Software and Systems Modeling manuscript No.
(will be inserted by the editor)

A Model-Driven Runtime Environment for Web Applications [⋆]

Stefan Haustein[1], **Joerg Pleumann**[2]

[1] Artificial Intelligence
University of Dortmund
Germany

[2] Software Technology
University of Dortmund
Germany

Received: date / Revised version: date

Abstract A large part of software development these days deals with building so-called Web applications. Many of these applications are database-powered and exhibit a page layout and navigational structure that is close to the class structure of the entities being managed by the system. Also, there is often only limited application-specific business logic. This makes the usual three-tier architectural approach unappealing, because it results in a lot of unnecessary overhead. One possible solution to this problem is the use of model-driven architecture (MDA). A simple platform-independent domain model describing only the entity structure of interest could be transformed into a platform-specific model that incorporates a persistence mechanism and a user interface. Yet, this raises a number of additional problems caused by the one-way, multi-transformational nature of the MDA process. To cope with these problems, the authors propose the notion of a model-driven runtime (MDR) environment that is able to execute a platform-independent model for a specific purpose instead of transforming it. The paper explains the concepts of an MDR that interprets OCL-annotated class diagrams and state machines to realize Web applications. It shows the authors' implementation of the approach, the Infolayer system, which is already used by a number of applications. Experiences from these applications are described, and the approach is compared to others.

[⋆] This is an extended and revised version of a paper originally presented at the UML 2003 conference in San Francisco [23]. The second author has been supported by the German Federal Ministry of Education and Research (BMBF), grant 08NM098.

1 Introduction

A large part of software development these days deals with building so-called Web applications, that is, server-sided applications that are remotely accessed via the Internet using a standard Web client. Communication between client and server is based on the Hypertext Transfer Protocol (HTTP) and uses the Hypertext Markup Language (HTML) for content description. For nontrivial applications, static HTML pages are usually not sufficient – instead, each page exists in two variants: the server holds the original page that consists of HTML code and embedded scripting commands which, for example, access the content of some underlying database. This template-like page is processed on the server, resulting in a pure HTML page which is then delivered to the client.

Web applications often employ a traditional three-tier architectural approach: the lower tier provides a persistence mechanism for the entities the application deals with. The upper tier provides either the HTML user interface meant to be consumed by humans or a communication interface for other applications based on, for example, the Simple Object Access Protocol (SOAP). The middle tier ties the other two together and implements the application's business logic. While this approach is relatively common, it raises a number of problems:

– The database used in the persistence tier is likely to be a relational one. Given that the rest of the application is modeled using the Unified Modeling Language (UML) [11,2] and later implemented using an object-oriented programming language like Java, a mapping between the object-oriented and relational worlds is necessary. This mapping is further complicated by the need for a normalization of database tables and the expressive mismatch between the Standard Query Language (SQL) and a modern object-oriented language like Java.
– As mentioned, most approaches use some form of scripting language to separate static and dynamic portions of a Web page. While it is possible that the scripts are implemented in (roughly) the same language as the rest of the application – like in the combination of Java and Java Server Pages (JSP) – this is not a necessity. It may well be a different language like PHP or Perl, which results in at least five languages being used in the overall system: UML, SQL, HTML, Java plus the scripting language. This poses high demands on the developers' skills, it raises development time and cost and it complicates maintenance.
– In a significant number of cases, the application's business logic is pretty uniform. Take, for example, the typical simple Web application used to realize the Web site of a university department (as depicted in Fig. 1). The database stores instances of some entity classes, and the user interface provides access to them. Often, even the navigational structure of the user interface is close to the entities' class structure, that is, there is a correspondence between domain classes and HTML pages used to display, manipulate or query instances of these classes. If the logic is

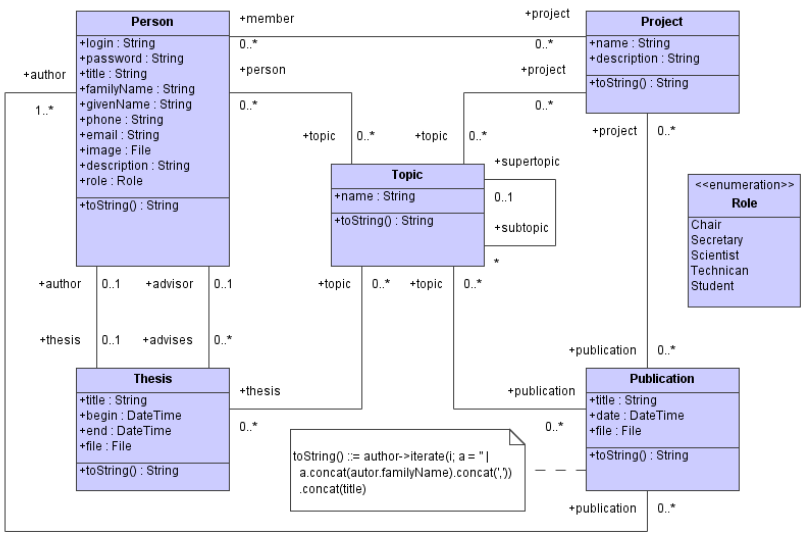


Fig. 1 A (simplified) domain model of a university department

not application-specific, it seems unnecessary to explicitly model and implement it. Instead of going through the effort of the full three-tier approach, one would want to focus on the entities and their presentation in the user interface and leave the rest to a tool.

A solution to the aforementioned problems, as mandated by OMG, is the use of Model Driven Architecture (MDA) [10,9]. For a given problem, MDA proposes that first a platform-independent model (PIM) be created. This model is then transformed to one or more platform-specific models (PSM) using appropriate transformation rules. In the given domain of Web applications, the PIM could encompass application-specific information like the domain model and the application's business logic. From the potentially infinite number of possible PSMs, a specific one could incorporate a user interface based on HTML and a persistence layer employing a relational database. A PSM can in principle be refined into an even more platform-specific model. Yet, at some point programming language code has to be emitted that can be compiled into an executable application.

From a programmer's point of view, MDA is not completely new. It is extending the traditional idea of a compiler to the earlier phases of the software development process, that is, to the models. While this is surely a powerful idea, it has some consequences for the overall process as well as the application under development:

– Evolution is complicated. As Heckel and Lohmann [16] have noted before, MDA doesn't pay enough attention to functional evolution of the system. Every such evolutional step, for example a new requirement, induces changes to the PIM or the specification of the transformations from the PIM to the PSMs. In either case, the whole transformational chain up to the executable application has to be applied over and over again. Given that a large part of Web application development deals with the creative process of designing an appropriate user interface, that is, small changes to HTML pages (or their equivalent in the model) are made and evaluated, these time-consuming transformations are likely to hamper development progress.

– Maintenance is complicated. This is due to the fact that the application has not only undergone the transformation(s) inherent in the traditional compilation step, but also additional ones for the models. Tracking a problem in the running application back to its roots in either the PIM or one of the transformations requires that the equivalent to "debugging" information is available for each model that was derived from a less-specific one. Currently, there seems to be no solution for this problem. Also, unless special care is taken, the generated models and source code might be hard to read, since they are not primarily meant to be consumed by humans.
– The process is one-way. While it is in principle possible to modify models generated during a previous transformation step, this is not recommended. Manually changing a PSM or some generated source code potentially results in an inconsistent description of the whole system architecture, since these changes are neither reflected in the other models nor gained "legally" through a transformation. They are lost when a complete rebuild of the system is done starting from the PIM. Thus, until there exists a means to propagate manual changes in any model to the rest of the system architecture, it is best to treat generated models and source code as read-only.

Since all three problems stem from the multiple transformations (or compilations) inherent in the MDA approach, the authors were looking for a solution that suited the Web application domain better. As a result, we propose a slight 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), an MDR implements selected parts of the UML metamodel and interprets them for a given application domain. Just like it is possible to derive multiple PSMs from a single PIM in the pure MDA approach, it is possible to have a number of very different MDRs executing the same PIM for different reasons. In our case, the MDR of interest is one that supports the development of Web applications like the one described in the university scenario of Fig. 1, that is, database-powered applications with limited business logic and a navigational structure that is close to the underlying entity structure.

