instance, `Publication` details should be reusable in different types of lists.

An option to satisfy both goals is to use a single file for each template rule. The file is simply stored in a directory named after the class it belongs to. The filename itself is the name of the template, allowing an unlimited number of templates for different purposes per class. This approach provides several advantages:

- When applying a template, the context is always an instance of the given class, enabling type safe expressions.
- Templates can be searched in the directories of more general classes if a specialized template is not found.
- Templates look much more similar to the anticipated target format, simplifying the initial creation with editors specialized for this format.

The Infolayer system uses a dynamic URL resolution mechanism for late binding that is explained in section 7.1.4. Here, we will start with the simpler static URL resolution mechanism, which is useful to generate pages that do not clearly depend on a particular object or class, such as the start page `index.html`.

### 7.1.2 Static URL Resolution

For static templates and non-template files, the path part of an URL denotes the name of the source file directly. In general, the path consists of a fixed part denoting the Infolayer servlet, and a variable part. The variable part is directly translated into a file name in the file system that is relative to the directory `html/static/` in the servlet root directory. Here, `html/static/` is used even for non-HTML formats such as PDF files or image files since they are usually closely connected to HTML pages. The only (predefined) exception are templates for the Wireless Markup Language (WML), which reside in a separate directory `wml/static/`.

For instance, if we assume that `http://localhost:8080/department/` is the base URL of the servlet, and `$basepath` denotes the installation directory of the Servlet in the file system, the path

```
http://localhost:8080/department/index.html
```

will be resolved to

```
$basepath/html/static/index.html
```

Static templates can be used and accessed just like "regular" static web pages, except that they should contain well-formed XML code. The handling of files depends on the file extension of the requested URL. The extensions `.html`, `.htm`, and `.xhtml` denote HTML-Templates. The extension `.xml` denotes plain XML templates without content format specific extensions. Additional options are presented in section 7.4.

### 7.1.3 The Page Evaluation Context

An important concept for Infolayer templates is the current context. The evaluation context of Infolayer template elements can be accessed in template expressions using the OCL pseudo variable `self`. Initially, the value of self is determined from the `self` URL parameter. If the `self` parameter is not given, `self` is set to `OclUndefined`. For static templates, the type of `self` always is `OclAny`.

Using the URL parameter `self`, it is possible to set an initial evaluation context from outside. The evaluation context can be any OCL literal, including the Infolayer extension for object references, where single objects are identified by a dollar sign followed by the object ID.

However, it is not possible to use general OCL expressions for the context parameter for security reasons—since the permissions are evaluated by the user interface elements, it would be possible to spy on the whole system content this way.

### 7.1.4 Dynamic URL Resolution

Dynamic URLs consist of the servlet base URL, "auto", a file extension denoting the MIME type, and an URL property `self`, holding the desired class name or object ID to specify the template search path and the evaluation context. The optional URL property `template` denotes the name of the template. If the `template` property is missing, "`default`" is assumed as template name.

Those dynamic URLs are resolved to a template file name as follows:

1. A base directory is selected based on the type of `self`. If `self` is a Class, the base directory is `$basepath/html/class`, where `$basepath` denotes the root directory of the servlet. For objects, the base directory is `$basepath/html/object`, for collections, it is `$basepath/html/collection`, and for all other types simply `$basepath/html` (WML files are searched in `$basepath/wml/...` instead).

2. The name of the (element-) type is appended to the base directory.

3. A file with the name of the template and the extension provided for "auto" is searched in the resulting directory.

4. If a corresponding template file is found, it is processed; otherwise, the same search is performed for the supertype of the current (element-) type.

If a template is still not found, the same search is performed in the `resource` directory of the Infolayer JAR file, which contains predefined generic templates for classes, objects and primitive types. Those predefined templates are used to generate the output described in section 3.1.1.

For instance, the URL

```
http://localhost:8080/department/auto.html?self=Person
```

will be resolved to the path

```
$basepath/html/class/Person/default.html
```

If the template is not found, `Object` and then `OclAny` are tried. If a template is still not found, the predefined default template is used.

Custom templates can either hide the existing templates, or have a different name. In the latter case, the template name must be appended to the URL using the parameter `template`. If a `template` URL parameter is present, the file corresponding to the given name is searched instead of the `default` file. The type of the evaluation context is set automatically depending on the directory where the template resides.

## 7.2 General Template Elements

The Infolayer system provides support for template elements that are independent of the chosen target format. Those elements are described in this section. Most of the elements are similar to XSLT elements, but there are also elements that do not have a direct XSLT counterpart, such as `switch` and `case`.

### 7.2.1 valueOf

A simple Infolayer template element is the `<t:valueOf>` element. It must contain an attribute named `expr`, providing an OCL expression that is evaluated when the page is delivered to the user. For instance, the expression

```
<t:valueOf expr="Infolayer.getCurrentUser()" />
```

inserts the current user, replacing the `valueOf` element on the page sent to the browser.

The complete source code for a working static "`whoami.html`" page is:

```
<html xmlns:t="http://infolayer.org/templates/html">
  <head><title>Who Am I</title></head>
  <body>
    <t:valueOf expr="Infolayer.getCurrentUser()" />
  </body>
</html>
```

If this page is stored at the location

```
$basepath/html/static/whoami.html
```

it will be accessible using the URL

```
http://localhost:8080/department/whoami.html
```

If the user currently logged in is "John Smith", and the `toString()` method of the user class returns the family name and the given name, separated by a comma, the generated XML code will be:

```
<html xmlns:t="http://infolayer.org/templates/html">
  <head><title>Who Am I</title></head>
  <body>
    Smith, John
  </body>
</html>
```

### 7.2.2 The Evaluation Context

For dynamic templates, the initial type of `self` is inferred from the location of the template file. For example, for the template file

```
$basepath/html/object/Person/default.html
```

the initial type of `self` will be `Person`.

The current context can be altered using the context element. The following example generates the same output as the previous example, but makes use of the context element:

```
<t:context expr="Infolayer.getCurrentUser()">
  <t:valueOf expr="self" />
</t:context>
```

Please note that the above example is only a fragment. To view this and following examples in a web browser, it is necessary to add the HTML root and body elements, as well as a namespace declaration, as shown in the previous example.

### 7.2.3 Iterating over Instances

The `<t:forAll>` element applies its child elements to all instances of a collection denoted by the `expr` attribute. The child elements are evaluated for each element of the collection, with the context adapted accordingly.

The following fragment illustrates the usage of `forAll`:

```
<t:forAll expr="Sequence{'a', 'b', 'c'}">
  This is Item <t:valueOf expr="self"> <br />
</t:forAll>
```

The generated XML code will look as follows:

```
This is Item a <br />
This is Item b <br />
This is Item c <br />
```

By setting the attribute `iterator`, it is possible to use a separate variable as iterator instead of `self`. The attribute `counter` can be used to declare a counter variable that will hold an integer counting the iterations:

```
<t:forAll expr="Sequence{'a', 'b', 'c'}"
       iterator="i" counter="c">

  Item No. <t:valueOf expr="c" />
  is <t:valueOf expr="i" /><br />
</t:forAll>
```

The above template will generated the following XML code:

```
Item no. 1 is a <br />
Item no. 2 is b <br />
Item no. 3 is c <br />
```

### 7.2.4 Conditional Processing

The Infolayer template mechanism provides three constructs for conditional template evaluation, `<t:if>`, `<t:choose>` and `<t:switch>`.

The `if`-element evaluates the OCL expression given in the `expr`-Parameter, and processes child elements only if the evaluation result is `true`.

The following example illustrates the usage of the `if`-element:

```
<t:if expr="user.oclIsUndefined()">
  No user is logged in!
</t:if>
```

The `<t:choose>` element makes it possible to handle a number of alternative options. It contains a number of `<t:when>` elements. If the `expr` attribute of a `when`-element evaluates to true, template processing continues with its child elements. Otherwise, the next `when`-element is considered. The `otherwise` element can be used to denote a default option. After the first match, processing of the `choose`-element is terminated; no further options are considered.

The `<t:switch>` element is structured similar to `choose`, but it has an additional `expr` attribute. In contrast to `choose`, the expressions of the `case`-child elements do not contain boolean conditions. Instead, the evaluation results of the `expr` attributes of the `switch` and `case` elements are compared. Template processing continues with the first matching case branch, all other sub elements are ignored. If no case matches, and an `<t:otherwise>` element is present, the `otherwise` branch is processed.

```
The Person class contains
<t:switch expr="Person.allInstances()->size()">
  <t:case expr="0">Zero</t:case>
  <t:case expr="1">One</t:case>
  <t:case expr="2">Two</t:case>
  <t:otherwise>Three or more</t:otherwise>
</t:switch>
instances.
```

### 7.2.5 Delegation to other Templates

The Infolayer system supports a call mechanism that allows to continue processing in a different template.

The simplest case of the `<t:call>` element is to include another static template, denoted by the `file` attribute:

```
<html xmlns:t="http://infolayer.org/templates/html">
  <head><title>FooBar</title></head>
  <body>
    <table>
      <tr>
        <td><t:call file="menu.html" /></td>
        <td>This is the Page content</td>
      </tr>
    </table>
  </body>
 </html>
```

The `call` element hands the current context (`self`) to the included template automatically.

```
<!-- common header -->
<html xmlns:t="http://infolayer.org/templates/html">
 <head><title>FooBar</title></head>
 <body>
  <table><tr><td>
     <b>Menu</b><br />
     <a href="foo.html">Foo</a><br />
     <a href="bar.html">Bar</a><br />
    </td><td>
<!-- Individual content starts here -->

     ...

<!-- Common footer -->
  </td></tr></table>
 </body>
</html>
```

Figure 7.1: Typical HTML File Structure

One problem we faced when designing the template mechanism was that for many XML documents, the outer frame had an identical structure, like sketched in figure7.1, but it was difficult to factor it into a single template. Separate header and footer templates would not be well formed XML because a header template would need to open several element not closed in the same file, and a footer template would need to close those elements without opening them. One option would have been some kind of callback mechanism, where the template of the outer frame is called with a parameter denoting which template to include for the actual content, but this seemed uncomfortable and error-prone.

Instead, the Infolayer makes it possible to continue processing of the child elements of the <t:call> element at a position denoted by the element <t:inner> in the called template. This way, it is possible to move more of the common structure to a single file, as depicted in figure 7.2.5.

For instance, the following XML code, holding the page structure and menu, could be stored as `frame.html` in the Infolayer static HTML template directory:

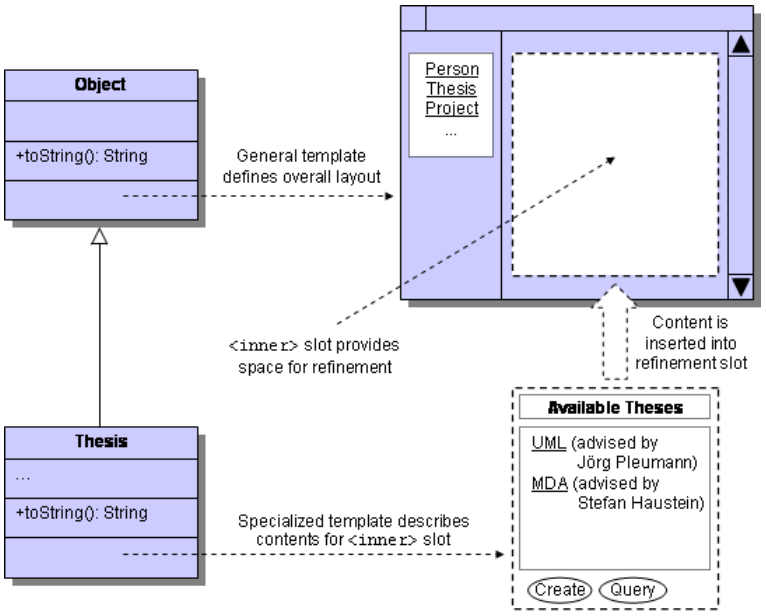


Figure 7.2: Page Composition from Different Templates with <t:inner>

```
<html xmlns:t="http://infolayer.org/templates/html">
  <head><title>FooBar</title></head>
  <body>
    <table><tr><td>
      <b>Menu</b><br />
        <a href="person.html">Person</a><br />
        <a href="thesis.html">Thesis</a><br />
        <a href="project.html">Project</a><br />
      </td><td>
        <t:inner />
      </td></tr></table>
  </body>
</html>
```

Now, using the `<t:call>` element, it is possible to arrange the menu around the actual page content:

```
  <t:call xmlns:t="http://infolayer.org/templates/html"
      file="menu.html">
<h1>Available Thesis</h1>
...
```

```
    </t:call>
```

Processing of the above template will result in the following XML code:

```
    <html xmlns:t="http://infolayer.org/templates/html">
      <head><title>FooBar</title></head>
      <body>
        <table><tr><td>
          <b>Menu</b><br />
          <a href="person.html">Person</a><br />
          <a href="thesis.html">Thesis</a><br />
          <a href="project.html">Project</a><br />
        </td><td>
           <h1>Available Thesis</h1>
...
        </td></tr></table>
      </body>
    </html>
```

In the Infolayer default HTML templates, the `call` element is used to generate a common menu on the left side of all pages. Thus, by modifying the static template `frame.html`, the appearance of all pages can be customized in a single central place.

The `expr` attribute can be used to set a different evaluation context for the called template. If the `call` element contains sub-elements, the previous context is restored.

Alternatively to the `file` attribute, the `template` attribute can be used to determine the included file based on the dynamic template resolution mechanism presented in section 7.1.4. The only difference is that the file extension is ".inc".

### 7.2.6 Variables and Parameters

In a template, variables can be declared using the element `<t:variable>`. The name of the variable is denoted by the `name` attribute. The initial value is determined by the `expr` attribute containing an OCL expression. If the `expr`-attribute is omitted, the initial value of the variable is `OclUndefined`.

A `type` attribute may be used to set the type of the variable. Variable declarations must not have child elements. Variables can be accessed by their name in OCL expressions.

Variables shadow other variables with the same name. The scope of a variable ranges from its declaration to the end tag of its immediate parent element.

If `<t:param>` is used in a template embedded via `<t:call>`, the parameter values can be set using `<withParam>` child elements of the `call`-element, like in the following example:

```
<t:call xmlns:t="http://infolayer.org/templates/html"
        file="~/frame.html">
  <t:withParam name="title" expr="'foo'" />
This is the content of the foo page!
</t:call>
```

In the example, the tilde character ('~') is used to point to `$basepath/html/static` (or `$basepath/wml/static`, depending on the content format). In URLs, it can be used in URLs to reference the servlet base directory.

In the included file, parameters can be accessed using the operation `context.getParam(name: String)`:

```
<html xmlns:t="http://infolayer.org/templates/html">
  <t:param name="title" type="String" />
  <head><title><t:valueOf expr="title" /></title></head>
  <body><t:inner /></body>
</html>
```

For top-level templates, parameter values can be set using URL properties of the form "p-`name`=`OCL literal value`". The XHTML template element `<t:link>` described in section 7.3.2 also supports the `withParam` element.

The value of a variable can be changed using the `<t:assign>` element. The `name` attribute must contain the name of the variable, and the `expr` attribute denotes the new value of the variable.

### 7.2.7 Access to Request Information and Cookies

s It is possible to access information about the HTTP request that triggered the current template evaluation in OCL expression inside XML templates. The predefined variable `request` provides access to request information such as the request URL or URL properties:

```
The request URL for this page was
<t:valueOf expr='request.getUrl()' />
```

The `request` variable also provides access to the current language settings of the browser using the `getLang()` method and access to cookies using the `getCookie()` and `setCookie()` methods.

The methods available for `IlRequest` and `IlUrl` are described in detail in the appendix.

### 7.2.8 Formatting and Dynamic Includes

For the `valueOf` template element, it is possible to specify one of the format strings described in the previous chapter in the `format` attribute.

If possible, the evaluation result is then inserted according to the given format.

For objects, the format string is interpreted as template name, and the template with the given name is included. To distinguish the included templates from regular templates, representing full pages, ".inc" is used as file extension instead of the default extension for the given content format (e.g. ".html").

### 7.2.9 Evaluated XML Attributes

The XML namespace standard allows to assign namespaces not only to elements, but also to attributes. By default, attributes are in no namespace, since their interpretation is usually depending on the owning element. In the Infolayer system, the option to assign namespaces to attributes is used for dynamic attribute content.

If an attribute of any element is in one of the Infolayer template namespaces, the content is not taken literally, but treated as OCL expression that is parsed and evaluated. In the generated page, the attribute content is then replaced by the evaluation result.

Please note that all attributes directly interpreted by the Infolayer system are excluded from this feature, except when explicitly noted otherwise.

### 7.2.10 Dynamic Content Construction

The infolayer provides a set of constructs that make it possible to generate XML content dynamically:

<t:element>: Generates the XML element denoted by the `name` and `namespace` attributes.

<t:attribute>: Generates an XML attribute denoted by the `name` and `namespace` attributes. The attribute value is determined by the element content.

<t:comment>: Generates an XML comment. The comment is determined by evaluating the element body.

<t:text> Generates an XML text node. The content is determined by evaluating the element body.

## 7.3 XHTML-Specific Template Features

In addition to the general XML template elements, which do not make assumptions about the XML language they are applied to, the Infolayer contains several template elements that are tailored towards the generation of XHTML code.

These XHTML template elements reside in a separate namespace, which is `http://infolayer.org/templates/html`. This namespace also provides access to the general XML template elements, so it is not necessary to declare separate prefixes for access to both.

### 7.3.1 Additional Capabilities of `valueOf`

The XHTML <t:valueOf> template element provides additional support for collection based HTML-specific format options such as `ol`, `li` or `table`, as described in appendix C. Those format options cannot be supported by generic XML template elements since they need to generate code that depends on the target XML language.

### 7.3.2 Hyperlinks to Objects

The `link` template element generates an `<a>` XHTML element, where the `href` attribute points to the URL of the object denoted by the OCL expression contained in the `expr` attribute of the element. By specifying a `template` attribute, it is possible to create a link to the given template instead of the default template.

If the expression evaluates to `OclUndefined`, no `href` attribute is generated.

```
<t:link expr="Person">All instances
  of the class Person</t:link>
<t:link expr="Infolayer.getCurrentUser()
  template="verbose">Verbose Information
  about the current User</t:link>
```

Like `call`, the `link` element supports the `withParam` sub element to transmit parameters to linked pages.

### 7.3.3 Properties and Forms

Current values of object properties can be included in pages using the `<t:valueOf>` element, but the `valueOf` element does not permit any user interaction. A more powerful alternative are the elements `<t:property>` and `<t:properties>`.

The `property` element displays a single object property which is denoted by the `name` attribute.

The `properties` element displays not just a single property, but a set of properties, including the property names, in an XHTML table. If the

`il-label` tagged value (or a fitting localized variant) is set for the property, this is used as label instead. The list of object properties to be displayed is controlled by the `properties` attribute. If not present, all public properties are displayed. Otherwise, the `names` attribute must contain a comma separated list of property names. The order of properties given in the list is preserved when the properties are displayed.

While the `properties` element can be used as a convenient alternative to `valueOf`, the real strength of the elements `property` and `properties` is their ability to generate input elements that can be used to edit the dynamic content of the Infolayer system, and to perform searches.

For this purpose, the elements must be embedded in a `<t:form>`-element. The `form`-element requires an attribute `type`, which specifies which kind of form to generate:

`edit`: Generates a form for editing an instance. `self` must be an object.

`query`: Generates an XHTML form for querying a class for matching instances. `self` must be a class.

For attributes, `property` and `properties` generate simple input fields. For association ends, selection lists and additional controls are generated, providing means for adding and removing linked instances.

Inside the form, the `<t:submit>` element creates a submit button, and the `<t:cancel>` element generates a cancel button. The labels of both buttons can be customized using the `label` attribute. The `followup` attribute instructs the Infolayer servlet which page to display when the given button was pressed. Table 7.1 shows an overview of the followup commands available.

The form and properties elements are used in the generic Infolayer default templates; 7.3 shows a simplified form of the default `edit` template.

### 7.3.4 Operations, Controls, and Actions

Buttons to execute operations that do not depend on parameters can be included in XHTML templates using the `<t:operation>` element. The `name` attribute denotes the name of the operation to be called. The name of the

| Followup Command | Description |
|---|---|
| `goto-url` | Sets the next page to the given URL, separated by a colon. |
| `goto-template` | Sets the next page to the given template, separated by a colon. The context object is the same as for the current page. |
| `call-url` | Sets the next page to the given URL, separated by a colon. The current page is saved on the URL stack. A return command can be used to return to the current address. |
| `call-template` | Sets the next page to the given template, separated by a colon. The context object is the same as for the current page. The current page is saved on the URL stack. A return command can be used to return to the current address. |
| `return` | Returns to a previous page saved on the URL stack. Default command for submit and cancel buttons. |
| `stay` | Stay on the current page. Default when an error occurs while processing the current page. |

Table 7.1: Followup Parameter Options for Submit and Cancel Buttons

```
<t:call xmlns:t="http://infolayer.org/templates/html"
    file="~/frame.html">

  <h1><t:valueOf expr="''+self" /></h1>

  <t:form type="edit">
    <p>
      <t:properties />
    </p>
    <t:submit /> <t:cancel />
  </t:form>
</t:call>
```

Figure 7.3: The Default Edit Template File "/instance/Object/edit.html"

operation is also used as a label for the button; if a `il-label` tagged value is set in the model, this is used instead. A local `label` attribute can be used to override `il-label`.

In contrast to properties, operations must not be embedded in a form element.

The return value of the operation is used to determine the page that is generated when the operation was performed. Using the `followup` attribute mechanism, the system can be instructed to use a specific template for displaying the evaluation result.

Similar to the `properties` element, the `operations` element generates a table containing all operations that are available for the current user.

The `<t:control>` element will generate a button that does not invoke an operation. It can be used to display a different page in the browser, for instance to display a link to the `edit` template consistently with operation buttons such as `clone` or `delete`.

The `<t:actions>` element displays a horizontal table containing all operations and statemachine actions available for `self`.

### 7.3.5 Login and Logout

For enabling users to login and logout, the template element `<t:login>` can be used. If the user is not logged in, it generates a table containing input elements for the user login and password.

Using the `type` attribute, it is possible to set one of the modes `input` or `select`. If the type is select, the user to log in can be selected from a list. Otherwise, the log in name must be entered in a text field. The following code fragment shows how to build a login dialog using a `form` element, the `login` element and corresponding buttons:

```
<t:form type="login">
  <t:login type="select"/>
  <t:submit /> <t:cancel />
</t:form>
```

The login dialog can be a separated page, but it can also be included in an template, hiding the content of a page if no user is logged in:

```
<t:if expr="Infolayer.getCurrentUser().oclIsUnknown()">
  <t:form type="login">
    <t:login type="select"/>
    <t:submit  followup="stay"/> <t:cancel>
  </t:form>
<t:else/>
  <t:inner/>
</t:if>
```

For the user, this option may be more convenient than a message saying to log in on a separate page before the content becomes visible. This option makes sense especially for content that shall not even be visible without login. In the example, the submit followup command was set to `stay` because the default behavior would return to the previous page, instead of showing the actual content of the page, that was hidden by the login form.

### 7.3.6 Tables

While it is possible to generate HTML tables based on the iterate element, the `<t:table>` element permits to interactively select a column determining the sort order of the table.

For tables, the current type of `self` must be a collection type.

The `table` element must contain a number of `<t:column>` elements, defining the columns of the table. The `column` element supports the attributes `title` and `expr`, defining the title and content of each column. For the `expr` attribute, `self` iterates over the element of the collection for the current row.

The result of the expression can be formatted using the `format` attribute. Alternatively, if the `column` element is not empty, the whole content is processed for each table cell. The evaluation context for the subtree is the result of the `expr` attribute.

## 7.4 Other Content Formats

Besides XHTML and the general XML support, the Infolayer provides special support for some other content formats, namely the the Wireless Markup Language (WML), the Portable Document Format (PDF), and non-XML files. The actual handling of a file depends on the extension; files with an unrecognized extension will not be processed further, but directly delivered to the client.

### 7.4.1 Non-XML Formats

For non-XML formats, the XML escaping rules for writing reserved characters such as '<' and '&' are switched off—otherwise, it would be impossible to generate those characters. The only extensions for non-XML formats that is recognized by default is ".csv".

### 7.4.2 Portable Document Format (PDF)

The XSL standard contains an XML based page description language that can be used to generate documents in the Adobe Portable Document Format (PDF) [2]. For this purpose, the Infolayer system supports the Apache XSL Formatting Object Processor (FOP) [5], if it is installed in the Servlet environment.

When the Infolayer receives a request with the Extension ".pdf", it looks for a corresponding template with the extension ".fo". The template is processed, and the resulting XML file is handed over to FOP, which generates an PDF document that is transmitted to the client.

Template elements in the source files should use the namespace `http://infolayer.org/templates/fo`, although it currently does not provide additional template elements.

### 7.4.3 Resource Description Format (RDF)

Template elements used for RDF generation should use the namespace `http://infolayer.org/templates/rdf`, although there is currently no RDF spe-

cific support. Files with the extension ".rdf" are recognized as RDF templates.

### 7.4.4 Wireless Markup Language (WML)

WML templates must have the file extension ".wml". They reside in the separate directory `wml`. The namespace for WML template elements is `http://infolayer.org/templates/wml`. For WML, a specialized version of `<t:valueOf>` is supported, as well as read-only versions of `<t:property>` and `<t:properties>`.

## 7.5 Servlet Request Handling

In some cases, it may be necessary to interact with the Infolayer Servlet from a client application directly, or to recreate the functionality of predefined Infolayer template elements, for example when an image button shall be used instead of the predefined `submit` button, or when functionality that was not foreseen in the existing templates is required.

In addition to the standard URL format described in section 7.1.2, the Infolayer supports several special purpose addresses, starting with "get-" or "post-", depending on the HTTP method.

The general URL syntax for the additional get and post requests is:

- `GET /get-`*type*`.`*ext*`?`*par1*`=`*value1*`&`*par2*`=`*value2*`&...&`*parN*`=`*valueN*
- `POST /post-`*type*`.`*ext*

In the case of post requests, the parameters are transferred with the request body and are not visible in the browser URL field. The "extension" is used to indicate the content type.

While the value of the `self` parameter must always be a valid OCL literal, the other parameter values may be plain strings not enclosed in quotes.

POST requests are used when the request changes the internal state of the system, since the HTTP specification requires that GET requests are free of side effects. If possible, the browser is forwarded to a valid "GET" URL when the post request was processed.

### 7.5.1 Navigation

When processing forms, it is not possible to assign different URLs to different buttons. The following general request parameters allow to make further processing dependent on the button pressed:

`cancel-`***id***: Cancel the current form. The form is not processed and the *id* is used to look up a corresponding `followup` command.

`followup-`***id***: Use the value as `followup`-command, if the submit or cancel button with the given id was pressed.

`submit-`***id***: Commit the current form. The form is processed and the id is used to look up a corresponding `followup` command.

`return` Used to store the URL previously visited, used by the `return` followup command.

### 7.5.2 Error Handling

When processing a request, it is possible that an error occurs. In that case, an `err`-parameter is added to the URL of the resulting page. Error reports can be displayed in the resulting page using the `<t:messages>` element.

### 7.5.3 Query Requests

The URL base for query requests is "get-query", followed by the content type indicator and query parameters. Supported parameters are:

`self` Denotes the class that is queried.

`q-`***name*** Contains the value the attribute or association end *name* must contain in order to make the instance appear in the result list. If more than one properties are queried, they are combined with a logical *and*. Empty values correspond to empty input fields and are not taken into account.

`template` Denotes the template that shall be used to render the query result. The template is searched in the `collection` template directory. If not present, the value defaults to `results`.

### 7.5.4 Instance Updates

Instance updates can be performed by setting the form post URL to `post-edit`, followed by the content type extension. The instance denoted by the `self` parameter is updated with respect to the following additional request parameters:

`self` Denotes the instance to be updated. The value must contain the instance id, preceded by a dollar sign ($).

`add-`*`name`* Add the value to the object property *name*. Strings are not quoted, but instance IDs are prefixed with a dollar sign. An asterisk followed by a class name means to add a newly created instance.

`remove-`*`name`* Removes the value from the property with the given name.

`set-`*`name`* Sets the value of the object property *name*.

### 7.5.5 Method Invocations

Any operation of the model can be invoked using the `get-exec` and `post-exec` URLs. The GET method and address can be used only for query methods; for methods having side effects, the POST method must be used. The request parameters are as follows:

`self:` Denotes the instance the method belongs to.

`op:` The name of the method, immediately followed by the parameter types in round brackets. The parameter names must be omitted. If the method has no parameter, an empty pair of round brackets must be appended to the method name.

`p`*`n`* The value of the *n*-th parameter, encoded as OCL literal.

`p`*`n`*`$` The value of the *n*-th parameter, encoded as plain string. This option allows to use input fields to generate function parameters.

If the method returns a result value, this value is used as evaluation context of the next page displayed, following the instructions of the *followup* request parameter.

### 7.5.6 User Login and Logout

For logout, it is sufficient to post to the address `/post-logout`. For login, the address is `/post-login` and the additional parameters `login` and `password` are required. If the login and password are valid, the user is logged in using a session cookie.

### 7.5.7 Setting cookies

The address `/post-cookie` can be used to set cookies. The request parameter format is:

`cookie-`***name***: Sets a cookie with the given name to the parameter value.

### 7.5.8 URL manipulation and URL based access restrictions

The Infolayer is able to restrict, rewrite and redirect URLs. For this purpose, the configuration file `httpd.conf` supports the commands `permission`, `rewrite` and `redirect`. All three commands require two parameters containing OCL expressions in double quotes, separated by a colon. The first expression is always a boolean expression denoting URLs the command applies to. In both expression, `self` is the `IlUrl` object for the current URL. In the case of `permission`, the second parameter is a boolean OCL expression determining if the permission to access the given URL is granted. For `redirect` and `rewrite`, the second expression contains the target URL. The difference is that `rewrite` immediately delivers the page corresponding to the new URL, whereas `redirect` asks the browser to send a request for the new address.

The following example limits access to the path "internal" to users who are currently logged in:

```
permission "getPath().startsWith('~/internal')",
           "Infolayer.getCurrentUser()->notEmpty()"
```

## 7.6 Completeness

The template mechanism provides full access to OCL and inherits its computational completeness, demonstrated in section 5.3 of chapter 5. However, this is not sufficient to show that any desired XML output can be generated. The naive approach—to interpret the tape symbols of the presented OCL Turing Machine as Unicode characters—does not work since any textual output of the template mechanism is subject to the XML escaping rules. The escaping mechanism writes symbols with a special meaning in XML—such as "`<`"—in an escaped form ("`<`" in the given case), ensuring that it is not possible to generate invalid XML. The literal symbol "`<`" can only be written by specialized template elements such as `<t:element>` or `<t:comment>`.

Of course it would be possible to work around this problem with a special literal output mode. However, it is relatively simple to show that we do not need an extension to write any XML infoset instance. The XML infoset is a formalization of the contents of an XML file. It abstracts XML syntax details that are not relevant for information processing, but still covers SGML legacy and syntactic sugar, namely DTDs, namespaces, comments, processing instructions and CData sections. To keep the proof readable, we will leave out those parts and use the following simplified definition:

- An *Element* is defined as a 3-tuple (*name*, *attributes*, *children*), where
  - *name* is an string denoting the name of the element
  - *attributes* is a finite set of *Attribute*s
  - *children* is an ordered set of *Element* and *String* objects.
- An *Attribute* is a pair of strings (*name*, *value*), denoting the name and the value of the attribute.

An XML document is simply defined as its root element. The corresponding OCL type definition is:

```
TupleType(
  name: String,
  attributes: Set(TupleType(name: String, value:String)),
  children: Sequence(OclAny))
```

In order to prove that the template mechanism is able to generate any desired XML output, we need to show that we can construct an XML template (named *element*) that properly writes all components, namely

- the element name,
- the attributes, and
- the child elements and text nodes.

**Claim 5** *It is possible to design an XML template that writes the element name.*

**Proof 5** *The following template fragment writes the name of an element in valid XML syntax:*

```
<t:element t:name="name" xmlns:t="http://infolayer.org/templates">
</t:element>
```

**Claim 6** *It is possible to write all attributes of an element with an XML template.*

**Proof 6** *The following template fragment writes all attributes of an element:*

```
<t:forAll expr="attributes">
  <t:attribute t:name="name" t:value="value" />
</t:forAll>
```

**Claim 7** *It is possible to design an XML template that writes the sequence of children, given a template that is able to write an element:*

**Proof 7** *The following template fragment writes all child elements. String content is written immediately; for element content, the template is called recursively.*

```
<t:forAll expr="children">
  <t:choose>
    <t:when expr="oclIsKindOf(String)">
      <t:valueOf expr="self" />
    </t:when>
    <t:otherwise>
      <t:call template="xmlwriter" />
    </t:otherwise>
  </t:choose>
</t:forAll>
```

**Claim 8** *It is possible to design an XML template that writes any XML infoset as defined above.*

**Proof 8** *The XML template can be constructed by combining all previous parts:*

```
<t:element t:name="name" xmlns:t="http://infolayer.org/templates">

  <t:forAll expr="attributes">
    <t:attribute t:name="name" t:value="value" />
  </t:forAll>

  <t:forAll expr="children">
    <t:choose>
      <t:when expr="oclIsKindOf(String)">
        <t:valueOf expr="self" />
      </t:when>
      <t:otherwise>
        <t:call template="xmlwriter" />
      </t:otherwise>
    </t:choose>
  </t:forAll>

</t:element>
```

## 7.7 Summary

In this chapter we have presented an XML generation mechanism for object oriented systems that is based on XSLT, but provides direct access to the sys-

tem state. This way, we have avoided an intermediate XML representation that has an inherent loss of information, resulting from the transformation of the object graph into a tree.

We have extended this template mechanism with server sided elements beyond the capabilities of the XML processing chain, opening possibilities such as building the default system state editor based on a set of default templates. In addition to the possibility to edit the content remotely without specialized tools, this also leads to a single consistent interface for both, browsing and editing the content.

We have proven that our template mechanism is able to generate any valid XML output, going one step beyond Kepser's proof of the computational completeness of XSLT [67] by not simply relying on literal output, but using the adequate XML generation elements.

# Chapter 8

# Persistent Storage

For a system like the Infolayer, it is a necessity to save the stored information persistently. Even if the content will fit into the main memory of modern computers in many cases, it is not desirable to lose it after a system restart.

For storing information persistently, there are two main options: the file system and a database system.

Database systems, relational, object-oriented, or mixed, provide several advantages over simple files, such as better scalability and sophisticated support for queries and transactions. However, while a database system needs to be installed separately, the availability of a file system can be assumed on most platforms.

Thus, by default, the content of the Infolayer system is stored in the file system, requiring no additional setup steps. However, to utilize the advantages of database systems or simply to connect to legacy systems (requirement V), it is also possible to connect the Infolayer system to one or more relational database tables. For this purpose, the Infolayer provides a mechanism to map the class diagram that drives the system to tables in a relational database system.

## 8.1 XML File Based Default Persistence Mechanism

The File System based default storage mechanism of the Infolayer consists of three main building blocks:

- The XML file *instances.xml*, representing a snapshot of the content of the system at a certain point of time.
- The XML file *instances.chg*, containing all changes of the object model relative to the *instances.xml* file.
- A set of binary files in the subdirectory *files*, storing the contents of binary fields that are used, for example, in *File* objects.

The default location of all files is the directory *instances* contained in the main directory of the corresponding web application.

Using an XML based file format is a straight forward solution for storing the structured content of the Infolayer system. The XML file consists of a root element <*data*>, containing one element named after the class name for each stored instance. The instance id is stored in an *id* attribute. Inside the instance element, there are sub elements for each attribute and association end, named after the corresponding property. Datatype values are stored directly as literal content of the corresponding element, while object references are stored using a *idref* attribute, containing the id of the related object. For cardinalities greater than one, the property elements are simply repeated. Properties may contain additional XML attributes *user* and *timestamp* denoting the user name and time of a content modification in ISO format, but those attributes are not essential for the system and may be omitted.

The *instances.chg* file inherits the structure from the *instances.xml* file, but it is interpreted as "difference" to the *instances.xml* file. In this file, all changes of the object model are journalled immediately. All properties of instances are added to the properties described in the *instances.xml* file, except for properties with a maximum cardinality of one. In that case, the property value is simply replaced. In the difference file, it is possible to

delete instances using the *deleted* attribute with a value of *true*. Associations can be annulled by setting the attribute of *related* to a value of *false* for the corresponding property and instance(s).

Attributes with binary content are stored in separate files, in order to avoid cluttering of the instance files with large amounts of data. The path to a binary attribute is constructed as follows:

```
<classname>/<instance-id>/<property-name>.bin
```

Please note that binary attributes must not have a cardinality greater than one. If that is desired, it is necessary to use an intermediate class such as *File*.

While it would have been possible to use XMI to externalize instances, the verbosity of XMI does not make the format very well suited for manipulation with a text editor. While the XML files are generated automatically for a given model, they are not automatically adopted to changes in the model that are not backwards compatible, such as renaming attributes. Thus, for those kinds of model changes it may become necessary to edit the XML files manually if corresponding instances are existing and shall be preserved.

## 8.2 Relational Database Connections

When compared to a "regular" relational database, the UML-Classes contained in the model correspond to tables, and the instances correspond to rows, and the columns correspond to UML-attributes. A class in the Infolayer system can be linked to a database table by setting the tagged value *il-connection*. The syntax of the connection string is:

```
table:<table-type>:<connection-details>
```

Supported table connection types are:

**jdbc:** Connects a class to a SQL table using the Java Database Connectivity (JDBC). The connection details consist of the jdbc database URL, the user name, the password, and the table name in the following form:

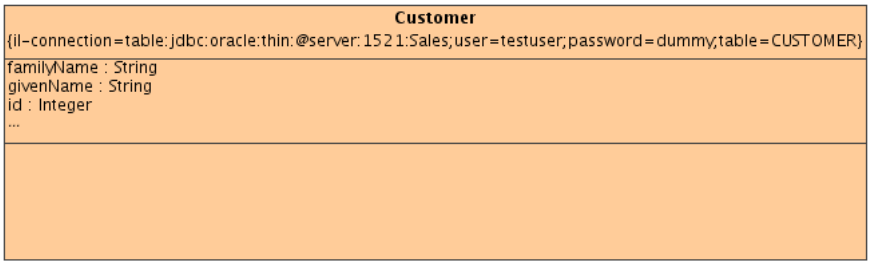


Figure 8.1: Example Database Table Connection

```
<jdbc-url>;user=<username>;password=<password>;
    table=<table>;driver=<classname>
```

Of course, the <*jdbc-url*>, <*username*>, <*password*>, <*table*>, and <*driver*> parameters must be replaced with actual values.

**bibtex:** Connects the associated class to a literature database in the BibTex format. The BibTex file name must be provided following the colon as connection details.

**dbase:** Connects the associated class to a DBase III+ database (.dbf) file. The file name must be provided as connection details following the colon.

**arff:** Connects the associated class to an Attribute-Relation File Format, developed by the Machine Learning Project at the Department of Computer Science of The University of Waikato for use with the Weka machine learning software [6]. Basically, the ARFF format is a list of comma separated values with a header section defining the column types. The file name of the ARFF file must be provided as connection details following the colon.

Figure 8.1 shows a sample specification of a database connection.

### 8.2.1 Mapping Columns and Attributes

For all table based formats, it is required that one column can be used as a unique identifier, serving as the primary key to access a row. If the table

| SQL | DBase | ARFF | DL | Infolayer Datatype |
|---|---|---|---|---|
| BIT / BOOLEAN | L (logical) | | N | Boolean |
| BLOB | | Y | Binary | |
| CHAR | C (character) | | N | String |
| DATE | D (date) | date | N | DateTime |
| DOUBLE PRECISION | | numeric | N | Real |
| FLOAT | | | N | Real |
| LONGVARCHAR | M (memo) | | Y | String |
| NUMERIC | N (numeric) | | N | Real (scale>0) |
| | | | | Integer (scale=0) |
| REAL | | | N | Real |
| TIME | | | N | DateTime |
| TIMESTAMP | | | N | DateTime |
| VARCHAR | | string | N | String |

Table 8.1: Mapping from SQL Types to Infolayer Types; The DL Column Shows the Default Value for "il-deferred-loading"

does not contain a column named `id`, or the column named `id` does not contain a unique identifier for each row, a corresponding column must be specified using the tagged value `il-id-field`.

If the database table contains columns that do not correspond to attributes in the model, those are generated at runtime. The opposite direction is not implemented: when defining attributes in the model that do not correspond to existing columns in the table, it is necessary to define at least a "getter" method; missing columns in the table are not created automatically.

Table 8.1 shows how SQL and DBase column types are mapped to the built in data types of the Infolayer system. Character column types can optionally be mapped to enumeration types by explicitly defining an attribute matching the column name but having the desired enumeration type. In this case, the column content must match one of the enumeration literals.