The rest of this paper is organized as follows: section 2 presents the basic concepts of an MDR for Web applications. It shows how UML is used to describe static and dynamic aspects of a Web application and how these descriptions are interpreted at runtime. Section 3 shows how the user interface of the MDR can be customized using templates. Section 4 presents our implementation of an MDR for Web applications, the Information Layer system, or Infolayer for short. Section 5 describes examples of concrete applications realized with the approach, with experiences from these being given in section 6. The final sections 7 and 8 compare our approach to others and draw a conclusion. A detailed case study is presented in the appendix.

2 Core concepts

The basic idea of an MDR is avoid the transformational steps from the PIM via the PSM and the generated source code to the working application. Instead, the MDR is to *execute* the PIM itself: a UML model designed in a Computer Aided Software Engineering (CASE) tool and exported to the standard Extensible Metadata Interchange (XMI) format supported by most contemporary tools shall be sufficient to invoke the system. The MDR then provides a user interface and a persistence mechanism that would be gained through explicit modeling/implementation or a suitable transformation in the traditional or MDA approach, respectively. To achieve this, the model information, possibly annotated with constraints specified in the Object Constraint Language (OCL) [30], is interpreted in several ways:

1. The model drives the database.
2. The model determines the user interface.
3. The model provides business logic.

The MDR can be seen as a domain-specific UML interpreter the core semantics of which is taken verbatim from the UML specification. Only in those areas that are not covered by the UML specification, we need to define additional semantics. These are typically areas which are specific to the domain of Web applications.

Fig. 2 depicts the system at a very high level of abstraction, with the MDR being in the center. The next sections provide more detail on the various aspects of the interpretation sketched above.

2.1 The model drives the database

At the heart of the UML model fed into the system is a class diagram describing the entities of interest. Based on this structural model, the MDR provides a persistence mechanism that supports creating, accessing, modifying, and deleting instances of the given classes. The semantics of the persistence mechanism is taken directly from the UML specification: attributes store values of the primitive types Boolean, Integer, Float, String, and DateTime. Associations describe which instances can be linked to each other. Inheritance is used to build specialization hierachies.

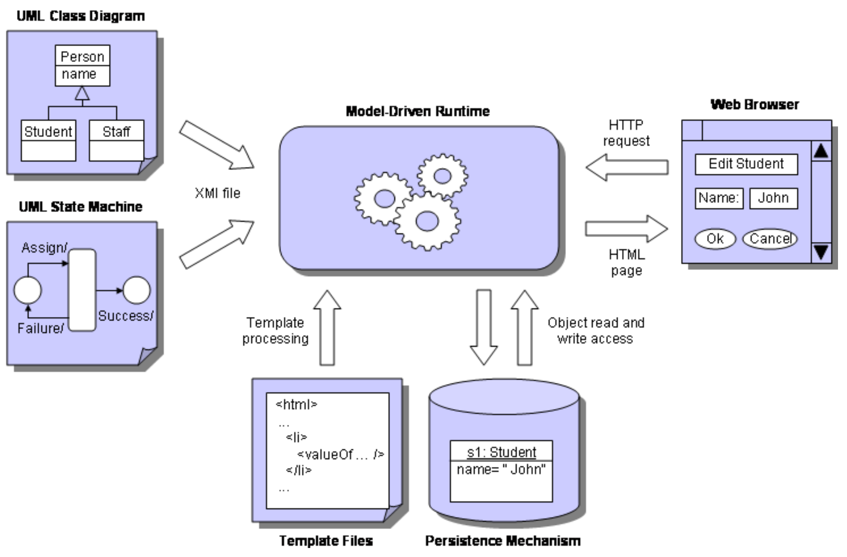


Fig. 2 Overview of an MDR-based Web application

Since we interpret the model instead of generating lower-level code, we do not lose valuable information inherent in the class diagram. Thus it is easy for the MDR to, for instance, treat associations between classes as first-order elements that are kept consistent by the system according to the multiplicities at the association ends.

Changes are made persistent to an underlying storage by serializing the objects. Object configurations that violate OCL constraints specified in the model are rejected. Practically speaking, the MDR makes it possible to construct a persistent object diagram that conforms to the given class diagram.

Operations are supported, too, with OCL serving as the primary language for evaluating any kind of expressions throughout the system:

- Query operations (that is, operations with no side effects) are implemented directly using OCL expressions. OCL is basically used like a functional programming language in that case.
- Non-query operations are implemented using UML action semantics. To minimize the learning effort for users, we chose a superset of OCL as the surface language (the concrete syntax) for action semantics. Details of this action language are presented in [14].

The system knows several predefined classes. One of them is the usual `Object` class that forms the root of the class tree. `Object` provides a built-in operation `toString(): String` the purpose of which is to derive a printable text from an object. It is used and redefined in the same way as, for example, in the Java class libraries. Another predefined class `User` serves as the basis for user management, authentication and access control to classes and objects based on permissions (again expressed in OCL).

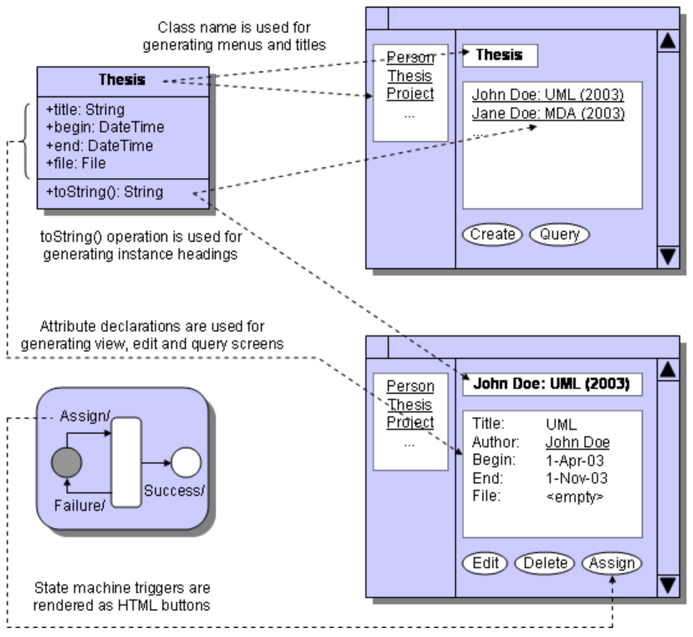


Fig. 3 Generic user interface derived from the model

2.2 The model determines the user interface

An MDR can be accessed in numerous ways, of course, but for the moment we are only interested in Web-based access. Technically, the system needs to run inside a Web server or provide Web server functionality itself. Conceptually, an HTML user interface needs to be generated by the interpreter when a client accesses the system. This interface is based both on the structural information inherent in the UML model and on the existing instances. The mapping from the model to user interface components is straightforward:

- The interface shows a clickable inheritance tree of known classes. When a specific class is selected, its current instances are listed, individual instances can be selected for display, and new instances can be created.
- The system shows for each instance a list of all attributes and associations, with associations being rendered as hyperlinks to the associated objects. The latter supports the user in easily navigating through the whole object diagram.
- When editing an object, the system takes care to restrict the user's input to sensible choices – like exactly those objects that can participate in a certain association under the constraints imposed by the multiplicities at the association ends.
- For querying the database a very similar screen is used. The results of a query are displayed as a list of instances where individual objects can be selected for display.

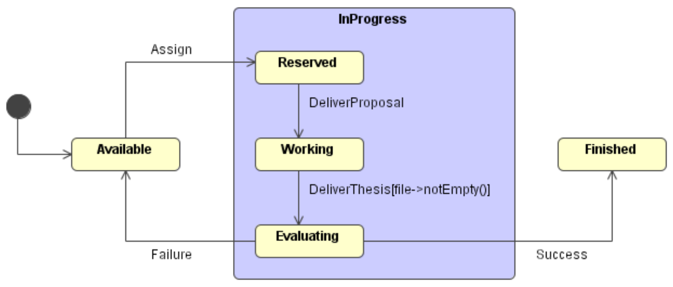


Fig. 4 State machine for the `Thesis` class

Fig. 3 depicts the user interface generated for the `Thesis` class of the university example. Underlined strings denote hyperlinks. Arrows are used to indicate which parts of the user interface are derived from which parts of a class declaration.