### 8.2.2 Deferred Loading

It is possible to mark fields for deferred loading by setting the tagged value *il-deferred* to true. Usually, all attribute values are loaded into memory when an instance is accessed. Deferred attributes are loaded only when the

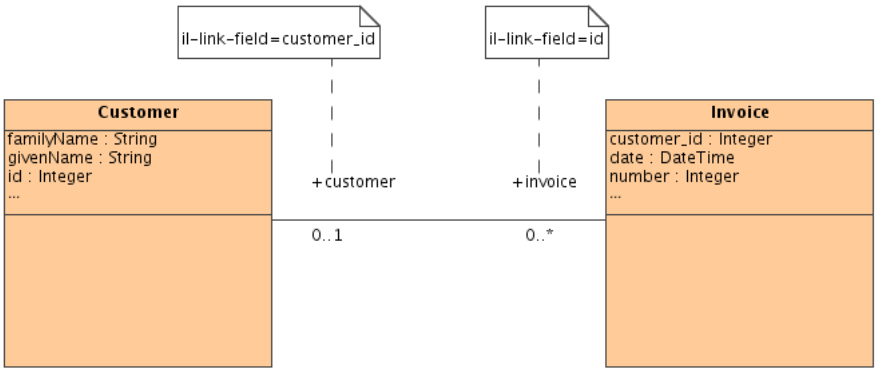


Figure 8.2: Tagged Values Providing Database Implementation Details of $1:n$ Associations.

attribute itself is accessed. The default values of the *il-deferred* property depend on the column type and are also shown in table 8.1.

## 8.3 Mapping Associations

While associations are a first class concept in UML, they are usually represented by key matches across different tables in the relational world. Thus, some kind of mapping is required. Here, UML tagged values again provide means to annotate the model with information about a specific implementation.

### 8.3.1 1:*n* Associations

The simplest case are 1:$n$ associations. For instance, consider a customer linked to a set of invoices, like depicted in figure 8.2. In the example, an invoice belongs to a customer if the customer *id* column matches the *customer_id* column of an invoice.

The tagged value named *il-link-field* makes this information available to the Infolayer system. The column name *id* is specified in a tagged value named `il-link-field` of the *invoice* association end. For the other end of the association, the tagged value is set to *customer_id*.

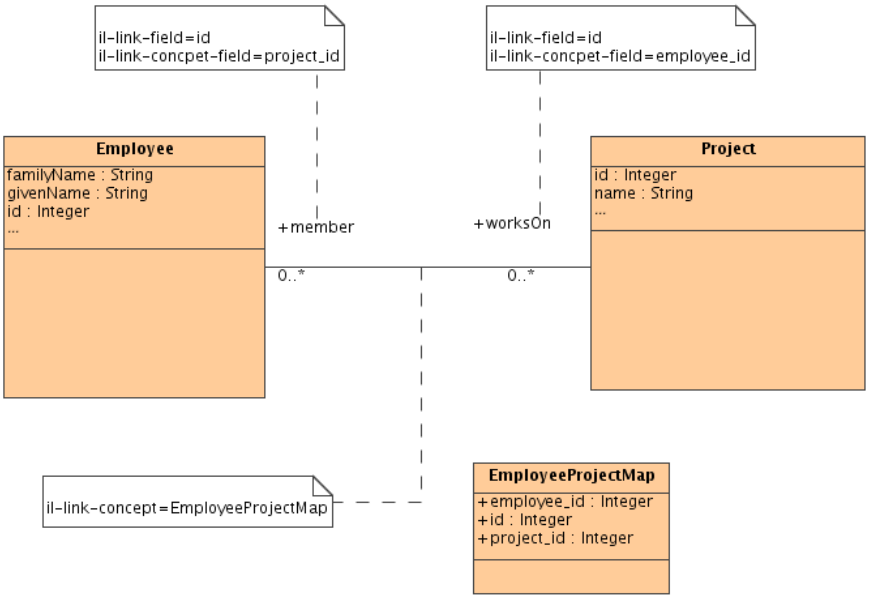


Figure 8.3: An n:m Association Including an Association Table and the Required Annotations

Internally, the *il-link-field* value is used to generate the following OCL getter expression for the corresponding association end:

```
let lf = <link-field> in
<type>.allInstances()->select(lf=<opposite-link-field>)
```

At runtime, <*link-field*> is replaced with the *il-link-field* value, <*type*> with the class name of the opposite association end, and <*oppositeLinkField*> with the *il-link-field* value of the opposite end.

A "setter" is only generated for the association end with the maximum cardinality of one. The expression is:

```
<link-field> := value.<opposite-link-field>
```

### 8.3.2 *n*:*m* Associations

While 1:n associations can simply be modeled in relational databases following the approach described above, for *n*:*m* associations an explicit as-

sociation table is required. This additional table links arbitrary numbers of associated rows by storing pairs of row identifiers from both tables. An example for n:m associations is the assignment of employees to projects: on the one hand, a single employee can participate in different projects. On the other hand, one project may have more than one employees assigned.

To model this association, the association table `employee_project_map` is used. Each row of this table links a pair of ids from both associated tables.

Again, for the Infolayer system the association must be annotated with implementation details: Both association ends must be annotated with the tagged values *il-link-field* and *il-link-concept-field*, denoting the key fields of the linked classes and their counterparts in the association table. Additionally, the association itself must be annotated with the tagged value *il-link-concept*, denoting the class making the association table accessible for the Infolayer system. Of course, also the *il-connection* value must be set properly for all three classes.

Figure 8.3 shows an annotation sample for the sketched employee-project link.

Also the mapping of the tagged value entries for the *n*:*m* case to "getters", "adders", and "removers" is a bit more complex than for the 1:*n* case:

- getter-definition:

```
let lf = <link-field>
let lcs = <il-link-concept>.allInstances()
  ->select(<link-concept-field> = lf)
  ->collect(<opposite-link-concept-field>) in

<type>.allInstances()->select
  (lcs->includes(<opposite-link-field>))
```

- adder definition:

```
let n = <link-concept>.createInstance() in
n.<link-concept-field := <link-field>;
n.<opposite-link-concept-field> :=
  value.<opposite-link-field>
```

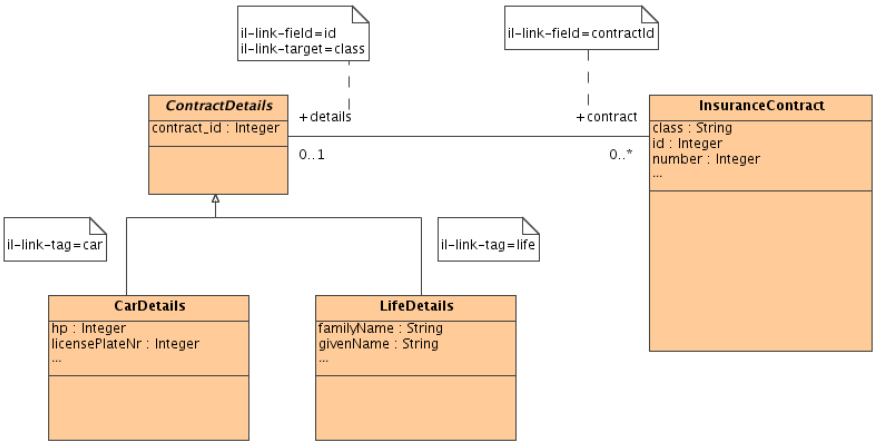


Figure 8.4: A Dynamic Association Between the Employee Class on the one Hand and the Classes BaseAttribute and Value on the other Hand.

- remover definition:

```
let lf = <link-concept-field>
let olf = value.<opposite-link-field> in
<link-concept>.allInstances()
 ->select (<link-concept-field> = lf
           and <opposite-link-concept-field> = olf)
 !delete();
```

### 8.3.3 Linking Different Tables Dynamically

A special case of an association is an association where the table at one association end is not fixed, but denoted by a column of the table at the other association end.

In order to model this special case in the information layer, all classes corresponding to dynamically associated tables must be subclasses of a common superclass. The association is not drawn to all target classes, but only to the common superclass. The tagged value `il-link-target` must contain the field that denotes the table name. If the content of this field does not contain the name of the target class directly, the target classes must be marked with a corresponding `il-link-tag` tagged value. Figure 8.4 shows a corresponding example taken from the UML model used to represent the table

structure of an insurance company. The *il-link-target* tag is only supported for 1:$n$ associations.

The generated getters and setters are:

- getter:

```
let lf = <link-field>
let lt = <linkTarget>  in

<type>.allSubtypes()->one
   (if not getTaggedValue('il-link-tag').isOclUnknown()
    then getTaggedValue('il-link-tag')
    else name
    endif = lt).allInstances()
  ->select(<opposite-link-field> = lf)
```

- setter:

```
let vt = value.oclType
let tag = vt.getTaggedValue('il-link-tag') in
<link-field> := value.<opposite-link-field>;
<linkTarget> := if not tag.isOclUnknown() then tag
                else value.oclType() endif
```

## 8.4 Mapping OCL Expressions to SQL

A simple way to apply OCL expressions to the contents of database tables is to translate the rows to the corresponding Infolayer object representation, making them accessible to the regular Infolayer OCL processor. However, the set of instances stored in database tables may be very large. Even if enough main memory is available, all instances would need to be transferred from the database to the infolayer system, before an OCL expression could be applied. Ideally, the OCL expression would be evaluated inside the database system, avoiding any unnecessary I/O costs. However, relational database systems usually support only SQL queries, but not OCL.

Some OCL constructs can be translated to SQL, though. For instance, attributes can be translated to column names, and most operations on simple

data types such as <, =, or *and* have matching SQL counterparts. The OCL construct *<Class>.allInstances()->select(<condition>)* can be translated to a corresponding SQL select statement if *<Class>* is linked to a database table and the condition can be translated to an SQL *where*-clause. However, this is not possible for the complete language. For instance, it is not possible to dirctly translate queries involving data from different sources. A general translation cannot be possible since OCL is computational complete and SQL is not.

However, even partial translations may be helpful. For an OCL *select()*-operation, a SQL where clause that is more general than the OCL source may still reduce the set of instances that need to be considered by subsequent stages. Thus, the full OCL select condition needs only to be applied to the reduced set of instances that passes the SQL filter. If an OCL select expression cannot be converted completely, the system tries to determine a more general SQL expression for a partial mapping, for instance by omitting the left or right side of a boolean "and".

In addition to this partial translation scheme, the Infolayer system uses delayed evaluation and referential transparency of OCL expressions to optimize generated SQL queries.

### 8.4.1 Partial Translations

A simple example for an OCL expression where a partial translation to SQL makes sense is:

```
Person.allInstances()->select(
   p|p.name = 'Beaker' and p.calcIncome() > 2000)
```

The name query may be translated to SQL easily, but the income calculation operation not. However, finding a more general SQL query that delivers a super set of the desired instances will already avoid a lot of IO and mapping operations:

```
select * from Person where name = 'Beaker' and true
```

| Generalization Parameter | Allowed SQL results for OCL result true | false |
|---|---|---|
| MAY_GENERALIZE | true | true or false |
| MAY_SPECIALIZE | true or false | false |
| EXACT_MATCH | true | false |

Table 8.2: OCL to SQL translation scheme

| OCL Expression | SQLTranslation |
|---|---|
| toSql(expr1 and expr2, X) | toSql(expr1, X) and toSql(expr2, X) |
| toSql(expr1 or expr2, X) | toSql(expr1, X) or toSql(expr2, X) |
| toSql(expr, MAY_GENERALIZE) | not toSql(expr, MAY_SPECIALIZE) |
| toSql(expr, MAY_SPECIALIZE) | not toSql(expr, MAY_GENERALIZE) |
| toSql(expr, EXACT_MATCH) | not toSql(expr, EXACT_MATCH) |
| toSql(NOT_TRANSLATABLE, MAY_GENERALIZE) | true |
| toSql(NOT_TRANSLATABLE, MAY_SPECIALIZE) | true |
| toSql(NOT_TRANSLATABLE, EXACT_MATCH) | NOT_TRANSLATABLE |

Table 8.3: OCL to SQL translation scheme

In order to perform this kind of SQL conversions, the class OclExpression contains an Infolayer-specific operation *toSql*, translating a given OCL expression to SQL. To take advantage of partial translations, the *toSql()* operation is parameterized whether the translation result may be more general (MAY_GENERALIZE), more special (MAY_SPECIALIZE), or an exact match with the OCL expression is required (EXACT_MATCH).

If a boolean sub-expression cannot be translated at all, it is replaced with *true* in the MAY_GENERALIZE case, and false for MAY_SPECIALIZE, making it possible to construct a valid (more general or more special) SQL expression, even if parts of an expression cannot be translated. The allowed deviations from the result of a boolean OCL expression, depending on the generalization parameter, are illustrated in table 8.2. The generalization parameter is also recognized by the translation operation for several boolean operations. For instance, the generalization mode for the argument of a *not* operation needs to be inverted. The mode-specific treatment of boolean operations is illustrated in table 8.3.

For all remaining OCL operations, the translations are either exact or not available, as documented in appendix E.

### 8.4.2 Pre-Calculation of Constant Values

All operations available in the Infolayer system are marked with meta-data determining whether an operation is referentially transparent, making it possible to pre-calculate values. The exclusion of side effects is not sufficient here since functions returning a random number or the current time are not generating a side effect but still cannot be pre-calculated.

The identification and elimination of referentially transparent sub-expressions that do not depend on the iterator is feasible in OCL since OCL is free of side effects, so it can be assumed safely that there are no hidden dependencies.

To support this kind of query optimization, the Infolayer implementation of the *OclExpression* class from the OCL Model was extended with two additional methods:

**boolean isResolvable(int var)** Determines whether the expression can be made dependent only on the given (iterator) variable.

**OclExpression resolve(Object[] bindings, int var)** Replaces all variables with the given bindings, except from the iterator variable. Now constant sub expressions are reduced to constants automatically.

### 8.4.3 Deferred Evaluation

With the above methods, it is possible to optimize several simple data type expressions. Unfortunately, filter chains applied to collections are often not given as a single expression. For instance, the input for the *table* XML template element is a collection. This collection may be filtered further or sorted when the table is actually displayed. One option would be to supply an expression instead of a collection for the table element. The alternative is to temporally allow intensionally defined collections for some special cases such as *allInstances()* and *select()*. The materialization of the collection is delayed, similar to the delayed evaluation of expressions in the functional programming language Haskell [49], making it possible to refine the underlying SQL expression before the materialization. In the Infolayer system, the following operations can defer the materialization of the resulting collection:

**allInstances():** Materialization is always deferred for classes connected to Database tables. If *allInstances()* returns instances from more than one class, the corresponding *union()* operation is also deferred.

**union():** The union operation may be delayed for optimizations based on the expressions defining the underlying collections.

**select():** If the underlying collection is not yet materialized, a deferred collection, based on the expression defining the original collection and the filter expression, is returned.

## 8.5 Summary

In this chapter, we have illustrated how the Infolayer system persists its state. In the simplest case, the existing objects are serialized to a human-readable XML file in the file system. This is sufficient for many application

scenarios. For those cases where increased reliability is needed or legacy date is required, the Infolayer can be connected to relational databases, addressing requirement V from the introduction. In addition to the technical integration, we have demonstrated how the generation of SQL statements can be optimized, for instance by using deferred evaluation techniques known from functional programming languages. The SQL optimization is not limited to the Infolayer system but could also be used in other OCL-based systems that need to generate SQL.

# Chapter 9

# Applications and Third Party Additions

The Infolayer system has been in development and use for several years now. The department sample presented in chapter 1 is actually a trimmed down version of the web presentation of our own unit, where we use the Infolayer system to manage the the public information about research and software projects, unit members, and about 3000 publications. The very first Infolayer application was a conference system accessible by humans and software agents in the scope of the CoMRIS EU project [98, 50, 51, 52]. This chapter describes some additional applications of the Infolayer system, and some extensions of the overall system that have been implemented for those applications.

## 9.1 The MuSofT Project

MuSofT is the acronym for a Germany-wide project that develops multimedia teaching material for software engineering education [32]. The goal of the web presence of the project, the so-called MuSofT-Portal, is to manage and distribute the learning objects contributed by the various project partners.

For the realization of the portal, existing content management systems (CMS) did not meet the requirements. Traditionally, CMS distinguish a


Figure 9.1: Screenshot of the MuSofT Application

developer and a presentation view, but are not providing the retrieval capabilities required for the designated work flow in the MuSofT project. Moreover, the set of meta attributes and value sets may change corresponding to the experiences with the portal, thus a simple adaptation option for meta attributes was required. Finally, for a user friendly interface, simple navigation options along the hierarchy of learning objects and the classification schema were desired. Thus, the MuSofT portal was implemented on top of the Infolayer system [4, 3].

The MuSofT-Portal is designed for archiving learning objects, which are created inside as well as outside of the MuSofT project. The most important activities that can be performed with the portal are uploading, modifying and querying learning objects. When inserting new objects, the meta data requirements must be taken into account, ensuring a consistent description of the objects. Here, the Infolayer user interface supports the user by providing input fields for the required attributes and select boxes for attributes with a fixed value set.

In addition to the upload process, the portal supports the hierarchical classification of learning objects according to the LOM standard [62]. Thus, a

new learning unit can easily be built as an arrangement of modules existing in the portal. A screen shot of the user interface, showing the topics available and the numbers of corresponding entries, is depicted in figure 9.1.

A first working version of the MuSofT portal was developed during the Winter of 2001 in only two weeks. It included all required features, though much of it was still in its infancy. After approval by the MuSofT project members, the system was successfully improved during 2002. The domain model was refined and templates were added to the system, based on the layout provided by a web designer. Most of the template work was done by a single student worker who had prior experience with HTML and some insight into UML and OCL. Over time, more features such as e-mail notifications and download counters were added. While still in development, the MuSofT application was already used by the project members to upload their learning material, so it was possible to incorporate feedback early.

Since the official public launch of the application in fall 2004, about 100 different learning objects on software engineering along with their meta data have been stored on the server. The learning objects range in complexity and size from simple slides over specialized applications to videos of several hundred megabytes. The class diagram that drives the MuSofT portal is depicted in figure 9.1.

## 9.2 SOAP Interface

To simplify the integration of existing structured resources, the MuSofT project partners asked for an XML based bulk upload option. To suit this need, a SOAP based upload mechanism was added to the Infolayer system. The upload mechanism consists of a server sided component that is able to generate an XML Schema corresponding to the UML diagram, and a component that is able integrate data conforming to the schema into the current object diagram.

A separate graphical client (figure 9.2) that is able to perform a client side validation prior to the upload can be used to select conforming XML files.

Both, client- and server sided components are not specific to the MuSofT model, but can be used in any Infolayer project if desired. The SOAP

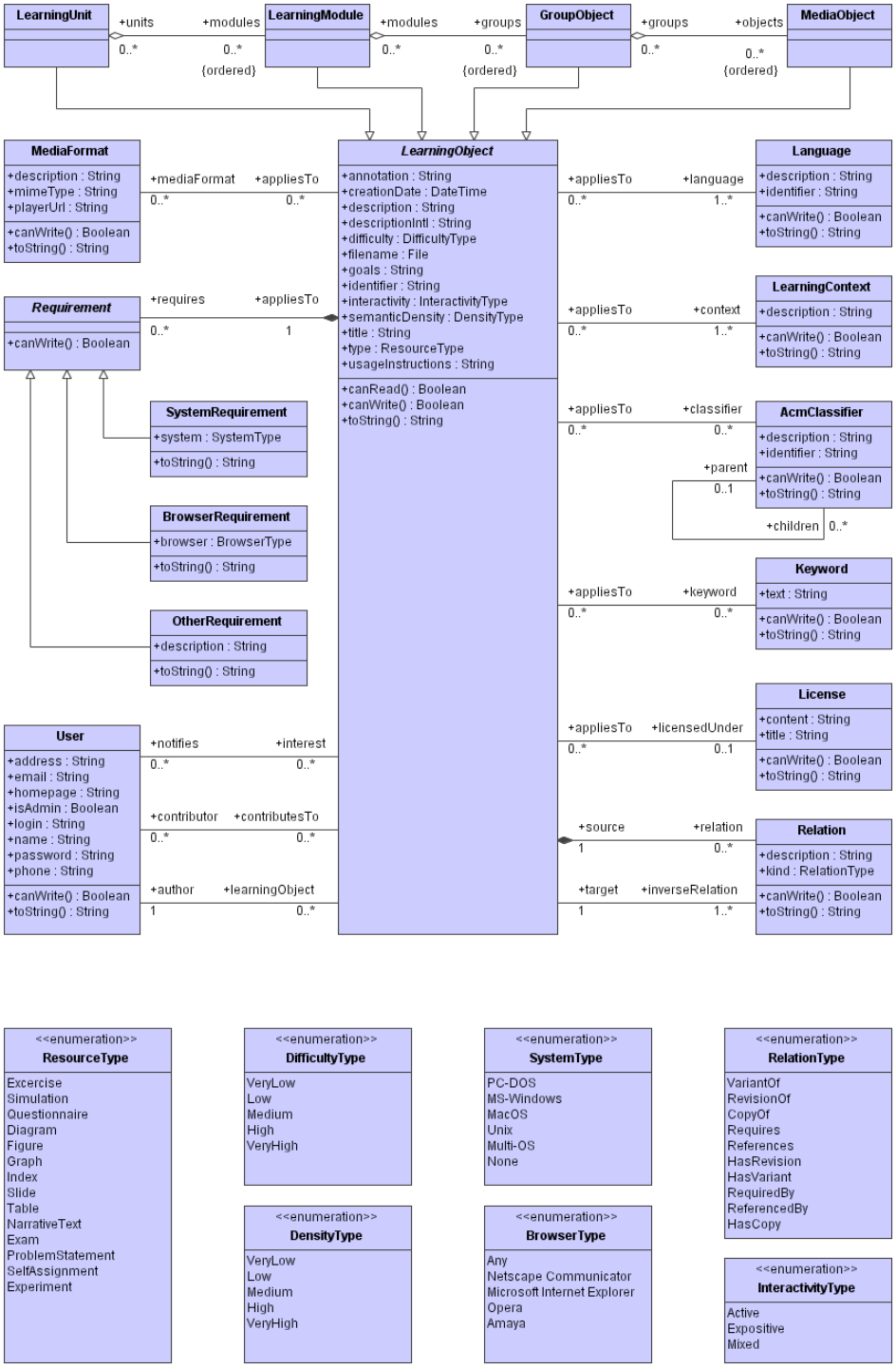


Figure 9.2: Class Diagram of the MuSofT Application

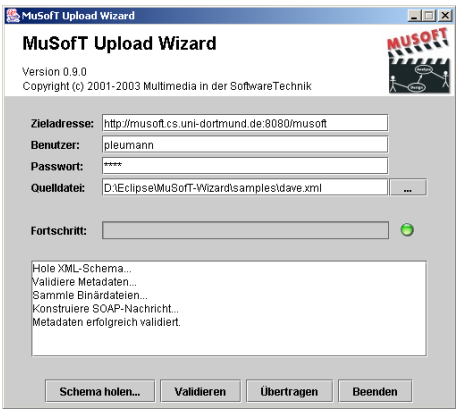


Figure 9.3: SOAP Upload Client Developed in the MuSofT Project

interface was later extended to enable the invocation of any methods defined in the model.