2.3 The model provides business logic

Experience with Web applications shows that a number of systems display workflow-like characteristics. Take, for example, the university department model from Fig. 1. The corresponding Web site would usually provide a list of master theses, each of which can be in a different state: a thesis can be *available*, it can be *reserved* for a student who is writing a proposal, it can be work *in progress*, and it can be *finished*. The possible transitions between these states are restricted, and the user interface should enforce these restrictions. For example, one should be able to go from *in progress* to *finished*, but going back should be prohibited. Such behavior is easily specified by means of a UML state machine, as Fig. 4 shows.

The MDR interprets state machines as additional business logic of the system. Every class in the domain model can be annotated with a state machine that describes its behaviour. Once a new object is created at runtime, it not only has all its attributes set to default values, but also starts in its initial state(s).

The statechart interpretation follows the core semantics of the UML specification. Only the generation of the user interface parts require additional semantics:

– When an object is displayed, the user interface shows a list of buttons representing the potential triggers. These are the triggers of exactly those transitions that are enabled, or, more precisely, would be enabled if these events were to enter the system. In Fig. 3, the *Assign* button is displayed because the corresponding trigger is attached to a transition that leaves the currently active state.

- The buttons take into account any guard expressions attached to the transitions. These guards are, again, specified in OCL and may thus encompass both the model information and the existing instances.
- When a button is pressed, the enabled transitions are taken, resulting in a new active state configuration. Actions executed by a transition can be used to modify an object's attributes, for example by assigning values to attributes or associations.
- The new state configuration is made persistent together with the object, just as the attributes and associations are. Once a state machine reaches a terminal state, the object it belongs to is deleted.

If required within an expression or operation, the current active state configuration of an object can be queried using the `OclInState(<state>)` function. As an example, the `toString()` operation of a class could take the active state configuration of an object into account when deriving the resulting string.

3 Customizing the user interface

Being relatively simple and thus not sufficient for real-life applications, the generic user interface needs to be tailorable to specific needs. It is tempting to achieve this on the level of the UML model, too, since it would result in a coherent approach that employs a minimum number of formalisms. Yet, for pragmatic reasons we find it more appropriate to perform this tailoring on the level of HTML pages. This is due to the observation that HTML is also the target language of the system and that a number of powerful and mature tools exist for creating HTML pages. We assume that people responsible for the artistic design of a Web application will be more comfortable with these and prefer them over a CASE tool.

What is required then is a means to intertwine fixed parts of HTML pages with variable content taken from the database, that is, with objects currently stored in the system. We solve this problem by augmenting HTML pages (XHTML, actually) with special XML elements that are evaluated on the server side before a page is being delivered to a client. Each of these augmented pages, or *templates*, is bound to a class of the UML model and is subject to inheritance and refinement. The next sections provide more detail.

3.1 Templates

A template is a named HTML page that is bound to a class of the UML model. When requested by a client, the template is loaded, and any special XML elements that access the model or its instances are evaluated. Standard HTML elements and those elements to be evaluated are easily distinguished by XML namespaces. The results of the evaluation are inserted into the

```
<h1>Theses Available</h1>

<ul>
  <!-- 'forAll' traverses a collection of objects and   -->
  <!-- repeats the enclosed elements for each of them.  -->
  <!-- 't' denotes the namespace for template elements. -->

  <t:forAll expr="Thesis->select(author->isEmpty())">
    <li>
      <!-- 'self' holds the current iteration item (optinal). -->
      <!-- 'valueOf' evaluates an expression and prints the   -->
      <!-- result.   -->

      <t:valueOf expr="self.title"/>

      <!-- 'if' encloses a conditional block. -->

      <t:if expr="advisor->notEmpty()">
        (advised by <t:valueOf expr="advisor.givenName"/>
                    <t:valueOf expr="advisor.familyName"/>)
      </t:if>
    </li>
  </t:forAll>
</ul>
```

Fig. 5 Template incorporating OCL expressions

HTML page, and the page is finally delivered to the client. Figure 5 shows an excerpt from a template of the `Thesis` class. The template displays a list of all theses that are available, that is, those theses that don't already have have an author assigned. For each thesis, the title and a possible advisor are printed.

The template mechanism is similar in spirit to the embedding of a programming or database query language into HTML pages. Yet, there are also differences:

– Templates are not just a collection of HTML pages. They are properties of classes, loosely related to operations: templates are requested (or *invoked*) in the context of either their class or an instance of their class (similar to class and instance operations). The invocation context is stored in a variable `self` that can be accessed from inside the template.
– Instead of incorporating yet another query language, OCL is used as an expression language for all sorts of queries in templates. For example, `<t:valueOf expr="advisor.givenName"/>` outputs the `givenName` attribute of an object that is referenced by the `advisor` association of a `Thesis` (`self` is optional, as usual).

Additional XML elements provide a limited degree of what might be called control flow while generating an HTML page. As an example, there is an element that iterates the constituents of an `OclCollection` (the result of a `<class>.allInstances->select(<condition>)` expression) and outputs a certain HTML fragment for each. Another XML element provides an equivalent to the usual `if` construct known from programming languages. Its condition is an OCL expression that evaluates to a boolean value.

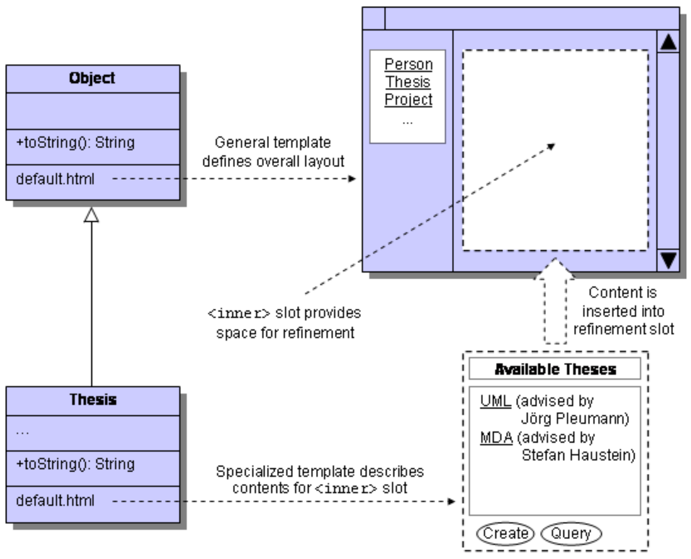

Object
+toString(): String
default.html
General template defines overall layout
Person
Thesis
Project
...
<inner> slot provides space for refinement
Content is inserted into refinement slot
Thesis
...
+toString(): String
default.html
Specialized template describes contents for <inner> slot
Available Theses
UML (advised by Jörg Pleumann)
MDA (advised by Stefan Haustein)
Create
Query

Fig. 6 Inheritance and refinement of templates

Several template names are reserved for specific purposes. For example, `edit.html` is used for editing the details of an object. The effect of providing templates with these reserved names is that the default pages for listing, displaying, editing, and searching objects are changed, while the overall structure of the system is still determined by the model. By providing additional templates, it is possible to build systems with a more complex navigational structure.

3.2 Inheritance and refinement

Being properties of classes, the templates propagate according to the inheritance rules dictated by the class hierarchy: subclasses inherit the templates defined by their superclasses, but can override them by providing a template of the same name. If no specific templates are defined at all, classes inherit their templates from `Object` – which results in the default output behaviour described in section 2.

Refinement of templates is supported by an XML element `<inner>` that behaves similar to the way refinement is handled in the BETA programming language [21]: each template can define a slot where it wants to be refined by specialized templates. When a specialized template that refines a more general one is processed by the MDR, the resulting HTML page contains the general template, with the `<inner>` element being replaced by the specialized template.

It should be noted that the BETA way of handling refinement is rather different from the `super` construct used in programming languages like Java or C++. Conceptually, `super` inserts inherited code into overriding code at a position where the call occurs, so specialized code “encloses” inherited code. BETA does it the other way around. For the purpose of dealing with HTML pages (or general XML documents, actually), the BETA approach is more appropriate, because it supports a more natural way of organizing templates: the most general template contains the usual outermost HTML elements (`<html>`, `<head>`, ...) and provides a general page layout (for example, a title and a menu bar). In particular, this template is already a complete and well-structured (X)HTML document. Specialized templates simply fill in their details into the appropriate refinement slot.