## 9.3 State Machines

Another addition to the system that was performed mainly by Jörg Pleumann in the scope of the MuSofT project is an interpreter for UML state chart diagrams that was originally implemented as the simulation component of a CASE tool [99]. State charts are used to model the life cycle of objects. A special HTML template element makes it possible to display the state of an object as a state chart diagram where the current state is highlighted. State transitions can be triggered via the user interfaces using HTML buttons.

## 9.4 MLnet and KDnet

The Information Layer system is also used for the training information server of the MLnet project and it successor, the KDnet project. Both projects are similar to the MuSofT portal, but for training information from the areas of Machine Learning and Data Mining.


Figure 9.4: Infolayer Anatomy Application

## 9.5 Medical Application

The medical computer science unit of the University of Applied Sciences Dortmund and the University of Münster both use the Infolayer system for information access with mobile clients [70, 108]. In the Münster Project, images of dissected human corpses are taken in order to assist students during their studies in anatomy. The Infolayer is used as middleware providing SOAP based access to the image repository. Both, the mobile clients and the HTML interface can be used for chat communication. The off-campus teaching capabilities in anatomy were extended, since students at home have direct access to their fellow students in the dissecting room.

Here, the Infolayer was chosen because the SOAP interface provided convenient access to the whole functionality from mobile clients. The mobile clients provide a simplified user interface, specialized for the tasks to be performed remotely. The HTML interface (figure 9.5) can still be used from regular terminals to browse and refine the content, and for administrative purposes. In the project, also some server sided tools for handling DICOM

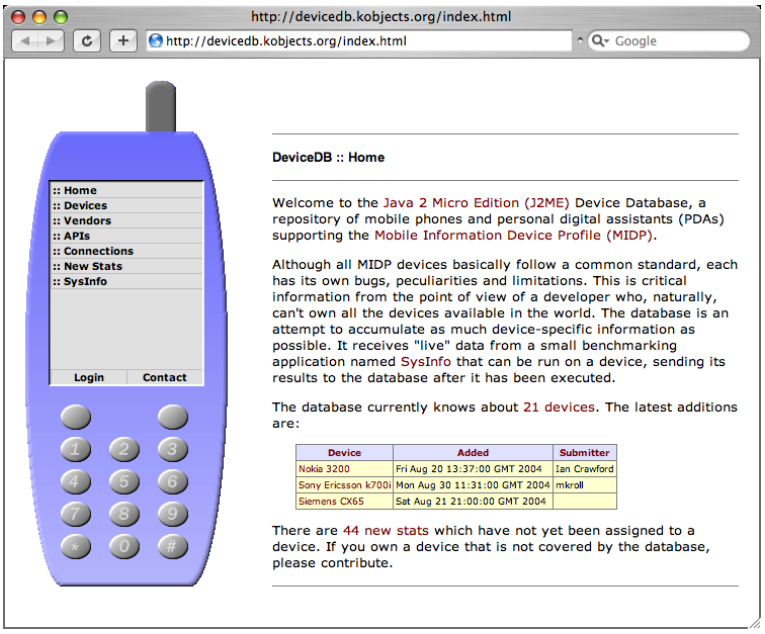


Figure 9.5: Screenshot of the DeviceDB Application

[31] images were added to the Infolayer Servlet. The project consists of 10 classes and about 100 objects.

## 9.6 DeviceDB

Another project that utilizes the Information Layer is a database for Java-enabled small devices like cell phones and personal digital assistants[1]. Here, the model describes the devices, their capabilities such as available protocols, and vendors.

The basic profile of a Java-enabled mobile device is obtained automatically by a mobile client software that can be downloaded from the DeviceDB web pages. The device information is directly uploaded using the SOAP interface. Figure 9.6 shows the title page of the DeviceDB application, indicating that currently 21 different device types are stored in the data base, and 44 new records are waiting for approval.

[1] http://devicedb.kobjects.org

## 9.7 Summary

Some of the above applications have been in use for about three years. The general approach of modeling relevant structural and dynamic parts of a Web application in a CASE tool and then executing this model works well. For the applications mentioned in the previous sections, there was practically no additional Java programming necessary (only the MuSofT application required the two additional classes to handle e-mail notifications and the export functionality). With the model itself becoming executable, we were able to produce a working prototype for any of the applications very early. If a database schema proved to be insufficient for the application, it was possible to go back to the CASE tool, change the model, and then execute it again. No implementation effort was spent on thrown-away prototypes, which makes the Infolayer ideal for a rapid application development (RAD) approach in the Web application or database context. We think a similar approach should be possible in other areas, too. Prototypes directly generated from an initial class diagram can gradually be turned into a working system by adding classes and templates. Where the amount of data became too large for storage in plain XML files, the database connectivity ensured that the system is able to grow with increasing demands.

# Chapter 10

# Conclusion and Outlook

In this chapter, we first summarize our work, following the XML toolchain analogy, and then reconsider how this translates back to our original goal, the improvement of web application development. Finally, we provide an outlook for further research or development based on this work.

## 10.1 Summary

The increasing size and complexity of web applications has led to a situation where the traditional approach of creating and managing a set of plain HTML files is inappropriate in many cases. Consistency in structure, look and feel, and hyperlinks needs to be maintained, and support for different content formats may be required. The combination of XML Schema, XML and XSLT is able to improve this situation, but the expressive power of XML Schema is insufficient for application domains where more than a pure hierarchical structure is required.

In this work, we have chosen the XML toolchain as a guideline to construct an alternative basis for web information systems at a higher level of abstraction, namely UML class diagrams. We have identified a UML counterpart or implemented a substitute for each constituent of the XML processing chain, showing that it is possible to build a consistent UML-based system for model driven web applications.

The XML Schema has been replaced by UML class diagrams, and the XML editor that is used for editing the schema can be replaced by any UML

tool that is able to generate the XMI model interchange format. We have developed a default transformation from a set of objects and classes to a set of HTML pages that can not only be used for browsing, but also for editing the contents of the system. Hence, no second editor is needed; compared to the XML toolchain, the functionality just has been shifted to a different place in the system.

In order to create a replacement for XSLT, a template mechanism based on XSLT elements, but using OCL as an expression language instead of XPath, has been developed. XSLT could have been used unchanged by serializing the system state into an XML tree. However, by forcing the system state into a tree structure, many of the advantages of a higher level of abstraction would have been lost. Moreover, adding template elements such as input fields supporting interaction with the servlet made it possible to use the template mechanism to take over the content editing functionality, leading to a single coherent interface for content inspection and modification. Compared to the XML toolchain, using a web interface for editing the content provides the advantage that content can be maintained from anywhere without needing specialized tools. However, the web interface also raised the demand of a more sophisticated user management, which is also addressed by our system.

In many areas, the capabilities of the system go far beyond the XML tool chain. Operations in UML class diagrams are fully supported. While query operations can be specified in OCL, for non-query operations a new surface language for the UML action semantics has been developed. Compared to existing action languages, our language has the advantage that it integrates OCL expressions and provides a consistent syntax, instead of duplicating OCL functionality with a new syntax. The support for operations makes it possible to trigger actions that modify the system state via the user interface—without leaving the UML framework.

Another feature that goes beyond the scope of the XML toolchain is the clearly specified connection to relational database systems, including a translation mechanism that is able to generate SQL queries from OCL expressions. By deferring the evaluation of expressions such as *allInstances()* and generating over-generalized expressions where an exact match is not possible, we are able to achieve a high level of efficiency in the generated queries.

Many aspects of this work such as the HTML generation rules, templates, database connectivity or the action semantics surface language are not tied to the Infolayer system, they could easily be used for other applications. For instance, the action semantics surface language could be used in *YATL* (Yet another transformation language) [96] to increase its consistency with the remainder of the UML. YATL is a transformation language that has been defined to perform transformations within the OMG's MDA framework. YATL currently uses OCL and its own imperative features (not based on the UML action semantics) for unidirectional model transformations.

## 10.2 Conclusion

Leaving the XML analogy and looking back at our original goal, improving web development methodologies, we can conclude that the Infolayer system demonstrates that direct model interpretation is a viable alternative to traditional implementation approaches as well as the upcoming Model Driven Architecture.

Especially for data-intensive applications that do not require much specialized business-logic, a prototype can be generated by simply drawing a UML class diagram—a step that is required in the relevant development methodologies anyway. By making this first step immediately operational, the gap between web development methodologies and actual system implementation has been narrowed significantly.

Modeling the link structure and the page layout itself are covered at a technical level by the Infolayer template mechanism. Built on XML, the template language fits well with XML target languages (XHTML, WML, XSL-FO, ...), preserving the strength of XSLT and much of its structure, while reaching a higher level of abstraction by providing direct access to the user model via OCL expressions.

Of course there is still a gap between additional navigation diagrams suggested by web engineering methodologies and the Infolayer template mechanism. However, by using OCL in the templates—instead of languages that are one or more steps further away from the UML—this gap has been narrowed significantly. Moreover, a large part of page design consists of artistic aspects that are not covered by any of the web engineering approaches. In

our approach, this part—the design of the XML templates—is clearly separated from the model design, without being limited to cosmetic changes only. Building the template expressions mostly on UML instead of programming language artifacts may also help non-technicians to better see the links to the design documents. Allowing designers to stick with their favorite tools may improve the interaction with software engineers, which is often a problematic area in web engineering [78].

Cutting the number of involved technologies for structured web applications down to a small coherent set required some amount of new "glue" to connect them. Fortunately, this could be limited to the replacement of XPath by OCL in XSLT, and the creation of a simple OCL based surface language for the UML Action Semantics.

Experiences with developing and using the applications presented in the previous chapter have been very positive so far, and we feel that it should be feasible to apply the same ideas to other application domains, too.

## 10.3 Outlook

The direct interpretation of UML diagrams creates new opportunities beyond the current scope of the Infolayer system. For instance, it seems interesting how support for additional UML diagram types could fit into the system. While state charts are already supported to control the life cycle of an object, they may also help to create dialog structures for the user interface. Moreover, the interpretation approach could be explored for other types of applications than web applications.

### 10.3.1 System Extensions

For the Infolayer system itself, there are three concrete improvements that we would like to perform:

- Support for refactoring

  The model interpretation approach puts the option to perform model changes (refactoring [94]) at runtime within reach. To make this hap-

pen, at least write access to the meta model would be required, although an integrated graphical modeling tool would be more user friendly.

- Replacement of the model implementation with the Eclipse UML 2 framework

  As noted in chapter 3, using standard components could reduce the probability of errors in the system. The Eclipse UML 2 implementation[1] seems to be well suited as a solid foundation for our system, allowing us to drop large parts of our own model representation and the XMI parser. This is especially interesting since the Eclipse framework includes XMI write capabilities, a functionality that is needed to add refactoring capabilities.

- Web services based on Representational State Transfer

  The Infolayer supports web services via SOAP, but recently there has been a lot of discussion whether web services should be based on Representational State Transfer (REST)—the general architecture of the World Wide Web [37]—instead. The SOAP interface works very well, but SOAP does not allow to use the URL to address individual data objects as service endpoints, in contrast to the use of URLs in the remainder of the system. Large parts of SOAP duplicate functionality that is handled by HTTP, while still building on HTTP to bypass firewalls.

### 10.3.2 Correctness

It does not seem practical to provide a formal proof that the Infolayer implementation is actually fully consistent with the UML semantics. However, there are several options to reduce the probability of errors that could be exploited in the future:

- *Use standard components:* When using "standard" components (e.g. the Dresden OCL toolkit [61, 38]) that are used by many developers, the probability that errors are not noticed decreases.

[1] http://eclipse.org/uml2

- *Use the OCL definition of an operation where available:* If the definition of the result of an operation is available in OCL, it can be used as implementation. However, there may be a performance penalty.
- *Use test cases to find errors:* Conformance tests seem to be the most practical option, used for instance in the Java Community Process. While they do not provide a proof of correctness, they are able to detect many errors, especially simple copy-paste issues and common problems such as special case handling if well constructed.

In the Infolayer system, it may make sense to replace parts of the OCL framework by "standard"-components. However, the existing tools are mainly addressing Java code generation, whereas we are interested in direct interpretation. Of course running conformance tests would be a desirable option, but currently the availability of tests is limited. We were able to collect some general tests from the precise UML mailing list, which actually helped to find and eliminate several bugs in the Infolayer OCL implementation. However, the tests are limited in scope. The MuSoft project is using JUnit [21] test to test some aspects of the HTML code generated by the Infolayer, but not for testing OCL and model correctness in general. The availability of a more sophisticated "official" test suite from the OMG would probably help to improve the quality and interoperability of UML tools significantly.

The source code of Infolayer system is available at http://infolayer.org.

# Bibliography


[1] Action Semantics for the UML, August 2001. OMG ad/2001-08-04.

[2] Adobe Systems, Inc. *PDF Reference: Version 1.4 (3rd Edition)*. Addison-Wesley, 2001.

[3] Klaus Alfert, Ernst-Erich Doberkat, and Gregor Engels. MuSofT Bericht Nr.2: Ergebnisbericht des Jahres 2002 des Projektes MuSofT – "Multimedia in der SoftwareTechnik". Technical Report 133, University of Dortmund, Computer Science X, March 2003.

[4] Klaus Alfert, Ernst-Erich Doberkat, Gregor Engels, Marc Lohmann, Johannes Magenheim, and Andy Schürr. MuSofT: Multimedia in der SoftwareTechnik. In Johannes Siedersleben and Debora Weber-Wulff, editors, *Software Engineerung im Unterricht der Hochschulen Berlin 2003 (SEUH 8)*, pages 70 – 80. dPunkt-Verlag, Heidelberg, February 2003.

[5] Apache Software Foundation. Formatting Object Processor (FOP). http://xml.apache.org/fop/.

[6] Attribute-relation file format (arff). http://www.cs.waikato.ac.nz/~ml/weka/arff.html.

[7] K. Baclawski, M. Kokar, P. Kogut, L. Hart, J. Smith, W. Holmes, J. Letkowski, and M. Aronson. Extending UML to support ontology engineering for the Semantic Web. In C. Kobryn M. Gogolla, editor, *In Fourth International Conference on The Unified Modeling Language*, number 2185 in LNCS, pages 342–360. Springer-Verlag, October 2001.

[8] H. Baumeister, N. Koch, and L. Mandel. Towards a UML extension for hypermedia design. In *Proceedings of UML'99*, 1999.

[9] Tim Berners-Lee and Daniel Connolly. Hypertext Markup Language (HTML), June 1993. Internet Draft (Working Document).

[10] Boldsoft et al. Response to the UML 2.0 OCL RfP (ad/2000-09-03), January 2003. Revised Submission, Version 1.6. OMG Document ad/2003-01-17.

[11] Alex Borgida. On the relative expressiveness of Description Logics and Predicate Logics. *Artificial Intelligence*, pages 353–367, 1996.

[12] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, April 1985.

[13] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a formalization of the Unified Modeling Language, 1997.

[14] Rob Brooks-Bilson. *Programming ColdFusion*. O'Reilly, 2001.

[15] Jean-Michel Bruel. Transforming UML models to formal specifications. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[16] M. Buchheit, F. Donini, and A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.

[17] R. G. G. Cattell and Douglas K. Barry. *The Object Data Standard ODMG 3.0*. Morgan Kaufmann, 2000.

[18] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web Modeling Language (WebML): A modeling language for designing Web Sites. *Computer Networks*, 33(1–6):137–157, 2000.

[19] Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James Rice. OKBC: A programmatic foundation for knowledge base interoperability. In *AAAI/IAAI*, pages 600–607, 1998.

[20] P.P Chen. The entity/relationship model: toward a unified view of data. *ACM Transaction on Database Systems*, pages 9–36, 1976.

[21] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. Technical Report 01–12, Iowa State University, 2001.

[22] Jim Conallen. Modeling web application architectures with UML. *Communications of the ACM*, 42(10):63–70, 1999.

[23] The World Wide Web Consortium. XSL Transformations (XSLT) version 1.0, 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[24] The World Wide Web Consortium. SOAP Version 1.2 Part 1: Messaging Framework, 2002. http://www.w3.org/TR/2002/CR-soap12-part1-20021219/.

[25] The World Wide Web Consortium. SOAP Version 1.2 Part 2: Adjuncts, 2002. http://www.w3.org/TR/2002/CR-soap12-part2-20021219/.

[26] Microsoft Corporation. ASP.NET Web: The official Microsoft ASP.NET site. http://www.asp.net/.

[27] Aaron Crane. Experiences of using PHP in large websites. In *Linux 2002*, 2002.

[28] S. Cranefield and M. Purvis. UML as an ontology modelling language. In *Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-23/cranefield-ijcai99-iii.pdf.

[29] Stephen Cranefield, Stefan Haustein, and Martin Purvis. UML-based ontology modelling for software agents. In *Proceedings of the Autonomous Agents 2001 Workshop on Ontologies in Agent Systems*, May 2001.

[30] D.H.Akehurst and B.Bordbar. On querying UML data models with OCL. In *UML 2001 Modeling Languages, Concepts and Tools*, October 2001.

[31] Digital imaging and communication in medicine (DICOM), 2004.

[32] Ernst-Erich Doberkat and Gregor Engels. Musoft – multimedia in der softwaretechnik. *Informatik Forschung und Entwicklung*, 17(1):41–44, 2002. Springer Verlag.

[33] Michael Eichberg. MDA and programming languages. In *OOPSLA 2002 Workshop Generative Techniques in the context of Model Driven Architecture*, 2002.

[34] H. Eriksson, R. W. Fergerson, Y. Shahar, and M. A. Musen. Automatic generation of ontology editors. In *Twelfth Banff Knowledge Acquisition for Knowledge-based systems Workshop*, Banff, Alberta, Canada, 1999.

[35] Andy Evans and Tony Clark. Foundations of the Unified Modeling Language. In *Proc. of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, 23-24 September 1997*, 1997.

[36] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, 1999. http://www.ietf.org/rfc/rfc2616.txt.

[37] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[38] Frank Finger. Design and implementation of a modular OCL compiler. Master's thesis, TU Dresden, March 2000.

[39] S. Flake and W. Müller. OclType - a type or metatype? In *Workshop OCL 2.0 – Industry standard or scientific playground? at UML 2003*, Electronic Notes in Theoretical Computer Science, San Francisco, CA, USA, October 2003. Elsevier, Amsterdam, The Netherlands.

[40] WAP Forum. Wireless Markup Language (WML), April 1998. http://www.wapforum.org/.

[41] D. Frankel. *Model Driven Architecture – Applying MDA to Enterprise Computing.* OMG Press, 2003.

[42] F. Garzotto and P. Paolini. HDM — a model-based approach to hypertext application design. *ACM Transactions on Information Systems*, 11(1):1–26, Jan 1993.

[43] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[44] W. E. Grosso, H. Eriksson, R. Fergerson, J. Gennari, S. Tu, and M. Musen. Knowledge modeling at the millennium – the design and evolution of protege-2000, October 1999.

[45] Thomas R. Gruber. A translation approach to portabele ontology specifications. In *Knowledge Aquisition*, 1993.

[46] Volker Haarslev and Ralf Möller. Racer: A core inference engine for the semantic web. In *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003), located at the 2nd International Semantic Web Conference ISWC 2003*, pages 27–36, Sanibel Island, Florida, USA, October 2003.

[47] Frank Halasz and Mayer Schwartz. The dexter hypertext reference model. In *Hypertext Workshop*, NIST Special Publication, pages 95–133, Gaithersburg, Md, January 1990. National Institute of Standards and Technology.

[48] Siegfried Handschuh, Steffen Staab, and Alexander Maedche. Cream — creating relational metadata with a component-based ontology-driven annotation framework. In *First International Conference on Knowledge Capture*, 2001.

[49] Report on the programming language Haskell version 1.1, 1991.

[50] Stefan Haustein. Information environments for software agents. In Wolfram Burgard, Thomas Christaller, and Armin B. Cremers, editors, *KI-99: Advances in Artificial Intelligence*, volume 1701 of *LNAI*, pages 295–298. Springer Verlag, September 1999.

[51] Stefan Haustein. Utilising an ontology based repository to connect Web miners and application agents. In *Proceedings of the ECML/PKDD Workshop on Semantic Web Mining*, September 2001.

[52] Stefan Haustein and Jörg Pleumann. Easing participation in the Semantic Web. In Martin Frank, Natasha Noy, and Steffen Staab, editors, *WWW2002 International Workshop on the Semantic Web*, volume 55 of *CEUR Workshop Proceedings*, 2002.

[53] Stefan Haustein and Jörg Pleumann. Is Participation in the Semantic Web too Difficult? In Ian Horrocks and James Hendler, editors, *First International Semantic Web Conference*, volume 2342 of *LNCS*, pages 448–453. Springer, 2002.

[54] Stefan Haustein and Jörg Pleumann. OCL as expression language in an action semantics surface language. In Octavian Patrascoiu, editor, *OCL and Model Driven Engineering*, pages 99–113. University of Kent, 2004.

[55] Stefan Haustein and Jörg Pleumann. A model-driven runtime environment for Web applications. *Software and Systems Modeling*, 2005.

[56] P. J. Hayes. In defence of logic. In *Proc. of the 5th IJCAI*, pages 559–565, Cambridge, MA, 1977.

[57] Reiko Heckel and Marc Lohmann. Model-based development of web applications using graphical reaction rules. In M. Pezz, editor, *Fundamental Approaches to Software Engineering*, pages 170–183. Springer, 2003.

[58] Brian Henderson-Sellers and Franck Barbier. Black and white diamonds. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30*, volume 1723 of *LNCS*, pages 550–565. Springer, 1999.

[59] Rolf Hennicker and Nora Koch. Systematic design of web applications with uml. In *Unified Modeling Language: Systems Analysis, Design and Development Issues*, pages 1–20. Idea Group Publishing, 2001.

[60] I. Horrocks. Using an expressive description logic: Fact or fiction? In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pages 636–647, San Francisco, California, June 1998. Morgan Kaufmann Publishers.

[61] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language.*

*Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000.

[62] IEEE Learning Technology Standards Committee. Final draft of the IEEE standard for Learning Objects and metadata. http://ltsc.ieee.org/wg12, 2002.

[63] Tomás Isakowitz, Edward A. Stohr, and P. Balasubramanian. RMM: A methodology for structured hypermedia design. *Communications of the ACM*, 38(8):34–44, 1995.

[64] Kabira Technologies, Inc. Kabira Action Semantics. http://www.kabira.com.

[65] Tomihisa Kamada. Compact HTML for Small Information Appliances (cHTML). W3C Note, February 1998. http://www.w3.org/TR/1998/NOTE-compactHTML-19980209.

[66] Kennedy Carter Ltd. Action Specification Language (ASL). http://www.kc.com.

[67] Stephan Kepser. A proof of the turing-completeness of XSLT and Xquery. Technical report, University of Tuebingen, May 2002. SFB 441.

[68] Anneke Kleppe and Jos Warmer. Extending OCL to include actions. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000.

[69] Andreas Kraus and Nora Koch. Generation of web applications from UML models using an XML publishing framework. In *Integrated Design and Process Technology, IDPT-2002*, 2002.

[70] Michael Kroll. Designing embedded Java software for the visualization and analysis of DICOM waveform objects on resource constrained computer platforms. Master's thesis, University of Applied Sciences, Dortmund, 2004.

[71] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) model and syntax specification. Technical report, World Wide Web Consortium, 1999.
http://www.w3.org/TR/1999/REC-RDF-SYNTAX-19990222.

[72] Håkon Wium Lie and Bert Bos. Cascading style sheets, level 1, January 1999. W3C Recommendation 17 Dec 1996, revised 11 Jan 1999.

[73] Marc Lohmann, Stefan Sauer, and Tim Schattkowsky. ProGUM-Web: Tool-support for model-bases development of web applications. In *UML 2003*, number 2863 in LNCS, pages 101–105. Springer, 2003.

[74] Bertram Ludäscher, Ilkay Altintas, and Amarnath Gupta. Time to leave the trees: From syntactic to conceptual querying of xml. In *Intl. Workshop on XML Data Management (XMLDM), in conjunction with Intl. Conf. on Extending Database Technology (EDBT)*, Prague, March 2002.

[75] Luis Mandel and María Victoria Cengarle. On the expressive power of OCL. In *FM'99 - Formal Methods. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Proceedings, Volume I*, volume 1708 of *LNCS*, pages 854–874. Springer, 1999.

[76] Murali Mani, Dongwon Lee, and Richard R. Muntz. Semantic data modeling using XML schemas. In *20th International Conference on Conceptual Modeling (ER 2001)*, 2001.

[77] Bruce Martin and Bashar Jano. WAP Binary XML Content Format (WBXML). W3C Note, June 1999. http://www.w3.org/TR/wbxml.

[78] A. McDonald and R. Welland. Web engineering in practice. In *Proceedings of the Fourth Workshop on Web Engineering*, pages 21–30, May 2001.

[79] Santiago Meliá, Cristina Cachero, and Jaime Gómez. Using MDA in web software architectures. In *2nd International Workshop on Generative Techniques in the Context of MDA*, Anaheim, California. USA, October 2003.

[80] Stephen J. Mellor and Marc Balcer. *Executable UML – A Foundation for Model-Driven Architecture*. Addison Wesley Longman, 2002.

[81] Stephen J. Mellor, Steve Tockey, Rodolphe Arthaud, and Philippe LeBlanc. Software-platform-independent, precise action specifications for UML. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 281–286, 1998.

[82] Sun Microsystems. JavaServer Pages – Dynamically Generated Web Content. http://java.sun.com/products/jsp/, 2003.

[83] Marvin Minsky. A framework for representing knowledge. Memo 306, MIT-AI Laboratory, June 1974.

[84] Rolf Möller and Volker Haarslev. Description logics for the semantic web: Racer as a basis for building agent systems. *Künstliche Intelligenz*, pages 10 – 15, March 2003.

[85] Peter D. Mosses. Theory and practice of action semantics. Technical Report BRICS RS-96-53, Department of Computer Science, University of Aarhus, Ny Munkegade, building 540, DK-8000 Aarhus C, Denmark, 1996.

[86] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2000.

[87] Peter Naur and J. Backus. Report on the algorithmic language ALGOL 60. *j-CACM*, 3(5):299–314, May 1960.

[88] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*. Number 422 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1990.

[89] Natalya Fidman Noy, Ray W. Fergerson, and Mark A. Musen. The knowledge model of Protégé-2000: Combining interoperability and flexibility, 2000.

[90] Object Management Group (OMG). Homepage. http://www.omg.org.

[91] Object Management Group (OMG). Model Driven Architecture (MDA). http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, 2001.

[92] Object Management Group (OMG). XML Metadata Interchange (XMI) Specification 1.2, 2002. http://www.omg.org/cgi-bin/doc?formal/2002-01-01.

[93] Object Management Group (OMG). Unified Modeling Language (UML) 1.5 Specification. http://www.omg.org/cgi-bin/doc?formal/03-03-01, 2003.

[94] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[95] Ese Ozkarahan. *Database Machines and Database Management*. Prentice Hall, 1986.

[96] Octavian Patrascoiu and Peter Rodgers. Embedding OCL expressions in YATL. In Octavian Patrascoiu, editor, *OCL and Model Driven Engineering*, pages 60–68. University of Kent, 2004.

[97] PHP: Hypertext Preprocessor - homepage, 2001. http://www.php.net.

[98] E. Plaza, J. Arcos, P. Noriega, and C. Sierra. Competing agents in agent-mediated institutions. *Personal Technologies Journal*, 2(3):1–9, 1998.

[99] Jörg Pleumann. Erfahrungen mit dem multimedialen didaktischen Modellierungswerkzeug DAVE. In *2. Deutsche e-Learning Fachtagung (DeLFI)*. Gesellschaft für Informatik, September 2004.

[100] Jörg Pleumann and Stefan Haustein. A model-driven runtime environment for Web applications. In *UML Conference 2003*, LNCS. Springer Verlag, 2003.

[101] Project Technology, Inc. BridgePoint Action Language (AL). http://www.projtech.com.

[102] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[103] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. The Architecture of a UML Virtual Machine. In *2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)*, pages 327–341. ACM Press, 2001.

[104] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International Editions, New York, NY, 1991.

[105] Bernhard Rumpe. A note on semantics (with an emphasis on UML). In *Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, 1998.

[106] Tim Schattkoswki and Marc Lohmann. Rapid Development of Modular Dynamic Web Sites Using UML. In J.M.-Jezequel, H. Hussmann, and S. Cook, editors, *UML 2002*, volume 2460 of *LNCS*, pages 336–350. Springer, 2002.

[107] U. Schöning. *Theoretische Informatik – kurzgefasst*. Spektrum Akademischer Verlag, 3 edition, 1999.

[108] B. Schütze, M. Kroll, H.-G. Lipinski, and T.J. Filler. Mobiles computerbasiertes Lernen in der anatomischen Lehre. In H. Höpfner and G. Saake, editors, *Workshop Grundlagen und Anwendungen mobiler Informationstechnologie des GI-Arbeitskreises Mobile Datenbanken und Informationssysteme*, pages 104–113, 2004.

[109] Daniel Schwabe, Gustavo Rossi, and Simone D. J. Barbosa. Systematic Hypermedia Application Design with OOHDM. In *UK Conference on Hypertext*, pages 116–128, 1996.

[110] Mika Siikarla, Jari Peltonen, and Petri Selonen. Combining ocl and programming languages for uml model processing. In *Workshop OCL 2.0 – Industry standard or scientific playground? at UML 2003*, Electronic Notes in Theoretical Computer Science, San Francisco, CA, USA, October 2003. Elsevier, Amsterdam, The Netherlands.

[111] Richard Soley et al. Model Driven Architecture. Object Management Group White Paper, November 2000.

[112] J. M. Spivey. *The Z Reference Manual*. Prentice Hall, 1992.

[113] P. Spyns, D. Oberle, R. Volz, J. Zheng, M. Jarrar, Y. Sure, R. Studer, and R. Meersman. Ontoweb – a semantic web community portal. In *PAKM*, 2002.

[114] York Sure. SWRC - Semantic Web Research Community Ontology, 2001. http://ontobroker.semanticweb.org/ontos/swrc.html.

[115] Thomas Kudrass Tobias Krumbein. Rule-based generation of XML schemas from UML class diagrams. In *Berliner XML Tage 2003*, pages 213–227, 2003.

[116] Jeffrey D. Ullman. *Principles of Database Systems*. Computer Science Press, 1982.

[117] International Telecommunication Union. Specification and description language (SDL). Technical Report Z.100, ITU-T, 1999.

[118] World Wide Web Consortium (W3C). The semantic web. http://www.w3.org/2001/sw/.

[119] Dave Winer. XML RPC specification, June 1999. http://www.xmlrpc.com/spec.

[120] World Wide Web Consortium (W3C). Homepage. http://www.w3.org.

[121] World Wide Web Consortium (W3C). XML Path Language (XPath) Version 1.0, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[122] World Wide Web Consortium (W3C). Extensible Markup Language (XML) Specification 1.0 (Second Edition), 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[123] World Wide Web Consortium (W3C). XML Schema part 1: Structures, 2001. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/.

[124] World Wide Web Consortium (W3C). OWL Web Ontology Language semantics and abstract syntax, March 2004. http://www.w3.org/TR/2004/REC-owl-semantics-20040210/.

[125] XOpus. The friendly XML editor. http://xopus.com/.

The URLs in the bibliography have been last visited on 2005-09-01.

## Joint Publications

In joint publications, the author's contributions were 30 % for [29], 70 % for [52], 70 % for [53], and 80 % for [54]. To [100], Jörg Pleumann contributed the integration of state machines and the SOAP interface; to the extended version of the paper for the SoSyM journal [55], he additionally contributed the case study.

# Appendix A

# Installation and Configuration

This appendix contains an overview of all configuration files and the internal file system structure. After the installation, it is recommended to change the password of the default user immediately via the web interface.

## A.1 Configuration File Overview

Table A.1 shows the typical file system structure of an infolayer servlet, relative to the servlet root directory. Files marked with an *R* are searched in the *ressources* directory of the Infolayer JAR file if missing in the servlet directory. For files marked with an *C*, the contents of both locations are combined.

## A.2 Model File Location

The location of the XMI file containing the model is denoted by the model parameter in the *web.xml* file:

```
<init-param>
  <param-name>model</param-name>
  <param-value>model.xml</param-value>
</init-param>
```

For testing the Infolayer, mainly UML Magic Draw (versions 5.5 and 7.0) has been used.

| File | | Purpose |
|---|---|---|
| WEB-INF/web.xml | | General information about the Servlet: model location (app. A.2) and telnet port (app A.3) |
| WEB-INF/lib/il-servlet.jar | | JAR File containing the Infolayer Software |
| model.xml | | Default XMI Model file; name and location may differ, depending on *web.xml*. |
| instances/instances.xml | | Instance data XML file (chapter 8). |
| instances/instances.chg | | Instance data updates relative to *instances.xml* |
| html/ | R | HTML and possibly other template files, depending on *httpd.conf* (chapter 7). |
| conf/http.conf | C | Allows to define URL manupulation and restriction rules (sec. 7.5.8). Also defines connections between file extensions, template loaders and template directories (sec. D.4). |
| conf/mime.types | C | Mapping from MIME types to file name extensions. |
| i18n.dat | C | Internationalization information (sec. A.5). |

Table A.1: Infolayer File Structure

## A.3 Telnet Interface

The command line interface of the Infolayer described in chapter 3 is accesible with a telnet client via a configurable TCP port. It can be accessed by any user with administrative rights. A default port number is determined by the name of the installation directory and is displayed on the "administration" page. The port number can be set to a fixed value using the *shellport* init parameter in the *web.xml* file, e.g.:

```
<init-param>
  <param-name>shellport</param-name>
  <param-value>2001</param-value>
</init-param>
```

The command line interface can be disabled by setting the *shellport* parameter to -1.

## A.4 Administrative Users

In order to determine whether a user has administrative rights, the Infolayer uses the boolean property *admin* of the user class. If the user class does not contain an *admin* property, the user with the login "admin" is the only user with administrative rights. Table 4.2 shows an overview of the properties of the user class that are interpreted by the Infolayer system.

If the system does not contain any instance of the user class, a user with the login name and password "admin" is created automatically. However, this is not possible if the user class does not contain the login property. In that case, it is necessary to create an initial user manually in the persistent storage system that is holding the actual user data.

## A.5 Internationalization

It is possible to localize the standard elements of the Infolayer user interface, such as *Create* or *Edit* buttons.

For this purpose, the resource file *i18n.dat* in the Infolayer resource directory contains a set of entries of the form "*element-name[lang-code]=localized-value*", for example:

```
create[de]=Erzeugen
create[en]=Create
```

In order to support further languages, according entries can be added to the *i18n* file.

# Appendix B

# OCL Overview

This appendinx contains an overview of the Object Constraint Language, including additions to the OCL standard library that we needed for our application. The full set of predefined classifiers and operations is described in appendix E.

## B.1 Context and *self*

In OCL, the UML context of an OCL expression within an UML model can be specified using a *context* declaration. In the Infolayer system, the expression must always be attached to the contextual model element, which is easily done in most case tools. If a context declaration is given, it must match the attached element.

The reserved word *self* is used to refer to the contextual instance. For instance, if the context is *Person*, then *self* refers to an instance of *Person*.

## B.2 Constraints

While OCL can be used to specify invariants and pre- and postconditions, only invariants that are attached to classes are actually verified at runtime in the Infolayer system.

Invariants must be OCL expressions of type Boolean and must be true for all instances of the attached class at any time. Constraint violations are clearly highlighted in the Infolayer user interface.

The type of the contextual instance of an OCL expression, which is part of an invariant, is written with the *context* keyword, followed by the name of the type. The label *inv:* declares the constraint to be an "invariant" constraint.

For example, the following constraint requires that all persons with the role *Assistant* must have at least one publication, where *self* is an instance of type *Person*:

```
context Person inv:
    self.role = role::Assistant
        implies self.publication->notEmpty()
```

Here, the keyword *self* can be dropped because the context is clear:

```
context Person inv:
    role = Role::Assistant implies project.notEmpty()
```

## B.3 Types and Type Conformance

In addition to the types defined by the user in the UML model, several predefined types and operations are available in OCL expressions. Most predefined types were already introduced in the previous sections: the Infolayer system supports the basic OCL data types *Boolean*, *Integer*, *String*, *Enumeration* and *Real*, as well as the abstract base type *OclAny*. Additionally, the primitive data types *Binary* and *DateTime*, which are not part of the OCL specification, are available. In contrast to objects, basic types do not have attributes or an identity.

A special literal value that conforms to all types is *OclUndefined*. It is used to denote an undefined value, for instance unassigned attribute values that have no default value. A simple OCL expression that yields *OclUndefined* is *Sequence{1}->any(false)*.

### B.3.1 OclAny

OCL provides a set of general methods that can be applied on any type in OCL. the abstract type *OclAny* is the implicit super type of all types available in OCL. It provides the basic infix operators = and <>, returning a boolean determining whether the compared objects are equal or different. The operation *oclIsKindOf()* determines whether the given object is compatible to a known type, and the operation *oclAsType()* performs a type cast.

Please note that the OCL operation *oclIsNew()* is not supported in the Infolayer system.

The Infolayer provides a set of additional operations for *OclAny*, which are not contained in the OCL specification. The method *toString()* returns a String representation of the object. The comparison operators <, <=, > and >= operate

based on the *toString()* method. Sub types such as *Real* overload those operations with more adequate functionality.

The operations defined for *OclAny* are inherited by all other types.

### B.3.2 String

The String type can be used to store Unicode character strings. It provides the *concat()* method to concatenate strings, *toUpper()* and *toLower()* to convert between upper and lower case. In OCL expressions, String literals must be contained in single quotes. The backslash character is used as escape character and has a special meaning in String literals.

An example for an OCL string expression is:

```
'can you hear me?'.toUpper()
```

### B.3.3 Boolean

The domain of the Boolean data type are the boolean values *true* and *false*. Supported operations are *not* in prefix notation, and *and*, *or*, *xor* and *implies* in infix notation.

A special boolean operation is the functional *if* operation:

```
if b then expr1 else expr2 endif
```

If *b* is true, the result is the value of evaluating *expr1*; otherwise, the result is the value of evaluating *expr2*. *expr1* and *expr2* must have identical types.

### B.3.4 Real

Real values are stored in the Infolayer as double precision 64 bit double precision IEEE 754 floating point numbers [43].

For Real, the mathematical prefix operation − and the infix operations +, −, /, and ∗ are supported. The comparison operations =, <>, <, <=, >, and >= inherited from *OclAny* are overwritten with adequate numeric comparisons. Additionally, the Real data type provides the methods *abs()*, *floor()*, *round()*, *max()*, and *min()*.

An example for an OCL expression of type Real is:

```
38.86 + 3.14
```

### B.3.5 Integer

The domain of the Integer data type are 64-Bit signed integers. Like for Real, the mathematical standard operators −, =, <>, <, <=, >, >=, +, −, /, and ∗ are provided. While +, −, and ∗ return an integer value if both parameters are integers, the /-operator always returns a real value. Additionally, the *Integer* data type provides the methods *abs()*, *div()*, *mod()*, *max()*, and *min()*.