Figure 6 illustrates template refinement in the context of the university example. The `default.html` template for the `Object` class defines the general page layout for all classes in the system. This layout includes a fixed menu on the left side. By means of the `<inner>` element, the template reserves the dashed rectangle for refinement. Subclasses that override and refine `default.html` specify their additional information here. The `Thesis` class, for instance, adds the aforementioned list of available theses.

4 Implementation

We have implemented the above ideas in our Infolayer system. The roots of this Java-based system go back to the COMRIS [13] project funded by the European Community. Fig. 7 shows a very rough approximation to the architecture. At the heart of the system lies an implementation of selected portions of the UML metamodel:

– A core part, implementing basic properties of all model elements. This part has a loose conceptual relationship to the UML meta-metamodel, but it is not an implementation of the Meta Object Facility (MOF).
– A part that implements UML classes and objects, including all the necessary properties, such as attributes, associations, methods, inheritance, etc. In addition to the usual built-in datatypes, a primitive `File` type provides for easily storing binary files and thus supports a limited form of content management system (CMS) functionality.
– A part that implements UML state machines, including a runtime component that is able to simulate a state machine.
– An OCL parser and evaluator. This component implements the OCL as defined in the UML 1.5 specification. As mentioned, a few additional constructs have been introduced to turn the OCL into a full UML action semantics surface language.
– An XMI loader that is used to feed the model into the system. Since XMI operates on the UML meta-metamodel, this part requires the core part only and handles arbitrary metamodel elements through the reflective capabilities introduced in the core part.
– A part that implements an XML-based persistence mechanism as well as other ones, for example one based on Java Database Connectivity

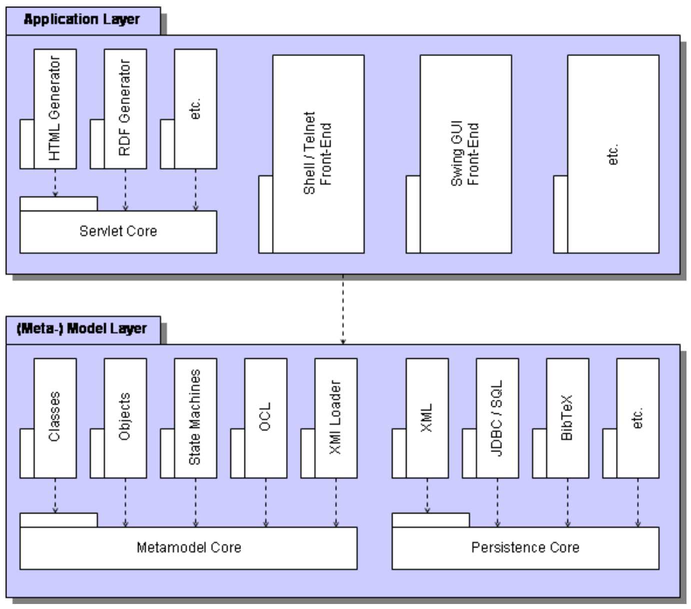
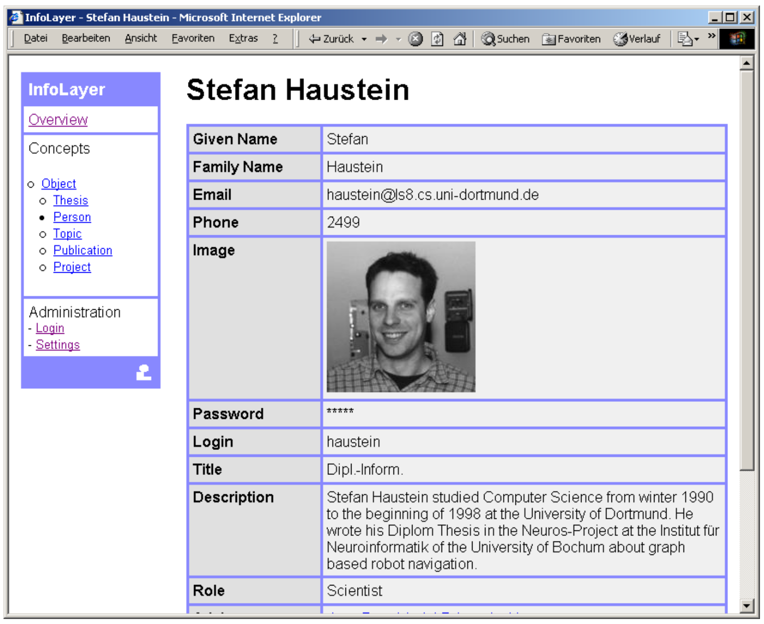


Fig. 7 Architecture of the Infolayer

(JDBC) that makes it possible to integrate an existing relational database into the Infolayer. Actually, it is even possible to have the Infolayer operate on a BibTeX file – in that case a model is automatically provided that represents the various BibTeX entry types.

All these components contribute to the system's model layer or backend, which is still largely application-independent. Atop the model layer lie different application front-ends which provide the functional equivalent to transformations from a PIM to a PSM in MDA, among them the servlet already mentioned in the previous sections. The servlet uses the template mechanism to generate output based on the model and instance information. HTML is one possible output format, but it is not the only one. The template processor can actually be used for generating output in any XML-based target language. As an example, an approach to produce output conforming to the Resource Description Framework (RDF) and thus open the Infolayer to the Semantic Web is described in [15]. Further template sets could easily be used to address mobile phones using the Wireless Markup Language (WML) or a limited subset of HTML.

Fig. 8 shows the default HTML user interface for the university example. This is an actual screenshot from an Infolayer installation working on the class diagram depicted in Fig. 1 when no specific templates are used. Fig. 9 shows the improved user interface after templates have been added to the system. The templates make use of the refinement mechanism, that is, they


InfoLayer - Stefan Haustein - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück
Suchen
Favoriten
Verlauf

InfoLayer

Overview

Concepts

- Object
  - Thesis
  - Person
  - Topic
  - Publication
  - Project

Administration
- Login
- Settings

Stefan Haustein

Given Name	Stefan
Family Name	Haustein
Email	haustein@ls8.cs.uni-dortmund.de
Phone	2499
Image	
Password	*****
Login	haustein
Title	Dipl.-Inform.
Description	Stefan Haustein studied Computer Science from winter 1990 to the beginning of 1998 at the University of Dortmund. He wrote his Diplom Thesis in the Neuros-Project at the Institut für Neuroinformatik of the University of Bochum about graph based robot navigation.
Role	Scientist

Fig. 8 User interface without templates

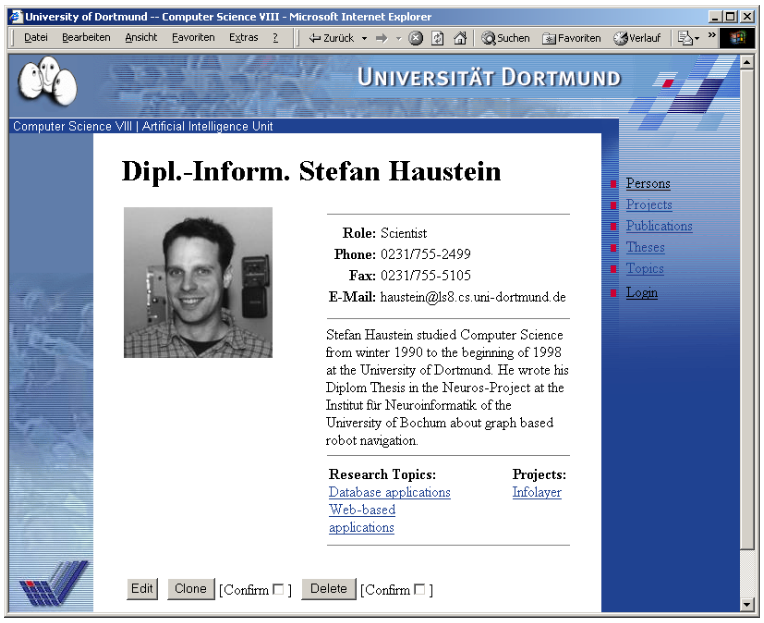
all inherit the main layout (the darker frame parts of the page) from `Object` and fill in their details into an `<inner>` slot (the white rectangle).

Given a Web browser that is capable of rendering the Scalable Vector Graphics (SVG) format, even an object's state machine can be displayed in the user interface, employing the visual information taken from the UML model exported by the CASE tool.

Several other front-ends exist in addition to the servlet. A command-line front-end can be used to access the system using OCL expressions only, and a similar front-end uses the Telnet protocol to access an Infolayer installation from a remote host. A Swing-based graphical application front-end that adapts its user interface to the model has also been implemented, but is currently not maintained, because the focus lies on Web-based access. Finally, a front-end implementing the Simple Object Access Protocol (SOAP) make it possible to remotely access the Infolayer from third-party applications. Using SOAP, objects can be created, updated, or deleted, and publicly visible methods can be called, as long as security settings permit this.

5 Applications

In the spirit of "eating one's own food", that is, the idea that a (software) engineer should always be the first one to apply his or her own systems, the Infolayer is being actively used in a number of different projects.


University of Dortmund -- Computer Science VIII - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Suchen Favoriten Verlauf
UNIVERSITÄT DORTMUND
Computer Science VIII | Artificial Intelligence Unit
Persons
Projects
Publications
Theses
Topics
Login
Edit
Clone
[Confirm]
Delete
[Confirm]

Dipl.-Inform. Stefan Haustein

Role: Scientist
Phone: 0231/755-2499
Fax: 0231/755-5105
E-Mail: haustein@ls8.cs.uni-dortmund.de

Stefan Haustein studied Computer Science from winter 1990 to the beginning of 1998 at the University of Dortmund. He wrote his Diplom Thesis in the Neuros-Project at the Institut für Neuroinformatik of the University of Bochum about graph based robot navigation.

Research Topics: Database applications, Web-based applications
Projects: Infolayer

Fig. 9 User interface with templates

The largest of these applications is probably the Web site for MuSofT (`http://www.musoft.org`), a distributed project that develops multimedia teaching material for software engineering education in Germany [7]. The goal of this application is to manage and distribute the material contributed by the various project partners and to facilitate sustainable (re-) use inside and outside this community. The corresponding database is rather complex: it not only features authors, their (binary) material and access rights, but also metadata conforming both to the Learning Objects Metadata (LOM) standard [5] and a subset of the ACM computing classification system [8] for proper structuring and efficient retrieval within the database. The MuSofT application is described as a detailed case study in the appendix.

A second application is the Java 2 Micro Edition (J2ME) Device Database (`http://devicedb.kobjects.org`), a database of mobile phones and personal digital assistants (PDAs) supporting J2ME. Although these devices follow a common standard, they all have their own bugs, peculiarities and limitations – which is critical information from the point of view of a developer who, naturally, cannot own all the devices available in the world. The database accumulates this information. It receives "live" data from a small benchmarking application that is publicly available and can be run on the different devices by their owners, sending its results to the Infolayer database after it has been executed.

Other applications include the Machine Learning Net (MLnet) teaching server (`http://kiew.cs.uni-dortmund.de:8001`), a system that provides information for the artificial intelligence community, and several chairs of the University and the University of Applied Sciences in Dortmund who use it to manage their Web sites. In addition to these, the Infolayer is used in numerous smaller projects where a database with Web-based front-end and an easy-to-use navigational structure is required without spending much effort on its implementation.

6 Lessions learned

Some of the above applications have been in use for about three years. Experiences with these applications and with the overall Infolayer approach have shown a number of things. For this paper, we would like to concentrate on the areas of feasibility, methodology, OCL usage, and tool support.

6.1 Feasibility

First, 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 section, 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.

6.2 Methodology

Second, 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.
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. Since changes to the templates are recognized by the running system without a restart being necessary, they can be applied and evaluated easily and with near-zero turnaround time.
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.

6.3 OCL usage

Third, the decision to use OCL both as a query language and as the basis for the action language inside the system proved to be very helpful. A previous version of the Infolayer used the Object Query Language (OQL) [3] instead, so the authors 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. Also, OCL is tightly integrated with the various UML diagrams, in particular class diagrams. This results in a more coherent approach and the possibility to leverage existing knowledge. After all, if someone has prior experience with UML, he or she is likely to have run into OCL, too.

As a consequence, we propose to use OCL as a general (side-effect free) query language for OODBMS. As mentioned in section 2, a limited number of additional constructs suffices to turn OCL into a full action semantics language, thus even providing the functionality required for arbitrary database modifications.

6.4 Tool support

Fourth, it has become clear that contemporary CASE tools have a number of limitations when used for modeling Infolayer applications.

– While most CASE tools have excellent support for editing the graphical parts of a UML model, the incorporation of non-graphical information, in particular OCL constraints, is rather weak. As a result, Infolayer applications that make heavy use of OCL or the OCL-based action language are sometimes difficult to develop and maintain. OCL information is distributed over tagged values or general comment fields, depending on what is supported better by a particular CASE tool. Some tools provide a limited form of OCL syntax checking, but are far from the features of typical programming language editors. Hence, small errors in OCL constraints often only show up at runtime and then need a correction of the model and a restart of the whole application.

– XMI support is another weak point in many existing CASE tools. Different XMI variants are used in different tools, and often it is difficult if not impossible to load a UML model generated by one tool into another tool. This multitude of XMI variants naturally also affects the Infolayer, which, in the ideal case, should load any well-formed XMI file. The XMI loader used inside the system tries to detect and accept as many XMI variants as possible, but the incorporation of special cases required for even the handful of tools regularly used by the authors has made its implementation rather complex.

Both problems are not limited to the Infolayer approach, though. We expect the importance of OCL and action languages to grow with the acceptance of model-driven development in general. Something similar holds for other forms of annotation that are required for directing model transformations or code generation in MDA. CASE tools will hopefully reflect this by better support for editing and managing non-graphical parts of a model. While dedicated syntax support for arbitrary action languages might not be feasible for CASE tool vendors, support for an action semantics surface language based largely on OCL would only be a small step from supporting OCL itself. XMI support, too, has to mature, because interoperability of tools is critical to any form of model-driven development. The XMI 2.0 specification seems to head into the right direction here, because it supports simpler and more compact descriptions of UML models, which will hopefully result in more robust implementations.

7 Related work

The Infolayer is not the first system that is based on the idea of interpreting and executing a graphically specified formal model. Harel's Statemate tool [12] uses state machines to describe and simulate system behavior. The Real-Time Object-Oriented Modeling (ROOM) [27] language specifies both system structure and behavior using a mixture of classes and state machines. Both systems are targeted at generating code for embedded systems, not for database systems or Web applications. As a result, user interface considerations are not a concern for them.

Riehle [24], Mellor [22] and Frankel [9] mention UML virtual machines (VM) capable of interpreting arbitrary UML models. Yet, these VMs seem to focus on simulating and model checking a given model or use it for research purposes in areas like refactoring [17]. As far as we can see, most of them are not targeted at a certain application domain, as is the Infolayer, yet the systems are very close in spirit to ours.

A commonplace alternative to the UML/OCL/HTML combination used in our approach is a pure XML approach as mandated by the W3C. This approach would use XML Schema [31] for structural modeling, XPath [28] as a query language, and XSLT stylesheets [29] for transforming XML documents to HTML output renderable in a browser. Unfortunately, it can be shown that XML Schema does not allow to model arbitrary associations between concepts without information loss [19]. Hierarchical relationships in XML Schema cannot express cyclic associations such as the *Person–Publication–Project–Person* cycle in our university scenario. The identifier-based references between elements of an XML Schema are unable to ensure a mutual reference between two instances of these elements that take part in a bidirectional association: if a person is linked from a project, the reverse link is not necessarily set, too. Even if XML Schema were extended to cover those cases, it would not seem adequate to use a language primarily designed to describe document tree structures for more general graphs, such as our domain model. Another problem is that the XPath language – used for expressions inside XSLT stylesheets – operates on XML instances only, but igores valuable information that is available in the corresponding XML Schema. XML queries are evaluated against XML documents based on element names and nesting structure only, ignoring the data types and other information inherent in the original model [20].