An example for an OCL integer expression is:

```
4 - 4 - 4
```

### B.3.6 DateTime

The data type *DateTime* is not contained in the OCL specification. It can be used to store dates, times, or both.

### B.3.7 Binary

The Binary data type represents a sequence of bytes. It is not contained in the OCL specification. It can be used to store large binary objects such as media files in the Infolayer.

If support for multiple media file formats is required, then the predefined *File* object may be a more suitable option. In addition to the binary field *data*, holding the file content, the class File contains a string field *name* containing the file name.

### B.3.8 Enumeration and Object Literals

In OCL expressions, enumeration literals must be prefixed with their type, separated by a double colon, e.g. *Job::Scientist*.

In addition to the OCL standard, the Infolayer provides access to objects using a dollar sign ($), immediately followed by the instance ID.

### B.3.9 The Classes Object, Infolayer, User, and File

In addition to the primitive types, the Infolayer system defines the predefined classes *Object*, *Infolayer*, *User*, and *File*, as described in the previous chapter.

### B.3.10 Type Conformance

The OCL types are organized in a type hierarchy. This hierarchy determines conformance of the different types to each other. Each type conforms to the transitive closure of its super types.

The operation *oclAsType(OclType)* can be used to re-type (cast) a type to one of its subtypes. The *oclAsType(OclType)* operation results in the same object, but the type is *OclType*.

In some cases, the evaluation result of parts of OCL expressions may be undefined. In this case, the whole expression will be undefined. The only exception to this rule are the boolean operators *and*, *or* and *implies*, which make use of short-circuit evaluation:

- True or anything is True
- False and anything is False
- False implies anything

## B.4 Properties and Operations

In addition to the predefined OCL and Infolayer types, the classes, attributes, associations and operations defined in the UML model can be accessed in OCL expressions.

### B.4.1 Attributes and Association Ends

The values of attributes and association ends can be accessed by a dot, followed by the name. For example, in a Person context, the attribute *givenName* can be accessed using the following expression:

```
self.givenName
```

The type of this expression is the type of the attribute *givenName*, which is the basic type String.

Starting from a given object, it is possible to navigate an association on the class by using the name of the opposite association end. If the association end is unnamed, the type of the opposite end of the association, starting with a lowercase character,

can be used, as specified in the OCL specification. However, this is possible only if the role name is not ambiguous.

For attributes and association ends with upper cardinality limits greater than one, a collection type is returned. For instance, using the following expression, it is possible to navigate from a person to the Set of projects the person is participating in:

```
self.projects
```

When the ordering of the attribute or association end is set to *Ordered*, a Sequence is returned instead of a Set.

Collections like Sets, Bags and Sequences are predefined types in OCL. Using the arrow ('->'), it is possible to access the properties of the collection types. For example, the following expression would return an integer value determining the number of instances for the previous query:

```
self.projects->size()
```

Using the dot and arrow syntax, properties can be combined to more complex expressions, which are evaluated from left to right.

Applying the dot syntax to attributes or association ends with a maximum cardinality greater than one performs an implicit collect operation (section B.7.4). For instance, *self.projects.name* would be expanded to *self.projects-¿collect(name)*, delivering a collection of the names of all projects linked to *self*. Similarly, when using the arrow syntax for properties with a maximum cardinality of one, an implicit conversion to a collection with one or none elements is performed.

Please note that association classes and qualified associations are not supported in the Infolayer system.

### B.4.2 Operations

The results of method invocations can be accessed similar to attributes, except that the method name must be followed by a comma separated argument list, enclosed in parentheses.

For example, the following expression returns a String representation of *self*:

```
self.toString()
```

In OCL expressions, all operations of the model without side effects are accessible.

| Precedence | Operators |
|---|---|
| 1 | @pre (not supported in the Infolayer system) |
| 2 | '.' and '–>' |
| 3 | unary *not* and unary '-' |
| 4 | '*' and '/' |
| 5 | '+' and binary '–' |
| 6 | *if - then - else - endif* |
| 7 | '<', '<=', '>', '>=' |
| 8 | item '=', '<>' |
| 9 | *and*, *or*, and *xor* |
| 10 | *implies* |

Table B.1: OCL Precedence Order for Operators

## B.5 Keywords and Operators

The following identifiers are reserved keywords in OCL, and cannot be used as variable names: *context, def, endif, endpackage, else, if, implies, in, inv, not, let, or, package, post, pre, then, xor.*

In OCL, the operators '*', '/', '+', '–', '<', '>', '<=', '>=', '=', and '<>' are defined as infix operators. Infix operators will be interpreted as method invocations on the first operand with the second operand as parameter. The prefix operators *not* and unary '–' are interpreted as method invocations on their only operand.

Table B.1 shows the OCL precedence order for operators, starting with the highest precedence. Regular parentheses can be used to override the default precedence.

## B.6 Let Expressions

Values and functions that are needed multiple times can be bound to a simple name using a let expression. Let expressions must always be listed at the beginning of an OCL expression, immediately following the context declaration:

```
context Person inv:
let type = Role::Assistant
let magicNumber = 38.86 + 3.14

self.job = type
  implies self.project->notEmpty()
```

In the Infolayer, it is possible to use let expressions to define named values, but functions are not supported. All functions must be defined as regular operations in the model.

## B.7 Collections

Since property values with upper multiplicity restrictions greater than one are returned as collections, collections play an important role in OCL. The type *Collection* itself is the abstract base type of the four supported collection types *Set*, *Bag*, *Sequence* and *OrderedSet*:

- A *Set* in OCL corresponds to a mathematical set. It does not contain duplicate elements. There is no order defined on a set.
- Like for a set, there is no order defined on a *Bag*. In contrast to sets, bags may contain duplicate elements
- Like a bag, an OCL *Sequence* may contain duplicate elements. Additionally, the ordering of the elements in a sequence is preserved.
- A *OrderedSet* does not contain duplicate elements, but in contrast to the *Set*, the order of the elements is preserved.

Collections can be specified in OCL by the type of the collection, followed by a comma separated list of elements, enclosed in curly brackets. Examples for collections are:

```
Set{'Hello', 'World'}
Bag{5, 5, 5, 5403}
Sequence{4, 3, 2, 1, 1, 2, 3, 4}
```

### B.7.1 Collection Type Conformance

OCL collections are typed depending on the contained element type. If *Collection(X)* is a collection of type $X$, and $Y$ is a super type of $X$, then the type *Collection(X)* conforms to the type *Collection(Y)*. For the given base types, *Set(X)* conforms to *Set(Y)*, *Collection(Y)*, and *Collection(X)*.

| Operation | Description |
|---|---|
| any(OclExpression) | Returns any object from the collection where the given boolean expression evaluates to true |
| asBag() | Returns the collection as Bag |
| asSet() | Returns the collection as Set |
| asSequence() | Returns the collection as Sequence |
| at(Integer) | Returns the element with the given index; only available for sequences |
| collect(OclExpression) | Collects the evaluation results of the given expression for each element into a new Bag or Sequence. |
| count(OclAny) | Counts the occurrences of the parameter object in the collection. |
| excludes(OclAny) | Returns true if the parameter is not contained |
| excludesAll(Collection) | True if none of the parameter objects is contained |
| exists(OclExpression) | True if the expression holds for any of the elements |
| forAll(OclExpression) | True if the expression holds for all elements |
| includes(OclAny) | Returns true if the collection contains the given object. |
| includesAll(Collection) | Returns true if the collection includes all objects contained in the parameter. |
| isEmpty() | True if the collection is empty. |
| isUnique(OclExpression) | True if the evaluation results of the given expression are distinct for all elements. |
| iterate(OclExpression) | Iterates over the collection, accumulating values in an accumulator variable, which is returned as result. |
| max() | Returns the maximum of all elements. |
| min() | Returns the minimum of all elements. |
| notEmpty() | True if the collection is not empty. |
| one(OclExpression) | True if the expression holds for exactly one element |
| reject(OclExpression) | Returns a sub-collection, containing only the elements where the given boolean expression evaluates to false. |
| select(OclExpression) | Returns a sub-collection, containing only the elements where the given boolean expression evaluates to true |
| sum() | The sum of all elements. |

Table B.2: Collection Operation Overview

### B.7.2 Collection Operations

OCL defines a large number of predefined collection operations. Consistent with the definition of OCL as an expression language, those operations never change collections. Operations resulting in a collection create a new collection instead of manipulating existing collections. Table B.2 shows an overview of the most important collection operations. A full list is provided in appendix E.

### B.7.3 Iterators and Select

In addition to simple operations like *size()*, *isEmpty()*, and *notEmpty()*, OCL defines several operations on collections that take an OCL expression as parameter. For these operations, the result depends on applying the given expression to all elements of the collection.

For example, the *select* operation results in a filtered collection. If the expression parameter evaluates to true for an element of the source collection, it will be contained in the resulting collection, otherwise not. Thus, the following OCL expression returns all instances of Person where the *role* field is set to "Assistant":

```
Person.allInstances()->select(role = Role::Assistant)
```

In the above example, it is not possible to refer to the persons themselves, only properties of them like the job attribute can be accessed. To refer to the elements of a collections themselves, there is a more general syntax for iterative operations:

```
collection-> <operation> (v | expression with v)
```

The variable v is called iterator. When the operation is evaluated, v iterates over the elements, and expression-with-v is evaluated for each of them. For instance, to select all numbers greater than 10 from a set of numbers, one could write:

```
Set{2, 4, 8, 16, 32}->select(v | v > 10)
```

It is possible to specify the type of the iterator, separated by a colon, like in the example below:

```
Set{1, 3.14, 7, 10, 20, 42}
  ->select(oclIsKindOf(Integer))
  ->select(v : Integer | v mod 10 = 0)
```

### B.7.4 Path expressions and Collect

In contrast to select, *collect* returns a collection containing the evaluation results of the given expression for each element, instead of just filtering the collection. If the original collection is a Sequence, the resulting collection is also a Sequence, otherwise, a Bag is returned.

For example, the following expression returns a bag of strings containing all the names of projects of a given person:

```
person.project->collect(name)
```

To make path expressions more readable, the collect operation is used as default operation in path expressions involving collections. Thus, the following expression is equivalent to the one given above:

```
person.project.name
```

In general, applying a property to a collection of objects will automatically be interpreted as *collect* over the members of the collection with the specified property.

### B.7.5 The Iterate Operation

In contrast to all other collection operations, the iterate operation provides a special accumulator variable, that can be used to accumulate elements of the original collection into a single result. The accumulator variable is declared similar to the iterator, except that it needs an initial value:

```
   collection->iterate(
elem: Type,
acc: Type = init-value
| expression-with-elem-and-acc)
```

The variable *elem* is the iterator variable that is available in all other iterative collection operations as well. The variable *acc* is the accumulator. It gets an initial value of *init-value*.

When iterate is evaluated, *elem* iterates over the collection and the given expression is evaluated for each element. The resulting value is assigned to *acc* after each evaluation. Since *acc* may be used in the iterator expression, a value that depends not just on the current element but also on the previous value of *acc* can be built.

Using iterate, it is possible to provide the functionality of all other predefined collection operations. Some examples are listed below:

**collection->size()** :

```
collection->iterate(i, v = 0 | v + 1)
```

**collection->collect(property)** :

```
collection->iterate(i, v = Bag{} | v->including(i))
```

For sequences, the elements are iterated in the order of the sequence, otherwise, the iteration order is undefined.

## B.8 Tuples

Tuples are sets of typed name–value pairs. Tuple literals are enclosed in curly brackets, and the parts are separated by commas. The type names are optional, and the order of the parts is not relevant. Examples of tuples are:

```
Tuple{givenName = 'John', lastName = 'Smith'}
Tuple{a:String='Foo', b=Sequcence{1..5}}
```

The parts of tuples are accessed by their names, using the same dot notation that is used for accessing attributes. For instance, the expression

```
Tuple{x=5, y=7}.x
```

results in 5.

# Appendix C

# Customization with Tagged Values

The Infolayer provides a powerful template mechanism for customization. However, some basic preferences can also be configured by setting tagged values in the UML model.

Table C.1 shows an overview of all tagged values that are interpreted by the Infolayer, influencing the appearance of the user interface.

Since the mechanism for entering tagged values is inconvenient in some UML tools, tagged values that are interpreted by the Infolayer can also be specified in the documentation of the corresponding element. For that purpose, the documentation must contain a line starting with @<*tag-name*>, followed by a whitespace and the value of the property. Subsequent lines are added to the property value, allowing simple specification of longer entries in tools that provide only small input fields for tagged values (e.g. UML Magic Draw).

## C.1 Labels and Descriptions

For presenting model elements, the Infolayer uses the element name as label and the element description to generate a description of the element where adequate. Those defaults can be overridden by setting the tagged values *il-label* and *il-description*. For both values it is possible to add a suffix that determines the language, separated by a hyphen. This way, different labels and descriptions can be set for different supported languages. Based on the user's browser settings, the appropriate label and description are chosen automatically at runtime. For instance, by using the

| Name | Applicable to | Description |
|---|---|---|
| il-label | classes<br>attributes<br>association ends<br>operations | The given label is shown instead of the class or property name in the user interface |
| il-children | classes | This tagged value can be used to set an association that determines a hierarchy on the instances of this class. This will change the instance select box and the default instance list format to a tree representation. Both association ends must be attached to the same class. |
| il-format | attributes<br>association ends | Determines the format of the attribute or association end, depending on the property type. |
| | classes | Determines the format of the instance list of the given class. |
| il-hide | attributes<br>association ends | Can be used to hide the given property from specific screens of the user interface. Supported values are *view*, *edit* and *query*. |
| il-mimetype | attributes | Allows to set a mimetype for Binary attributes, or to restrict the possible mime types for File attributes. |

Figure C.1: Tagged values for customizing the presentation

tagged value *il-label-de*, it is possible to define a German name for a class. If a description is not available, the regular class and property documentation contained in the XMI file is used as description.

In the default HTML user interface, class descriptions are inserted below the list of instances and property descriptions are attached to the corresponding label as "hover" text.

## C.2 Format String Syntax

The tagged value *il-format* can be used to customize the appearance of property values and instance lists using format strings. Those format strings can not only be used inside *il-format*, but also in the format attribute of several XML template elements such as *<t:valueOf>*. The XML template mechanism is described in detail in the next chapter.

### C.2.1 Number Format Options

The format string for *Real* and *Integer* values is interpreted as a pattern. Internally, the pattern string is used as parameter for the creation of a Java *DecimalFormat* object, which is used for formatting. The following paragraphs summarize the behavior described in the *DecimalFormat* API documentation.

A number format pattern consists of special characters such as "#" or "0" that are replaced by digits or other characters when the format is applied to a number. For example, "#,##0.00" applied to "8765.4321" will result in "8,765.43". Table C.2 shows an overview of the characters that have a special interpretation in number formats. Other characters are taken literally and are unchanged during formatted. In contrast, format characters must be quoted, if they are to appear as literals.

A pattern may contain a positive and negative sub pattern, for example, "#,##0.00;(#,##0.00)". Each sub pattern has a prefix, numeric part, and suffix. The negative sub pattern is optional; if absent, then the positive sub pattern prefixed with the minus sign is used as the negative sub pattern. If there is an explicit negative sub pattern, it serves only to specify the negative prefix and suffix; the number of digits, minimal digits, and other characteristics are all the same as the positive pattern. That means that "#,##0.0#;(#)" produces the same behavior as "#,##0.0#;(#,##0.0#)".

The grouping separator is commonly used for thousands, but may also be used for other grouping sizes. If a pattern contains multiple grouping characters, the interval between the last one and the end of the integer is used.

| Symbol | Description |
|---|---|
| 0 | Digit |
| # | Digit, zero shows as absent |
| . | Decimal separator or monetary decimal separator |
| - | Minus sign |
| , | Grouping separator |
| E | Separates mantissa and exponent in scientific notation. Needs not be quoted in prefix or suffix. |
| ; | Subpattern boundary; Separates positive and negative subpatterns |
| % | Multiply by 100 and show as percentage |
| ' | Used to quote special characters in a prefix or suffix, for example, "'#'#" formats 123 to "#123". To create a single quote itself, use two in a row: "# o''clock". |

Figure C.2: Pattern characters for formatting numbers

Scientific notation is indicated by the exponent character immediately followed by one or more digit characters indicates scientific notation. Example: "0.###E0" formats the number 1234 as "1.234E3".

### C.2.2 String Format Options

For formatting Strings, there are only a few predefined format strings. The actual results of the format options may depend on the user interface:

**memo:** The String is interpreted as a "longer" text, given in unformatted Unicode.

**xhtml:** The String is interpreted as a "longer" text, given in XHTML format.

**url:** The String is interpreted as a URL.

### C.2.3 DateTime Format Options

Similar to the number format string, date formatting is based on the pattern syntax supported by the underlying Java class *SimpleDateFormat*.

Date and time formats are specified by date and time pattern strings. Within date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters representing the components of a date or time string. Text can be quoted using single quotes (') to avoid interpretation. "''" represents a single quote. All other characters are not interpreted; they are simply copied into the output string during formatting.

Table C.3 shows the defined pattern letters.

Pattern letters are usually repeated, as their number determines the exact presentation.

For formatting numeric values, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this amount.

For formatting text values such as the day of the week, if the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used if available.

### C.2.4 Classes and Collections

For formatting class instance lists, collections or properties with a maximum cardinality higher than one, the following format strings can be used:

| Letter | DateTime Component |
|---|---|
| G | Era designator |
| y | Year; if the number of pattern letters is 2, the year is truncated to 2 digits |
| M | Month in year; If the number of pattern letters is 3 or more, the month is interpreted as text; otherwise, it is interpreted as a number. |
| w | Week in year |
| W | Week in month |
| D | Day in year |
| d | Day in month |
| F | Day of week in month |
| E | Day in week (Text) |
| a | Am/pm marker |
| H | Hour in day (0-23) |
| k | Hour in day (1-24) |
| K | Hour in am/pm (0-11) |
| h | Hour in am/pm (1-12) |
| m | Minute in hour |
| s | Second in minute |
| S | Millisecond |
| z | General time zone |
| Z | RFC 822 time zone |

Figure C.3: Pattern characters for formatting numbers

**compact:** The contents of the collection are displayed in a compact, comma-separated horizontal list.

**il:** Displays the contents of the collection in a vertical bullet (item) list.

**ol:** Displays the contents of the collection in a numbered (ordered) list.

**table:** The contents of the collection are displayed in a table. "table" is followed by a colon and a comma-separated list of titles and attributes, defining the columns of the table. For instance, the following table format shows the horizontal and vertical resolution (= the size in pixels) of a device in a table:

```
table:Screen Width,screenWidth,Screen Height,screenHeight
```

**simple:** The default format, a vertical list of instances separated by line breaks

All options except from *table* allow to provide a secondary format separated from the collection format by a colon. The secondary format is applied to the contents of the collection. For example, applying the format string "ul: #,##0.00" to a collection of numbers generates a bullet list, where each number is formatted according to the second part of the format string.

# Appendix D

# Extension Interfaces

When building Infolayer-based systems, it may be desirable to extend the Infolayer itself with specialized functionality that is hard or impossible to realize by menas of UML and OCL. For instance, an existing Java chart generation framework has been added to the Infolayer system for a fuel consumption tracking service that required graphical reports.

The Infolayer system provides several options for extensions:

- It is possible to map Java classes to classifiers in the UML model, thus making them accessible in OCL expressions.
- Special-purpose request handlers, such as the chart image generator, can be added to the Infolayer Servlet.
- Custom template elements can be added to the System.
- New template loaders for different content types can be added.
- The full runtime model can be accessed from Java.

## D.1 Making Java Classes available in OCL

The simplest extension of the Infolayer is to make Java classes available in OCL expressions. For that purpose, Java classes can be bound to UML data types. To link a UML data type and a Java class, the UML data type must have a tagged value *il-javaclass*. The value denotes the fully qualified name of the Java class. The Java class and all dependencies must be accessible from the Servlet at runtime.

If no methods are declared for the data type, all public methods of the Java class are made visible to OCL expressions. However, this way it is not possible for the

system to determine whether a method has any side effects, so all methods are registered as non-query methods. If operations are declared for the data type, the Java method with a matching name and signature is made available through the operation declaration. In this case, only the declared operations are available. The query property of the operation must match the actual behavior of the Java method. The registered Java class must provide a public constructor without parameters.

Instances of data types bound to Java classes do not have an "identity". It is not recommended to declare attributes having a data type that is not predefined. If attributes of non-predefined types are declared, it is necessary that the corresponding Java class provides a public constructor with a String parameter, which must be sufficient to construct an instance from its *toString()* presentation.

## D.2 Custom Request Handlers

In several cases, the Infolayer system needs to perform actions such as logging in users or handling method invocations via the HTTP interface that go beyond the capabilities of the XML template mechanism. For those cases, the Infolayer provides specialized request handlers. When receiving HTTP requests with an URL of the form "<base-URL>/get_<type>" or "<base-URL>/post_<type>", the Infolayer servlet searches for a class named *org.infolayer.servlet.Process_<type>*, implementing one of the interfaces *org.infolayer.servlet.GetHandler* or *org.infolayer.servlet.PostHandler*, depending on the request type. Predefined handlers are:

**Process_chart** (GetHandler) Generates a chart image.

**Process_cookies** Called by *IlRequest.setCookie()* to set cookie values.

**Process_download** Generates a ZIP files from content selected for download.

**Process_edit** Performes modifications of an instance, usually resulting from an edit fotm.

**Process_exec** (GetHandler, PostHandler) Executes an OCL or ASOQ operation. If the operation is not free of side effects, the HTTP POST method must be used.

**Process_log** (GetHandler) Displays system logging information.

**Process_login** (PostHandler) Tries to log in a user.

**Process_logout** (PostHandler) Logs out the current user.

**Process_query** (GetHandler) Performs a query.

**Process_statemachine** (GetHandler) Renders an image of a state machine, including the current state.

**Process_trigger** (PostHandler) Triggers a state machine transition.

The predefined handlers are described in more detail in 7.5. By adding own implementations of *org.infolayer.servlet.GetHandler* or *org.infolayer.servlet.PostHandler*, following the naming convention, it is possible to install custom request handlers. For more details, please refer to the Javadoc documentation of the above mentioned interfaces.

## D.3 Template Elements

Template elements are used in the Infolayer for XML transformations: an XML template element is generating XML code, depending on the definition of the behavior of the element. The predefined elements are described in chapter 7.

Using a dynamic class loading mechanism similar to the mechanism used for request handlers, it is possible to add custom template elements. When encountering an element in a template namespace, the template loader looks in a package depending on the namespace for a Java class named *TE_<element-name>*. All template elements must extend the class *org.infolayer.templates.TemplateElement*. The most important method is *apply()*, which receives an XML output stream, the current request and variable bindings as parameters. The apply method should be filled with the replacement functionality of the template. For more information, please refer to the Javadoc documentation of *org.infolayer.templates.TemplateElement*.

## D.4 Content Type Handling

The mapping from a template namespace to a Java package is defined by an instance of the class *org.infolayer.templates.TemplateLoader*. Template loaders are activated in the configuration file *conf/httpd.conf* using an *addTemplateLoader* statement.

To add support for a new content type, it is necessary to add an *addLoader* entry to the Infolayer *httpd.conf* file. The first parameter of the entry is the class name of the template loader, the second the base directory where templates of the matching content type are searched. The following parameters are content type extensions that are assigned to the loader. For instance, the *httpd.conf* entry for the HTML loader, assigning the HTML loader to the source directory *%basepath%/html* and the file extenstions *.html* and *.htm* looks as follows:

```
addTemplateLoader
  org.infolayer.templates.html.HtmlLoader html .html .htm
```

The following predefined template loaders are available:

**org.infolayer.templates.TemplateLoader** Loader for general XML templates. Defines the namespace *http://infolayer.org/templates*.

**org.infolayer.templates.html.HtmlLoader** Loader for XHTML templates. Defines the namespace *http://infolayer.org/templates/html* for XHTML-specific template elements. The template element package is *org.infolayer.templates.html*.

**org.infolayer.templates.plain.PlainLoader** Loader that is optimized for the generation of non-XML text files, such as CSV tables. The generated XML code must not contain XML elements. The document declaration, XML comments, and processing instructions are omitted from the output automatically. All character entities in the output are replaced by the corresponding characters.

**org.infolayer.templates.rdf.RdfLoader** Loader for RDF templates. Does not define any specific template elements, but if custom elements are added to the package *org.infolayer.templates.rdf*, they will be available via the namespace *http://infolayer.org/templates/rdf*.

**org.infolayer.templates.wml.WmlLoader** Loader for Wireless Markup Language (WML). Defines the namespace *http://infolayer.org/templates/html* for XHTML-specific template elements. The template element package is *org.infolayer.templates.wml*.

**org.infolayer.templates.pdf.Loader** handles the conversion of XML-FOP files to PDF files. It can be used to generate all kinds of printable forms or reports. This loader expects source files to have the file extension ".fo"

A mapping between the mime types and content type extensions is defined in the file *mime.types*; the XML namespace mapping is fixed for all template loaders and cannot be changed.

## D.5 Accessing the Model from Java

A model executed in the Infolayer system is fully accessible from Java. The Il-Request object—handed to the request hander and template elements—provides access to the model itself using the method *getModel()*. Java implementations of

UML classes are contained in the package *org.infolayer.model* and prefixed with the String "Uml". Some of the most important classes are:

**org.infolayer.model.UmlModel** Represents the UML model. Provides access to model and system properties and classifiers.

**org.infolayer.model.UmlClassifier** Base class for classes and data types. Provides access to instances and methods.

**org.infolayer.model.UmlClass** Represents classes. Provides access to instances, operations, and properties.

**org.infolayer.model.UmlObject** Represents instances.

**org.infolayer.model.UmlDatatype** Represents data types.

**org.infolayer.model.UmlEnumeration** Represents enumerations. Provides access to enumeration literals.

**org.infolayer.model.UmlOperation** Represents operations. Provides access to the parameter and result types and is able to perform the execution of an operation.

**org.infolayer.model.UmlAttribute** Represents attributes. Provides access to the type and cardinality and set- and get-methods.

**org.infolayer.model.UmlAssociationEnd** Represents association ends. Similar to UmlAttribute, but additionally provides access to the association and opposite end.

**org.infolayer.util.OclCollection** OCL collection interface.

**org.infolayer.util.DefaultOclCollection** Java collection based implementation of the OclCollection interface.

Detailed information about those classes and interfaces and links to other relevant classes are available in the Infolayer Javadoc documentation.

# Appendix E

# OCL and ASOQ Reference

This appendix describes all predefined methods available in the Infolayer system. The operation descriptions are annotated as described in table E. Operations without any of the those annotations are referential transparent OCL query operations.

## E.1 Basic OCL Types

### E.1.1 OclAny

#### Operations

**toString(): String [I]**
Returns a string representation of `self`. For primitive types, a canonical String representation is returned. For objects, the default implementation returns the name, the title or the login attribute if available. If not, the ID of the object is returned.

**oclType(): OclType**
Returns the type of `self`.

**oclIsTypeOf(t: OclType): Boolean**
Determines whether `self` is of the type `t`.

**oclIsKindOf(t: OclType): Boolean**
True if `t` is a type or super type of `self`.

**=(a: OclAny): Boolean**
True if `self` equals `a`. Please note that all comparisons wiht OclUndefined including '=' yield OclUndefined. Hence, please use `oclIsUndefined()` to

| Annotation | Description |
|---|---|
| [I] | The operation is an Infolayer-specific extension and not part of the OCL specification. |
| [N] | The operation is *not* referential transparent: the evaluation result of the operation does not only depend on the parameters. Thus, the result cannot be pre-calculated and the operation cannot be used in constant expressions. |
| [S] | The operation may have side effects and thus can be used only in ASOQ but not in OCL expressions. |

Table E.1: Operation Annotations

test for undefined values.
Syntax and SQL template: `self = a`

**<>(a: OclAny): Boolean**
True if `self` is different from `a`.
Syntax and SQL template: `self <> a`

**<(a: OclAny): Boolean**
True if `self` is less than `a`. By default, the comparison operations are based on the `toString()` representations of `self` and `a`. Sub types should overload this operation.
Syntax and SQL template: `self < a`

**>(a: OclAny): Boolean**
True if `self` is greater than `a`.
Syntax and SQL template: `self > a`

**<=(a: OclAny): Boolean**
True if `self` is less than or equal to `a`.
Syntax and SQL template: `self <= a`

**>=(a: OclAny): Boolean**
True if `self` is greater than or equal to `a`.
Syntax and SQL template: `self >= a`

**oclIsUndefined(): Boolean**
True if `self` is `OclUndefined`, false otherwise.

### E.1.2 Boolean

#### Operations

**not(): Boolean**
Returns `true` if `self` is false.
Syntax: not `self`

**and(b: Boolean): Boolean**
True if `self` and `b` are both true.
Syntax: `self` and `b`

**implies(b: Boolean): Boolean**
True if `self` is `false` or both, `self` and `b`, are true.
Syntax: `self` implies `b`

**or(b: Boolean): Boolean**
True if `self` or `b` is true.
Syntax: `self` or `b`

**xor(b: Boolean): Boolean**
True if either `self` or `b` is true, but not both.
Syntax: `self` xor `b`

### E.1.3 Real

#### Operations

**-(): Real**
Returns (0 - `self`)
Syntax and SQL template: - `self`

**abs(): Real**
Returns the absolute value of `self`.

**floor(): Integer**
Returns the largest integer which is less than or equal to `self`.

**round(): Integer**
Returns integer that is closest to `self`. If two integers are equally close to `self`, the largest is returned.

**+(r: Real): Real**
Returns the sum of `self` and `r`
Syntax and SQL template: `self` + `r`

**-(r: Real): Real**
Returns the difference of `self` and `r`.
Syntax and SQL template: `self - r`

***(r: Real): Real**
Returns the product of `self` and `r`.
Syntax and SQL template: `self * r`

**/(r: Real): Real**
Returns the quotient of `self` and `r`.
Syntax and SQL template: `self / r`

**max(r: Real): Real**
Returns the maximum of `self` and `r`.

**min(r: Real): Real**
Returns the minimum of `self` and `r`.

**toString(s: String): String [I]**
Returns a string representation of `self`, formatted according to `s`.

### E.1.4 Integer

#### Operations

**-(): Integer**
Returns (0 -`i`).
Syntax and SQL template: `-self`

**abs(): Integer**
Returns the absolute value of `self`.

**+(i: Integer): Integer**
Returns the sum of `self` and `i`.
Syntax and SQL template: `self + i`

**-(i: Integer): Integer**
Returns the difference of `self` and `i`.
Syntax and SQL template: `self - i`

***(i: Integer): Integer**
Returns the product of `self` and `i`.
Syntax and SQL template: `self * i`

**div(i: Integer): Integer**
Returns the number of times `i` completely fits within `self`.
Syntax: `self` div `i`

**mod(i: Integer): Integer**
Returns `self` modulo `i`.
Syntax: `self mod i`

**max(i: Integer): Integer**
Returns the maximum of `self` and `i`.

**min(i: Integer): Integer**
Returns the minimum of `self` and `i`.

**toChar(): String [I]**
Returns a string of length 1, containing the unicode character with the value `self`.

### E.1.5 String

#### Operations

**size(): Integer**
Returns the length of `s`.

**concat(s: String): String**
Returns the concatenation of `self` and `s`.
SQL template: `self | s`

**+(a: OclAny): String [I]**
Returns the concatenation of `self` and `o`. Equivalent to `self.concat(o.toString())`
Syntax: `self + a`; SQL template: `self | a`

**toUpper(): String [I]**
Returns a copy of `self` with all characters converted to upper case.
SQL template: upper(`self`)

**toLower(): String [I]**
Returns a copy of `self` with all characters converted to lower case.
SQL template: lower(`self`)

**substring(i1: Integer, i2: Integer): String**
Returns the substring of `self`, starting at the index position `i1`, ranging to and including `i2`. Please note that the index of the first character of a string is 1.

**substring(i: Integer): String [I]**
Returns the substring of `self`, starting at the index position `i`. Please note that the index of the first character of a string is 1.

**pos(s: String): Integer [I]**
Deperecated. Returns the first index position of `s` in `self`. If `s1` cannot be found in `s`, 0 is retuned.
SQL template: position(`s` in `self`)

**indexOf(s: String): Integer [I]**
Returns the first index position of `s` in `self`. If `s1` cannot be found in `s`, `OclUndefined` is retuned.
SQL template: position(`s` in `self`)

**toInteger(): Integer**
Converts `self` to an integer value.

**toInteger(s: String): Integer [I]**
Converts `self` to an integer value, assuming that `s` is formatted according to `s`.

**toReal(): Real**
Converts `self` to a real value.

**toReal(s: String): Real [I]**
Converts `self` to a real value, assuming that `self` is formatted according to `s`.

**toDateTime(): DateTime [I]**
Converts `self` from the ISO time format to a `DateTime` value.

**toDateTime(s: String): DateTime [I]**
Converts `self` to a DateTime value that is formatted with respect to `s` to a `DateTime` value.

**startsWith(s: String): Boolean [I]**
Returns true if `self` begins with `s`.
SQL template: position(`s` in `self`) = 1

**endsWith(s: String): Boolean [I]**
True if `self` ends with `s`.

**replace(s1: String, s2: String): String [I]**
Replaces all occurences of `s1` in `self` by `s2`.

**toAscii(): String**
Converts `self` to a string that contains only ASCII characters (Unicodes 32-127). Unsupported characters are replaced by a question mark.

**at(i: Integer): Integer**
Returns the Unicode value of the `i`-th character of `self`.

## E.2 Predefined Data Types

The Infolayer system provides two additional data types, *DateTime* to store date and time values and *Binary* for binary values.

### E.2.1 DateTime

#### Operations

**toString(s: String): String**
Returns `self` as string, formatted with respect to `s`.

**add(i: Integer): DateTime**
Returns a new date with `i` milliseconds added to `self`.

**getYear(): Integer**
Returns the `year` component of `self`.

**getMonth(): Integer**
Returns the `month` component of `self`.

**getDayOfMonth(): Integer**
Returns the `day` component of `self`.

**getMinute(): Integer**
Returns the `minute` component of `self`.

**getHour(): Integer**
Returns the `hour` component of `self`.

**toInteger(): Integer**
Returns the time in milliseconds since 1.1.1970 as integer value.

### E.2.2 Binary

#### Static Operations

**fromUrl(s: String): Binary [S]**
Creates a Binary object from the given URL

#### Operations

**length(): Integer [S]**
Returns the number of bytes of `b`.

## E.3 Collection Types

### E.3.1 Collection(T)

The type `Collection` is the abstract base type of the collection types `Set`, `OrderedSet`, `Bag` and `Sequence`.

#### Operations

**size(): Integer**
The number of elements contained in `self`.

**includes(t: T): Boolean**
True if `self` contains `t`.

**excludes(t: T): Boolean**
True if `self` does not contain `t`.

**count(t: T): Integer**
Returns the number of occurences of `t` in `self`.

**includesAll(c: Collection(T)): Boolean**
True if `self` contains all elements of `c`.

**excludesAll(c: Collection(T)): Boolean**
True if `self` contains none of the elements of `c`.

**isEmpty(): Boolean**
True if `self` does not contain any elements.

**notEmpty(): Boolean**
True if `self` contains at least one element.

**sum(): T**
Returns the sum of the elements. `T` must support the + operation.

**exists(e: OclExpression): Boolean**
Returns true if `e` holds for at least one of the elements contained in `self`.

**forAll(e: OclExpression): Boolean**
True if `e` holds for all elements in `self`.

**isUnique(e: OclExpression): Boolean**
True if `e` evaluates to a different value for each element in `self`.

**sortedBy(e: OclExpression): T2**
Returns a sequence, containing all elements of `self`. The ordering of the sequence is determined by the < operation, applied to the results of `e`.

**iterate(e: OclExpression): T2**
Applies `e` to elements of `self` and returns the value of `e` for the last visited element of `self`. If `self` is a sequence, the elements are iterated in the order of the sequence.

**any(e: OclExpression): T**
Returns any element of `self` for which `e` evaluates to true.

**one(e: OclExpression): Boolean**
True if `e` evaluates to true for exactly one element in `self`.

**min(): T**
Returns the minimum of all elements in `self`.

**max(): T**
Returns the maximum of all elements in `self`.

**asBag(): Bag(T)**
Returns a bag containing all elements of `self`.

**asSet(): Set(T)**
Returns a set containing all elements of `self` withou duplicates.

**asSequence(): Sequence(T)**
Returns `self` if `self` is a sequence, otherwise a sequence containing all elements of `self`. The order is undefined if `self` is a set or bag.

### E.3.2 Set(T)

Represents a mathematical set. Duplicate elements are stored only once.

#### Operations

**union(b: Bag(T)): Bag(T)**
The union of `self` and `b`.

**union(s: Set(T)): Set(T)**
The union of `self` and `s`.

**intersection(b: Bag(T)): Set(T)**
The set of all elements contained in `self` and `b`.

**intersection(s: Set(T)): Set(T)**
The set of all elements contained in `self` and `s`.

**-(s: Set(T)): Set(T)**
The elements of `self` which are not in `s`.

**symmetricDifference(s: Set(T)): Set(T)**
A set containing all the elements that are in `self` or `s`, but not in both sets.

**including(t: T): Set(T)**
A set containing all elements of `self` and `a`.

**excluding(t: T): Set(T)**
A set containing all elements of `self` except from `a`.

**select(e: OclExpression): Set(T)**
The subset of `self` containing the elements for which `e` evaluates to true.

**reject(e: OclExpression): Set(T)**
The subset of `self` containing the elements for which `e` evaluates to false.

**flatten(): T2**
If the element Type is not a collection type, this results in the same set. Otherwise collection type, the result is a set containing all the elements of all the elements of `self`

**collect(e: OclExpression): T2**
Equivalent to `s->collectNested(e1)->flatten()`.

**collectNested(e: OclExpression): T2**
The bag of elements resulting from applying `e` to every element in `self`. The element type of the bag is the type of the given expression.

### E.3.3 OrderedSet(T)

#### Operations

**append(t: T): OrderedSet(T)**
The ordered set consisting of all elements of `self`, followed by `t1`.

**at(i: Integer): T**
The element at index position `i`.

**first(): T**
The element at index position `1`.

**reverse(): OrderedSet(T)**
The ordered set containing all elements of `self` in reverse order.

**last(): T**
The last element of `self`.

**subOrderedSet(i1: Integer, i2: Integer): OrderedSet(T)**
The sub-sequence of `s` starting at index `i1`, ranging to and including `i2`.

**prepend(t: T): OrderedSet(T)**
The ordered set consisting of `t1`, followed by all elements of `self`.

**including(t: T): OrderedSet(T)**
The ordered set containing all elements of `self` and `t`, appended as last element.

**excluding(t: T): Set(T)**
An ordered set containing all elements of `self` except from `t`. The order of the remaining elements is not changed.

**indexOf(t: T): Integer**
The position of `t` in `self`; `OclUndefined` if not contained

**select(e: OclExpression): OrderedSet(T)**
The subset of `self`, containing the elements for which `e` evaluates to true. The order of the elements is preserved.

**reject(e: OclExpression): OrderedSet(T)**
The subset of `self`, containing the elements for which `e` evaluates to false. The order of the remaining elements is preserved.

**collect(e: OclExpression): T2**
Equivalent to `self->collectNested(e)->flatten()`.

**collectNested(e: OclExpression): T2**
The sequence of elements resulting from applying `e` to every element in `self`

**flatten(): T2**
If `T` is not a collection type, this results in `self`. Otherwise, the result is a ordered set containing all the elements of all the elements of `s`

**insertAt(i1: Integer, t2: T): OrderedSet(T)**
Returns a new squence with `t2` inserted at position `i1`, if `t2` is not already contained in `o`.

### E.3.4 Bag(T)

#### Operations

**union(s: Bag(T)): Bag(T)**
The union of `self` and `b`.

**union(s: Set(T)): Bag(T)**
The union of `self` and `s`.

**intersection(s: Bag(T)): Bag(T)**
The bag of all elements contained in `self` and `b`.

**intersection(s: Set(T)): Set(T)**
The set of all elements contained in `self` and `s`.

**including(t: T): Bag(T)**
A bag containing all elements of `self` and `t`.

**excluding(t: T): Set(T)**
A bag containing all elements of `self` except from all occurences of `t`.

**select(e: OclExpression): Bag(T)**
The Bag containing the elements of `self` for which `e` evaluates to true.

**reject(e: OclExpression): Bag(T)**
The Bag containing the elements of `self` for which `e` evaluates to false.

**collect(e: OclExpression): T2**
Equivalent to `self->collectNested(e1)->flatten()`

**flatten(): T2**
If the element type is not a collection type, this operation results in the same bag. Otherwise, the result is a bag containing all the elements of all the elements of `self`

**collectNested(e: OclExpression): T2**
The Bag of elements resulting from applying `e` to every element in `self`. The element type of the resulting bag is the type of `e`.

### E.3.5 Sequence(T)

#### Operations

**union(s: Sequence(T)): Sequence(T)**
The Sequence consisting of all elements of `self`, followed by all elements of `s`.