There's already a number of methodologies trying to employ the UML for creating Web applications. Conallen [6] uses UML stereotypes to model various aspects of a Web application, from client and server components to details of individual HTML pages. While this approach pays off for very large applications that incorporate traditional executable code development, we find it overly complex for the average small or medium Web application. Baumeister et al. [1] describe a systematic design method for Web applications combining ideas from the Object-Oriented Hypermedia Design Method (OOHDM) [26] and UML. The system specification is divided into a conceptual model, which is roughly equal to our domain model, and a navigational model. Since our premise is that the system's navigational structure is close or equal to the entity structure anyway, we do not see the necessity for the navigational model, and the examples given in [1] seem to concede more to our point than to theirs. WebML [4] is a high-level specification language for data-intensive Web applications. It seems closest to our approach in that it focuses on an entity-relationship model, that is, basically on a subset of UML class diagrams specifying the domain classes of interest. One then composes Web pages from these entities and high-level components like buttons, indexes etc.

Interestingly, all three approaches try to model HTML page layout by means of UML and find equivalents to single HTML elements using stereotypes. In our opinion this is putting the cart before the horse. The purpose of a Web application is determined by its content, that is, the entities it deals with, so these should come first and be central to the whole development process. Also, HTML is just one of many possible user interfaces for accessing the application. For HTML-centric approaches, incorporating alternative user interfaces or handling changes to the entity structure will be complicated. For a model-based approach – be it transformational or interpretative – this is easily done by either adding a new transformation or implementing a variant of the runtime environment.

Schattkowsky and Lohmann [25] describe a use-case-based development process for dynamic Web sites. While their work has a different emphasis in some areas, their premises are strikingly similar to ours. In particular, they target the special needs database-powered Web applications with limited business logic. Yet, their ProGUM system generates PHP and HTML code from UML models and thus is likely to run into the MDA problems mentioned in section 1, particularly those with maintaining generated code. The same holds for the UWE tool by Kraus and Koch [18] that generates XSLT stylesheets from a UML model. We hope to be more flexible with our interpretative approach.

8 Summary and Outlook

We have presented a novel approach to developing Web applications. It is highly model-driven, but instead of transforming a platform-independent model to a platform-specific one, as in the case of Model-Driven Architecture, it directly interprets or *executes* the domain model. For that purpose we have introduced the notion of a model-driven runtime environment that accepts a UML model consisting of a class diagram and a number of state machines and makes this model accessible through a servlet. A default HTML user interface is generated on-the-fly, but can be tailored to specific needs using a template mechanism. The template mechanism uses OCL plus a few additional constructs to access the domain model and the existing objects and renders them to HTML.

The idea has been implemented in the Infolayer system, which provides the basis for a number of different Web applications already in use. Experiences with developing and using these applications have been very positive so far, and we feel that it should be feasible to apply the same ideas to other application domains, too. Amongst the things we consider for future extensions to the system are additional UML diagram types and refactoring facilities.

The Infolayer system is available under the terms of the GNU General Public License (GPL) from `http://www.infolayer.org`.

Acknowledgements: The anonymous referees' comments have been very helpful to us. We want to thank the referees for their critical advice. We would also like to thank Perdita Stevens and Ernst-Erich Doberkat for their extensive feedback.

References

1. Hubert Baumeister, Nora Koch, and Luis Mandel. Towards a UML extension for hypermedia design. 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. 1999, Proceedings, volume 1723, pages 614–629. Springer, 1999.

2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley Longman, 1999.
3. Rick G. G. Cattell and Douglas K. Barry. *The Object Data Standard ODMG 3.0.* Morgan Kaufmann, 2000.
4. 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.
5. IEEE Learning Technology Standards Committee. Final draft of the IEEE standard for learning objects and metadata (LOM). http://ltsc.ieee.org/wg12, 2002.
6. Jim Conallen. *Building Web Applications with UML.* Addison Wesley Longman, 2000.
7. Ernst-Erich Doberkat and Gregor Engels. MuSofT – Multimedia in der SoftwareTechnik. *Informatik Forschung und Entwicklung*, 17(1):41–44, 2002.
8. Association for Computing Machinery. ACM computing classification system. http://www.acm.org/class, 1998.
9. David S. Frankel. *Model Driven Architecture – Applying MDA to Enterprise Computing.* OMG Press, 2003.
10. Object Management Group. Model driven architecture (MDA). http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, 2001.
11. Object Management Group. Unified modeling language (UML) 1.5 specification. http://www.omg.org/cgi-bin/doc?formal/03-03-01, 2003.
12. David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
13. 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.
14. 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.
15. 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.
16. Reiko Heckel and Marc Lohmann. Model-based development of web applications using graphical reaction rules. In M. Pezzè, editor, *Fundamental Approaches to Software Engineering*, pages 170–183. Springer, 2003.
17. Wai Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and Francois Pennaneac'h. UMLAUT – an extensible UML transformation framework, 1999. http://www.w3.org/TR/2002/CR-soap12-part2-20021219/.
18. 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.
19. Tobias Krumbein and Thomas Kudrass. Rule-based generation of XML schemas from UML class diagrams. In *Berliner XML Tage 2003*, pages 213–227, 2003.

20. 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.
21. Ole Lehrmann Madsen, Birger Möller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Addison Wesley, 1993.
22. Stephen J. Mellor and Marc Balcer. *Executable UML – A Foundation for Model-Driven Architecture.* Addison Wesley Longman, 2002.
23. Jörg Pleumann and Stefan Haustein. A model-driven runtime environment for web applications. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 – The Unified Modeling Language*, volume 2863 of *LNCS*, pages 190–204. Springer, 2003.
24. 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.
25. Tim Schattkoswki and Marc Lohmann. Rapid development of modular dynamic web sites using UML. In J.M.-Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002*, volume 2460 of *LNCS*, pages 336–350. Springer, 2002.
26. Daniel Schwabe, Gustavo Rossi, and Simone D. J. Barbosa. Systematic hypermedia application design with OOHDM. In *UK Conference on Hypertext*, pages 116–128, 1996.
27. Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley and Sons, 1994.
28. World Wide Web Consortium (W3C). XML path language (XPath) version 1.0, 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.
29. World Wide Web Consortium (W3C). XSL transformations (XSLT) version 1.0, 1999. http://www.w3.org/TR/1999/REC-xslt-19991116/.
30. Jos Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison Wesley, 1999.
31. World Wide Web Consortium (W3C). XML Schema part 1: Structures, 2001. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/.

Appendix: The *MuSofT* case study

MuSofT (Multimedia in Software Engineering) is a distributed project funded by the German Federal Ministry for Education and Research (BMBF) in which a number of German universities create multimedia content for software engineering education. The goal of the MuSofT Web site is to manage and distribute the material (so-called *learning objects*) contributed by the various partners and to facilitate sustainable (re-) use inside and outside this community. It was clear from the beginning that the corresponding database would be rather complex: it not only had to feature authors, their (binary) material and access rights, but also metadata conforming both to the Learning Objects Metadata (LOM) standard [5] and a subset of the ACM computing classification system [8] for proper structuring and efficient retrieval within the database. Since, apart from the complex entity structure, the application's business logic was rather straightforward – instances of the various entity classes had to be managed and made accessible via the Web –, the MuSofT project chose to implement its distribution site using the Infolayer.

8.1 Database

Figure 10 shows a slightly simplified version of the class diagram for the MuSofT application, which is basically the domain model that resulted from the project's analysis phase.

The central class `LearningObject` defines the properties that all sorts of learning objects share. The attribute `filename` stores the actual binary content of a learning object using the Infolayer's built-in type `File`. All other attributes and associations hold the learning object's metadata: some attributes, such as `title` and `description`, store arbitrary `String` values. Others, such as `difficulty` and `interactivity`, make use of enumeration types to ensure that only valid LOM values can be used. Associations to additional classes like `Language` are used when a metadata value has a complex structure itself, the multiplicity may be larger than one, or the set of possible values may evolve at run-time by adding new instances.

`LearningObject` itself is abstract and thus cannot be instantiated. The concrete descendants `LearningUnit`, `LearningModule`, `GroupObject`, and `MediaObject` have to be used instead. These classes extend `LearningObject` by capabilities to structure material into up to four levels of granularity.

UML initial values are used to set some attributes and associations to sensible defaults whenever a new learning object is created. They make use of the built-in `Infolayer` class that represents the run-time system itself. The initial value of `creationDate` is `Infolayer.getCurrentDate()`, that is, the current date and time. The default value of the `author` association is `Infolayer.getCurrentUser()`, which returns the currently logged in user (who also created the object).