**append(t: T): Sequence(T)**
The Sequence consisting of all elements of `self`, followed by `t`.

**prepend(t: T): Sequence(T)**
The Sequence consisting of `t`, followed by all elements of `self`.

**subSequence(i1: Integer, i2: Integer): Sequence(T)**
The sub-sequence of `self`, starting at index `i1` and ranging to and including `i2`.

**at(i: Integer): T**
The element at index position `i`.

**first(): T**
The element at index position `1`.

**last(): T**
The last element of the sequence.

**including(t: T): Sequence(T)**
The sequence containing all elements of `self` and `t`, appended as last element.

**reverse(): Sequence(T)**
The sequence containing all elements of `self` in reverse order.

**excluding(t: T): Set(T)**
A sequence containing all elements of `self` except from any occurences of `t`. The order of the remaining objects is not changed.

**select(e: OclExpression): Sequence(T)**
The sub-sequence of `self`, containing the elements for which `e` evaluates to true.

**reject(e: OclExpression): Sequence(T)**
The sub-sequence of `s`, containing the elements for which `e` evaluates to false.

**collect(e: OclExpression): T2**
Equivalent to `s->collectNested(e1)->flatten()`.

**collectNested(e: OclExpression): T2**
The sequence of elements resulting from applying `e` to every element in `s`

**flatten(): T2**
If the element Type is not a collection type this results in the same sequence. If the element type is a collection type, the result is a sequence containing all the elements of all the elements of `s`

**indexOf(t: T): Integer**
The index of the first occurence of `t` in `self`; OclUndefined if not contained

**insertAt(i1: Integer, t2: T): Sequence(T)**
Returns a new ordered set with `t2` inserted at position `i1` and the remaining elements are shifted accordingly.

## E.4 Metamodel Access

### E.4.1 OclModelElement

#### Attributes

**name** The name of `self`.

### Operations

**getTaggedValue(s: String): String [N]**
Returns the tagged value named `s`.

**getTaggedValue(s1: String, s2: String): String [N]**
Returns the tagged value named `s1`, localized to `s2`. The format for `s2` is usually a two-letter ISO country code (see localization configuration files).

## E.4.2 OclType

The static methods of OclAny are at the same time the instance methodsof the type OclType. OclType is the type of OclAny and the supertypeof all meta types. OclType provides the methods listed below to queryinformation about the given type.

### Attributes

**operation** Association to all operations available for this type

**attribute** Association to all properties (attributes and association ends) available for this type

**ownedOperation** Association to all operations (re)declared for this type

**ownedAttribute** Association to all properties (re)declared for this type

### Operations

**name(): String [N]**
Deprecated; included for OCL 1.x compatibility. Please use the `name` attribute (defined at `OclModelElement`) instead. Returns the name of `self`.

**supertypes(): Set(OclType) [N]**
Returns all immediate super types of `self`.

**allSupertypes(): Set(OclType) [N]**
Returns the transitive closure of the set of all super types of `self`

**subtypes(): Set(OclType) [N]**
Returns all immedidate subtypes of `self`

**allSubtypes(): Set(OclType) [N]**
Returns the transitive clousure of all subtypes of `self`.

**allInstances(): Set(TypeOf) [N]**
Returns the set of all instances of `self`, including all instances of subtypees.

**instances(): Set(TypeOf) [I] [N]**
Returns the set of all direct instances of `type`, not including instances of subtypees. This operation is specific to the Infolayer and not part of the OCL standard.

**getInstance(s: String): TypeOf [I] [N]**
Returns the instance with the given id, or `OclUndefined`, if there is no such instance for `t` and its subtypees. This operation is specific to the Infolayer and not part of the OCL standard.

**createInstance(): TypeOf [I] [S]**
This method returns a new Instance of `self`.

**canCreate(): Boolean [N]**
Returns true if the current user may create new instances of `self`.

**canQuery(s: String): Boolean [N]**
True if the current user may query `self`.

### E.4.3 OclOperation

Represents an operation. Currently, only the features inherited from OclModelElement are supported.

## E.5 Predefined Classes

The Infolayer system contains the predefined classes *Object*, *Infolayer*, and *File*.

### E.5.1 Object

#### Operations

**getId(): String**
Returns the unique ID of `self`.

**onChange(s: String): Void [S]**
This method is called when a property value has changed. The name of the property is given as parameter. Overwrite this method for classes where you are interested in particular changes. Please note that this method only reports changes made via the user interface (the same holds for onCreate and onDelete).

**onCreate(): Void [S]**
This method is called when a new instance has been created via the user inteface.

**onDelete(): Void [S]**
This method is called immediately before an instance is deleted via the user interface.

**clone(): Object [S]**
Creates a clone of an instance. Composite objects are also cloned, other associated objects are linked by reference.

**delete(): Void [S]**
This method deletes `self`, including all links to `self`.

**canRead(): Boolean [N]**
Queries whether the current user has read access to `self`.

**canWrite(): Boolean [N]**
Queries whether the current user may modify or delete `self`.

### E.5.2 Infolayer

The Infolayer class contains a set of static methods providing general information about the current state of the Infolayer.

#### Static Operations

**getCurrentUser(): User [I] [N]**
Returns the user logged in to this session context. Please note that the result type may vary if the user class is changed using the `il-userclass` tagged value.

**getCurrentDateTime(): DateTime [I] [N]**
Returns the current date and time.

**getMimeType(s: String): String [I] [N]**
Returns the mime type for the given file name or extension, as defined in conf/mime.types.

**getConfigurationProperty(s: String): String [I] [N]**
Returns the system/servlet/model property with the given name.

**getActiveUsers(): Set(User) [I] [N]**
Returns the set of users currently logged in. As for `getCurrentUser()`, the actual return type may differ depending on the `il-userclass` tagged value.

**getSystemProperty(s: String): String [S]**
Returns the Java property with the given name.

**eval(s: String): OclAny [S]**
Evaluates an arbitrary expression.

**resetPassword(u: User): String [S]**
Returns a new password.

### E.5.3 File

The class File provides a binary data field and a name field

#### Attributes

**name** The name of the file

**data** Binary content of the file

**counter** Number of downloads

## E.6 Special Purpose Classifiers

In the XML template mechanism, the Infolayer provides the predefined classifiers *IlRequest* and *IlUrl* for access to properties of the current HTTP request and to construct URLs.

### E.6.1 IlRequest

#### Operations

**getCookie(s: String): String**
Returns the value of the cookie named `s`.

**getHeader(s: String): String**
Returns the value of the request header named `s`.

**getHost(): String**
Returns the name of the host serving the request.

**getI18n(s: String): String**
Returns a localized version of `s`, depending on `getLang()`.

**getLang(): String**
Returns the language setting of the browser, or, if overriden by a cookie, the cookie value.

**getRequestUrl(): IlUrl**
Returns the URL of the original request, without modifications reflecting internal state changes or normalizations.

**getUrl(): IlUrl**
Returns the URL of the request, including modifications reflecting internal state changes or normalizations.

**setCookie(s1: String, s2: String): Void [S]**
Sets the cookie `s1` to the value `s2`.

### E.6.2 IlUrl

#### Operations

**getObject(s1: String, t2: OclType): OclAny**
Returns the object denoted by the URL parameter `s`.

**getPath(): String**
Returns the path part of the `self`.

**getProperty(s: String): String**
Returns the value of the URL query parameter `s`.

**getParameterValues(s: String): Bag(String)**
Returns all values of the URL query parameter named `s` as a bag of strings.

**isLocal(): Boolean**
Returns true if `self` is a local URL.

**withProperty(s1: String, s2: String): IlUrl**
Returns a copy with a query parameter `s1`, having the value `s2`.

**withPath(s: String): IlUrl**
Returns a copy of `self`, but with the URL path set to `s`.

**withObject(s1: String, a2: OclAny): IlUrl**
Returns a copy of `i`, but with a URL parameter named `s1`, containing `o2` encoded as OCL literal.

# Appendix F

# XML Template Elements

This appendix describes all template elements that are available in the Infolayer system. XHTML and WML specific elements are described in separate sections.

## F.1 General XML Template Elements

General XML template elements can be used in all templates, independent from the target language.

### F.1.1 <t:assign>

The <t:assign>-element can be used to alter the value of variables declared by in a <t:variable>-element.

#### Attributes

**name** (required) The name of the variable to be declared.

**expr** (required) An OCL-expression determining the initial value and type of the variable.

### F.1.2 <t:attribute>

The <t:attribute> element generates an XML attribute for a previous <t:element>-element, or for a literal XML element embedded in the template source code. The content of the attribute-element is used to determine the value of the attribute.

#### Attributes

**name** The name of the generated attribute

**namespace** The namespace of the generated attribute

### F.1.3 <t:call>

The `<t:call>` element is replaced by the content of the template denoted by the `template` or `file` attribute. The `expr` attribute denotes the context of the applied template, if present. Otherwise, the current context becomes the template context.

The `file` and `template` attributes are mutally exclusive. The `file` attribute denotes the path to the template file that is called. A relative path is resolved relative to the current directory; an absolute path is resolved relative to the base directory for static tenplates.

The `template` parameter is resolved to a template file by a dynamic lookup operation. A template file with the given name and the extension ".inc" is searched in the dynamic template directory depending on the type of the template context. Except from the different file extension, the lookup mechanism is identical to the lookup mechanism for top-level templates (used for URLs starting with "/auto").

If the called template contains an `<t:inner />` element, template processing continutes with the child elements of the `<t:call>` element. For the child elements, the original evaluation context is preserved, it is not influenced by processing the called template.

`<t:withParam>` elements can be used to transfer parameters to the called template.

#### Attributes

**file** Name of the template file to be called.

**expr** The evaluation result of this expression is used as context for the called template.

**template** Name of the template to be called.

**Restrictions for Child Elements** If `<t:withParam>` child elements are present, they must precede any other content.

### F.1.4 <t:case>

Please refer to the description of <t:switch>.

### F.1.5 <t:choose>

The choose element allows conditional processing of sub-elements. Template processing continues in the first <t:when>-sub-element, where the evaluation result of the expr-attribute is true. The evaluation context remains unchanged. If a matching case is found, no further sub-elements are considered. If no matching case is found, template processing continues at the optional <t:otherwise> element.

**Restrictions for Child Elements** Allowed sub-elements are any number of <t:when> elements, followed by up to one <t:otherwise> element.

### F.1.6 <t:comment>

The element <t:comment> generates a comment from the contained text.

### F.1.7 <t:context>

The context element changes the evaluation context for all child elements. The new evaluation context (self) is determined by the expr-attribute.

Without the expr attribute, the <context> may be used as helper element, for instance to hold namespace declarations that shall not appear in the generated XML code.

#### Attributes

**expr** Contains an OCL expression, determining the new evaluation context

### F.1.8 <t:element>

The <t:element> element generates an XML element. The name and namespace ot the element are defined in attributes.

#### Attributes

**name** The name of the generated element

**namespace** The namespace of the generated element

### F.1.9 <t:forAll>

Processes all child nodes for each element of the collection obtained from evaluating the `expr`-attribute. If the type of `expr` is OclType, an implicit call to `allInstances()` is performed to obtain a set. If the `expr` attribute is omitted, it defaults to the expression `self`.

When a `iterator` attribute is given, its value is used as the name of an iterator variable. Otherwise, the evaluation context (self) is set accordingly in each iteration step. The `counter`-attribute allows to declare an integer variable, holding the current element index.

#### Attributes

**expr** Contains an OCL expression that must result in an collection.

**iterator** If present, the value denotes an iterator variable name that is used instead of `self`.

**counter** If present, the given variable name is used to declare an integer variable holding the current element index.

### F.1.10 <t:if>

The `<t:if>` element allows conditional processing of child elements, depending on the `expr`-attribute. Child elements are only processed if the given expression evaluates to true.

#### Attributes

**expr** The boolean evaluation result of the contained OCL expression determines the processing of child elements.

### F.1.11 <t:inner>

See `<t:call>`.

### F.1.12 <t:otherwise>

The `otherwise`-element is only allowed as immediate child element of <`t:choose`> and <`t:switch`>, denoting the default branch. For a detailed description, please refer to the corresponding sections.

### F.1.13 <t:param>

The <`t:param`>-element declares a variable that is initialized from a parameter that is set via the <`t:withParam`> element. The variable can be accessed in template elements following the declaration. The scope of the variable ends at the closing tag of the immediate parent element.

#### Attributes

**name** (required) The name of the variable to be declared.

**expr** If the parameter is not set, this expression is used to determine a default value.

**type** The type of the variable. It must match the actual type set by the `withParam` element. If not set, the type defaults to `OclAny`.

### F.1.14 <t:recurse>

Reserved for future use.

### F.1.15 <t:recursion>

Reserved for future use.

### F.1.16 <t:switch>

The <`t:switch`> element allows conditional processing of <`t:case`> sub-elements. The evaluation result of the `expr`-attribute is compared to the evaluation result of the `expr`-attribute of the `case` elements. Template processing continues in the first matching <*t:case*> element. The evaluation context remains unchanged.

When a matching case is found, no further sub-elements are examined; if no matching case is found, template processing continues in the optional <*t:otherwise*> element.

#### Attributes

**expr** (required) : The result of the given OCL expression is compared to the expressions given in <t:case> sub-elements.

**Restrictions for Child Elements** Allowed sub-elements are any number of <t:case> elements, followed by up to one <t:else> element.

### F.1.17 <t:text>

Writes the contained text to the target document.

### F.1.18 <t:valueOf>

Inserts the evaluation result of the OCL expression contained in the expr-attribute. If an format-attribute is present, the contained format string is taken into account. Child elements of <t:valueOf> are processed only if the evaluation results in OclUndefined.

#### Attributes

**expr** (required) Determines the replacement value.

**format** Formatting instructions for the evaluation result.

### F.1.19 <t:variable>

The <t:variable>-element declares a variable. The variable can be accessed in OCL expressions in template elements following the variable declaration. The scope of the variable ends at the closing tag of the immediate parent element. In contrast fo XSLT, it is possible to alter the value of a variable with the <t:assign>-element.

#### Attributes

**name** (required) The name of the variable to be declared.

**expr** (required) An OCL-expression determining the initial value and type of the variable.

### F.1.20 <t:when>

See <t:switch>.

### F.1.21 <t:withParam>

The <t:withParam> element allows the transmission of parameters to other templates, referenced by the template element <t:call> and the XHTML template element <t:link>. In the target template, the parameters can be queried using the template element <t:param>.

#### Attributes

**expr** An OCL Expression determining the value of the parameter.

**name** The name of the parameter.

## F.2 XHTML Template Elements

XHTML template elements provide additional features that are useful for the generation of (X)HTML pages.

### F.2.1 <t:actions>

Generates a set of buttons: All public operations available for `self`, where the current user meets the permissions, plus additional edit or query buttons, depending on the type of `self` and the user permissions.

### F.2.2 <t:cancel>

If it is embedded in an <t:form> element, the <t:cancel> element creates a cancel button. The `label` attribute can be used to set a custom label. If no custom label is provided, "cancel" or a translation is displayed.

#### Attributes

**label** The optional label of the cancel button.

**followup** A followup command, determining the next page.

### F.2.3 <t:column>

This element determines the content of a column in a dynamic table. It can only be used as immediate child element of the <t:table> element.

#### Attributes

**title** The Column Title.

**expr** Expression determining the content of the colum. If not present, the child elements of the `colum`-element are used to determine the column content. If the `expr`-attribute is present, the `<t:column>` element must be empty.

**format** Used to format the result of the `expr`-attribute.

**sortedBy** Expression determining the row order of the table if this column is selected. If not given, `expr` is used to determine the order.

### F.2.4 <t:control>

Displays a button that performs a HTTP-GET for the address determined by the `followup`-Attribute. Can be used as replacement for a link when a coherent look with other buttons is desired.

#### Attributes

**followup** Determines the URL of the page that is requested when the button is pressed.

### F.2.5 <t:form>

Writes an HTML form element, including a method and URL depending on the `type` and `method` attributes. The type attribute is added to the current URL for all sub elements, so they can determine easily whether they are included in an active form.

#### Attributes

**type** A type constant determining the request type and URL for the form. The request URL is constructed by prefixing the type with "get-" or "post-", depending on the method, and appending ".html".

**method** Determines the request method used to submit the form. May be POST or GET. If omitted, the default value is GET if the type is query, POST otherwise.

### F.2.6 <t:image>

Displays an image that is contained in the field determined by the `name` attribute. The field must be of type `Binary` or `File`. The image is scaled to the given `width` or `maxWidth` automatically. Clicking on the image will zoom in the image, and a "zoom out" region will be marked on the zoomed image.

#### Attributes

**name** Name of the field holding the image data.

**width** If present, the image is always scaled to the given width.

**maxWidth** If present, and the image is wider than the value, the image is scaled down accordingly.

**zoomable** Decides whether zooming via clicking the image is enabled. Default is true.

### F.2.7 <t:link>

The <t:link> element creates a hyperlink to the object denoted by the `expr` attribute. Other uninterpreted attributes are added to the generated `<a>` element.

#### Attributes

**expr** The contained OCL expression is evaluated to determine the link address.

**template** If an template attribute is present, the given template is used instead of the default template of the object.

### F.2.8 <t:login>

Generates a complete login form, including input elements for the user and password, as well as login and cancel buttons. If a user is logged in already, a logout button is displayed.

This element must not have any content.

#### Attributes

**followup** Contains an optional followup URL command. By default, the previous page will be displayed when one of the buttons is pressed.

**type** Must contain one of the values `input` or `select`. If the type is select, the user to log in can be selected from a list. Otherwise, the log in name must be entered in a text field.

### F.2.9 <t:messages>

The `<t:messages>` element displays messages such as "Login Failed", resulting from processing forms. Thus, every page should contain a `message` element at a prominent place. For the default pages, a message element is contained in `frame.html`.

### F.2.10 <t:operation>

The `<t:operation>` element creates a button that performs the operation denoted by the `name` attribute. The button can only be activated successfully if the current user has the required permissions.

If the operation is a query operation, a HTTP GET request is used. Otherwise, if the method has side effects, an HTTP POST request is generated.

If the operation provides a return value, it is used to generate the result page.

#### Attributes

**name** The name of the operation to be executed.

**label** An optional labed for the button. If not present, the name of the operation will be displayed.

**followup** A followup command, determining the next page.

### F.2.11 <t:operations>

The `<t:operations>` element generates a table of buttons which are applicable to the current context for the current user.

### F.2.12 <t:properties>

The <t:properties> element generates a table consisting of labels and properties of the current object. This element can be used in regular pages for displaying object properties. Inside <t:form> elements, the <t:properties> element can be used to edit objects or to search for objects.

Please note that the context must be an instance for the display and edit mode, but a class for the query mode.

**names** A comma separated list of property names to be displayed.

### F.2.13 <t:property>

The <t:property> element displays the object property denoted by the `name` attribute. If the `property` element is embedded in a `form` of type `edit`, and the user has the permission to edit the given property, an input element corresponding to the object type is created.

If the `property` is embedded in a `form` of type `query`, and the property type is queryable, an input element for a query is displayed.

Please note that the context must be an instance for the display and edit mode, but a class for the query mode.

**name** The name of the property to be displayed, edited, or queried.

**compact** If "true", the element does not use empty space to align with other elements.

### F.2.14 <t:submit>

The <t:submit> element displays a submit button if included in a form element. If no `label` attribute is provided, "submit" or a translation is used as default label. The next page to be displayed can be determined by the followup attribute. No followup is interpreted as followup="stay", or, if the current template is a comma separated list, the remainter of the list is interpreted as a followup="goto-template:..." command.

The <t:submit> element must not have any child elements.

**Attributes**

**label** An optional label for the submit button.

**followup** A followup command, determining the next page.

### F.2.15 <t:table>

Generates a HTML table. The columns of the table are defined by <t:column>-sub elements; No other immediate content is permitted.

### F.2.16 <t:tree>

The `<t:tree>` element displays a nested tree of instances or classes. If the context or root element is a class, the class inheritance tree is displayed. Otherwise, the children attribute and the tagged value il-children are used to determine an association. The tree is built by simply following this association.

### F.2.17 <t:valueOf>

The XHTML `<t:valueOf>` element is identical to the general `<t:valueOf>` template element, but provides some additional format implementations for collections like `table` and `li` that cannot be provided for XML in general.

## F.3 WML Template Elements

WML template elements provide additional features that are useful for the generation of (X)HTML pages.

### F.3.1 <t:properties>

Simplified version of the HTML variant of this element (read-only): Displays all displayable properties of the current object (`self`). Properties with a maximum cardinality greater than one are put in separate cards. The optional `names` attribute allows to specify the properties to be displayed.

### F.3.2 <t:property>

Simplified version of the HTML variant of this element (read-only): Displays the value of the property denoted by the `name` attribute.

### F.3.3 <t:valueOf>

The WML eval element is identical to the general eval template element, but displays collections as comma separated lists and adds detail links to objects.