The `User` class serves as a basis for access management. `LearningObject` has a total of three associations to this class. One specifies the original author and owner of a learning object, which should be a single person. A number of additional users may be specified as contributors. Author and contributors form the set of people that are allowed to modify the learning object. Furthermore, users that apply a learning object in their classes may express their interest in being notified whenever the object changes. As mentioned in section 2.1, the Infolayer consults the instances of the `User` class whenever a user attempts to log in to the system.

`AcmClassifier`, finally, represents the nodes of the ACM classification system. Each node has attributes that store its classifier and title information. A parent/child association from the class to itself is used to organize the nodes into a proper tree.

All classes in the model implement the `toString()` method to derive a short, printable description of each instance. In the case of the `Language`

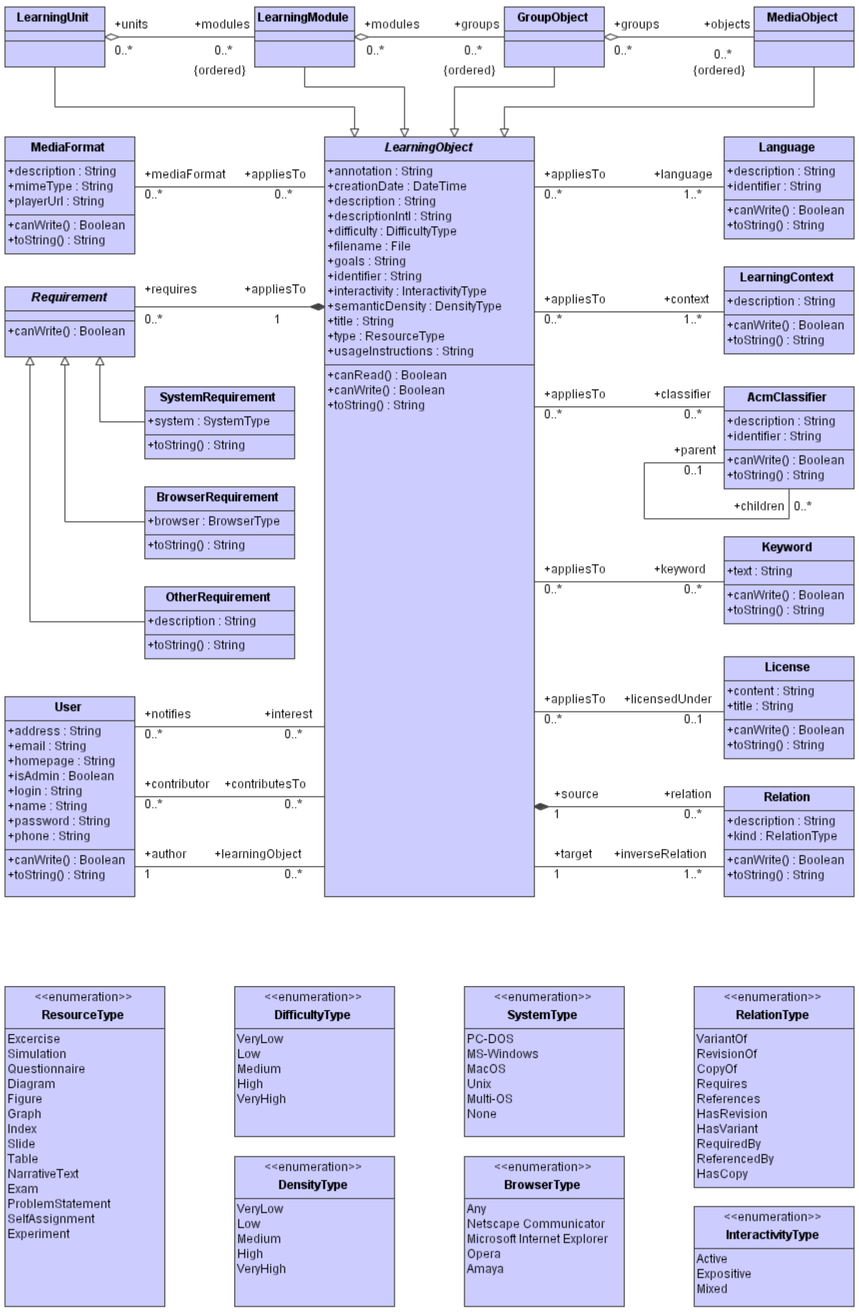


Fig. 10 UML class diagram for the MuSofT application

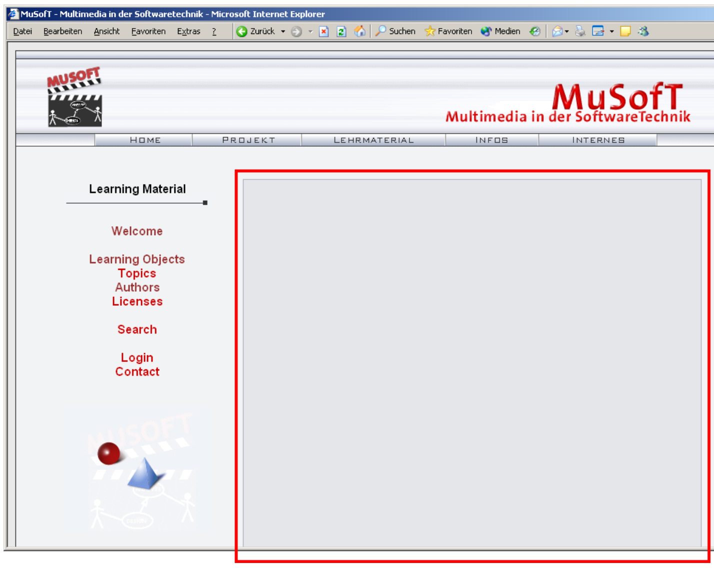

MuSofT - Multimedia in der Softwaretechnik - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Suchen Favoriten Medien
MUSOFT
MuSofT
Multimedia in der SoftwareTechnik
HOME
PROJEKT
LEHRMATERIAL
INFOS
INTERNES
Learning Material
Welcome
Learning Objects
Topics
Authors
Licenses
Search
Login
Contact

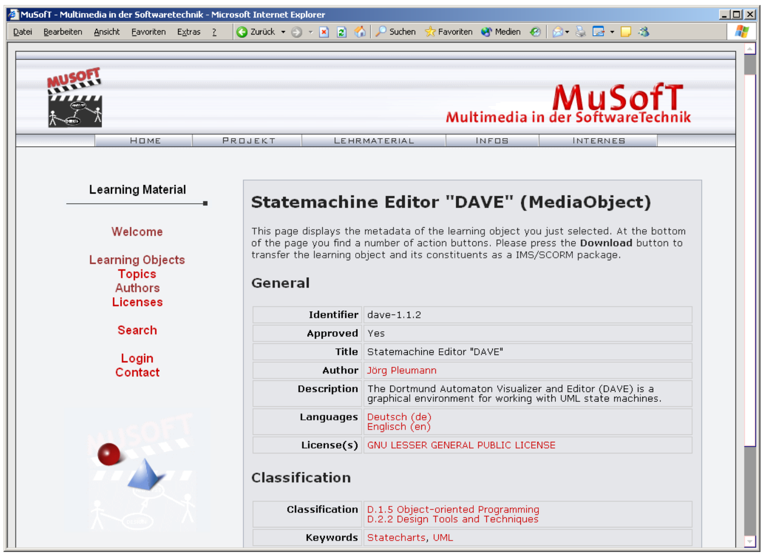
Fig. 11 Frame page with space reserved for refinement

class, for example, the method concatenates the value of `description` and `identifier`, the latter being in parentheses. Most classes also implement the `canRead()` and `canWrite()` methods, which are invoked whenever access rights for an object need to be checked. The implementation of `LearningObject.canWrite()`, for example, uses the expression `author->union(contributor)->contains(Infolayer.getCurrentUser())`, allowing the original author and all contributors to modify the object – just the desired behavior described above. Most other classes implement `canWrite()` based on the OCL expression `Infolayer.getCurrentUser()->notEmpty() and Infolayer.getCurrentUser()->isAdmin` to make sure only administrative users are allowed to create, modify, or delete instances.

8.2 User Interface

Up to this point, more or less standard UML features have been used. The creation of HTML output was not a concern. Thus, with the exception of possibly the built-in `Infolayer` class and the two operations used for deriving initial values, the above model should be roughly on the level of a PIM. Yet, the model already suffices for driving the Infolayer and getting a working prototype of the system. This prototype looks rather minimalistic (similar to Fig. 8), but it makes it possible to fill the database with content.

To improve both the layout and the navigational structure of the application, a number of XML templates were provided to the system along the lines of section 2.2. First of all, a prototypical plain HTML page was created


MuSofT - Multimedia in der Softwaretechnik - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Suchen Favoriten Medien
MuSofT
Multimedia in der SoftwareTechnik
HOME PROJEKT LEHRMATERIAL INFOS INTERNES
Learning Material
Welcome
Learning Objects
Topics
Authors
Licenses
Search
Login
Contact

Statemachine Editor "DAVE" (MediaObject)

This page displays the metadata of the learning object you just selected. At the bottom of the page you find a number of action buttons. Please press the **Download** button to transfer the learning object and its constituents as a IMS/SCORM package.

General

Identifier	dave-1.1.2
Approved	Yes
Title	Statemachine Editor "DAVE"
Author	Jörg Pleumann
Description	The Dortmund Automaton Visualizer and Editor (DAVE) is a graphical environment for working with UML state machines.
Languages	Deutsch (de) Englisch (en)
License(s)	GNU LESSER GENERAL PUBLIC LICENSE

Classification

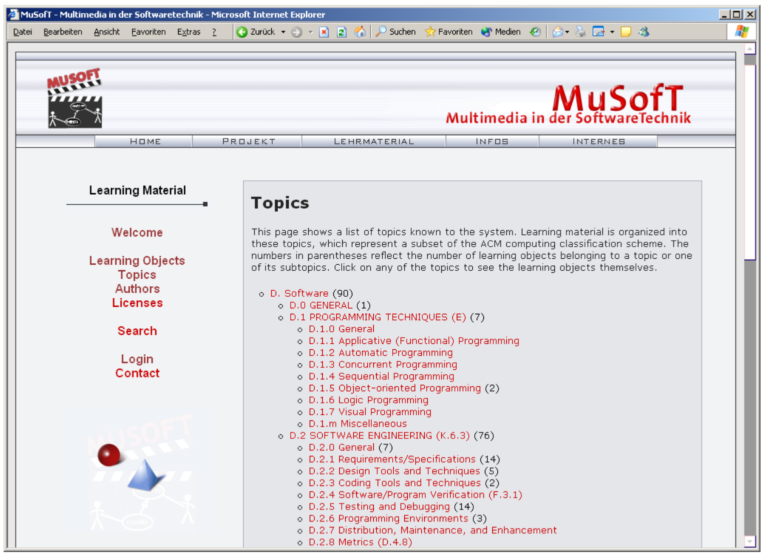
Classification	D.1.5 Object-oriented Programming D.2.2 Design Tools and Techniques
Keywords	Statecharts, UML

Fig. 12 HTML page for details of a learning object

by a Web designer. This page, which is depicted in figure 11, contains the general layout of the application as well as some HTML elements that are common to all pages (for example, the topmost menu with links to a number of external pages that are not handled by the Infolayer). The page was named `frame.html`, and it was attached as a template to the basic class `Object`, so that it became available to all other (sub-) classes in the system.

The highlighted area of the frame template was reserved for content determined by the various classes of the model and their instances using the XML element `<inner>` at the corresponding position of the HTML file. An example of a specialized page is shown in figure 12. This page displays the metadata of a learning object.

For each class in the model, four additional templates for listing, displaying, editing, and querying instances were defined. Each filled the gap provided by the `<inner>` element of the frame template. While it would have been possible to use only four templates attached to `Object` for all classes in the system – which is the Infolayer's default behavior –, this was not appropriate for MuSofT, since the templates were rather different: instructions on how to use each page had to be added to the templates, because the end users could not be expected to be experts in using the system. Some templates divided the rather long list of attributes and association ends into several groups separated by sub-headings. The list of available learning objects not only displayed `toString()` values for each object (which is the only detail the `Object` class would have known). They also incorporated attributes of a class, as in the case of the author list, where the number of learning objects was printed next to the name. Last, but not least, one



Topics

This page shows a list of topics known to the system. Learning material is organized into these topics, which represent a subset of the ACM computing classification scheme. The numbers in parentheses reflect the number of learning objects belonging to a topic or one of its subtopics. Click on any of the topics to see the learning objects themselves.

- D. Software (90)
  - D.0 GENERAL (1)
  - D.1 PROGRAMMING TECHNIQUES (E) (7)
    - D.1.0 General
    - D.1.1 Applicative (Functional) Programming
    - D.1.2 Automatic Programming
    - D.1.3 Concurrent Programming
    - D.1.4 Sequential Programming
    - D.1.5 Object-oriented Programming (2)
    - D.1.6 Logic Programming
    - D.1.7 Visual Programming
    - D.1.m Miscellaneous
  - D.2 SOFTWARE ENGINEERING (K.6.3) (76)
    - D.2.0 General (7)
    - D.2.1 Requirements/Specifications (14)
    - D.2.2 Design Tools and Techniques (5)
    - D.2.3 Coding Tools and Techniques (2)
    - D.2.4 Software/Program Verification (F.3.1)
    - D.2.5 Testing and Debugging (14)
    - D.2.6 Programming Environments (3)
    - D.2.7 Distribution, Maintenance, and Enhancement
    - D.2.8 Metrics (D.4.8)

Fig. 13 HTML page for ACM classification scheme

class did not show a flat list of instances at all: for `AcmClassifier`, a tree of instances was desired to better reflect the hierarchical nature of the classification system. Figure 13 shows this HTML page.

8.3 Business Logic

The MuSofT application makes limited use of statemachines. A very simple statemachine with states like *new*, *in review*, *approved*, or *deprecated* is used in the `LearningObject` class to describe the editorial workflow currently adopted in the project. Only approved objects are visible to external users. This workflow might become more complex in the future, depending on the experiences with the working system.

8.4 Additional Features

Most functionality of the MuSofT application was easily implemented by means of UML/OCL or the XML template mechanism. Only two features required additional Java programming:

– E-Mail notifications. Users may register for change notifications on learning objects. The author of a learning object decides whether a notification is actually sent when he or she changes the object.
– Standardized export. Users may download a number of interconnected learning objects in a single, standardized package format, basically a

ZIP file that contains both the binary content and the metadata of the exported objects.

Both features were implemented in the form of specialized XML template elements that add a button to generated HTML pages. Pressing that button results in the corresponding Java code being invoked. Since the Infolayer's core part is basically an implementation of the UML metamodel, accessing the model or its instances from Java is straightforward.

The MuSofT application also makes use of the Infolayer's SOAP capabilities: an external tool supports authors in uploading a number of interconnected learning objects along with their metadata to the system, bypassing the HTML user interface. Since the Infolayer is able to generate an XML schema corresponding to the UML model, the syntactical correctness of the uploaded file can be checked on the client side before actually executing the (possibly time-consuming) upload process.

8.5 Experiences

A first prototype of the MuSofT application was developed in December 2001. This prototype used a much simpler domain model than the one depicted in Fig. 10, but the important parts of the metadata, such as the ACM tree and the corresponding search facilities, were already included. A small number of templates were added to the system to establish a minimal corporate design and to give the other project members an impression of what the final system might look like. The prototype also featured the standardized export format, since this was considered a necessary feature, too. In only two weeks, a working version of the system was developed that included all the critical features, though much of it was still in its infancy.

After the Infolayer-based approach to the distribution site had been approved by the MuSofT project members, the system was successively improved during 2002. The domain model was refined. More templates were added to the system, including the layout provided by the 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/OCL. Over time, several new features were incorporated, such as the e-mail notifications, download counters, graphical reports on the database content, and the SOAP upload tool. Though still in development, the MuSofT application was already being actively used by the project members for uploading their learning material, so the developers were able to incorporate feedback early.

The application was made publicly available in the fall of 2002. Since then, about 100 different learning objects on software engineering along with their metadata have been stored on the server. These learning objects range in complexity and size from simple slides over specialized applications to complete videos of several hundred megabytes.