MDA Approach for Maintenance of Business Applications

Mila Keren[1], Andrei Kirshin[1], Julia Rubin[1], and Ahto Truu[2]

[1] IBM Haifa Research Lab, Mount Carmel, Haifa, Israel
{keren, kirshin, mjulia}@il.ibm.com
[2] WM-data, Tartu, Estonia
{ahto.truu@wmdata.ee}

Abstract. We present a case study that utilizes UML modeling methodology for typical business applications. Such applications generally contain a GUI front-end for manipulating database tables and are object-relational systems that deal with both relational databases and object-oriented technology. To model such applications, we use UML Profiles and metamodels based on a three-tiered application architecture for the different stages of the development lifecycles. The benefits of the model-driven approach include the possible use of the models for maintenance processes such as incremental code generation, updating test cases, and documentation. These models also enable developers to validate the application's flow by simulating its behavior through model execution.

1 Introduction

Today, more and more industry domains are beginning to understand the benefits of model-driven development for various products. The Model-Driven Architecture (MDA) was proposed by OMG as a radical move from object design to model transformation. The Unified Modeling Language (UML) was elected to play a key role in this architecture, being a general purpose modeling language. But being created for designing Object Oriented (OO) software applications, it often lacks the elements required to represent specific domains concepts. The solution proposed by OMG was to create profiles for certain application domains.

Our work focuses on a case study based on a UML profile and metamodel for business applications with a layered architecture. Our main goal is to support the application maintenance process through this modeling. The maintenance process includes requirements management for the customers changing needs and the subsequent updates of code and test procedures. At present, this is often done manually when changes in design and code documentation are made in an informal manner, where they are sometimes not even written down. In this situation it becomes difficult to maintain consistency between the requirement updates, code versions, and test sets. Making updates in the modelling environment with the possibility of automatically generated code and test cases may increase the efficiency of the maintenance process. In this way, the application model maintains the connectivity between the requirements, design elements, and the generated results.

A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 40 – 51, 2006.
© Springer-Verlag Berlin Heidelberg 2006

The paper is organized as follows. Section 2 provides an overview of the architecture for business applications and modeling practices. Section 3 describes the proposed metamodels and UML profiles for business application modeling. Section 4 describes an example of such an application model. Section 5 describes how the proposed model can be used for application maintenance. Finally, Section 6 contains conclusions and possible future directions.

2 Modeling of Business Applications

2.1 Business Application Architecture

"A Business, *an enterprise*, is a complex system that has a specific purpose or goal. All functions of the business interact to achieve this goal." [2]. Most business applications are designed to connect between end-user activities and different kinds of data repositories.

At the end of 90s there were many studies that looked for mapping solutions between Object Oriented (OO) and Relational Database (RDB) technologies [4]. The conceptual differences between RDB technology and OO technology make it difficult for the various parts of business applications to interact. To reduce the interaction and integration problems, the Business Object (BO) model was introduced for developing object-relational systems [1, 5]. In a BO solution, an application is divided into three tiers: the presentation tier, the RDB tier, and the business logic tier (also called the domain logic tier [1]). A typical three-tiered business architecture is characterized by loose coupling between the user interface and data repositories. The user interface has no direct interaction with the database, but instead, it interacts with the domain objects responsible for communication with the RDB. This separation allows each of the tiers to be developed independently, with a compact interface.

The upper, presentation layer handles the interaction between the user and the software implementing the business logic. It may be a rich client graphical user interface (GUI) application or the latest web-based clients. The software development practice of such applications is usually based on the Model-View-Controller paradigm (MVC) [1].

2.2 State-of-the-Art

Business application modeling took off in the late nineties with the increased usage of computer-aided tools. Since then, UML modeling has become a part of the development practices both for business process modeling and for system design [2, 5], together with the wider practice of EJB and J2EE patterns [3].

For modeling user interfaces, most of the existing work makes an effort to provide a means for modeling logical (abstract) views as well as concrete (physical) ones. Good modeling practices dictate that no interface details should be included in the early stages. The popular approach for designing GUI applications is based on the Model-View-Control (MVC) paradigm [1], which proposes separating between the view layout and its pieces of business logic. D.Anderson [6] describes this approach in a series of three papers; he also proposed using State Chart diagrams for

interaction design. J.Conallen [8] proposed using UML for Web application modeling. He also suggested integrating UI development with UML and Rational Unified Process (RUP), with its iterative validation and testing. More papers define UML profiles for modeling Web applications [10, 11, 12]. They use stereotypes such as client page, navigation, Java script, applet, and so forth to express the web specifics. Other works concentrate on concrete view presentation design [7, 10, 13]. These works, which define the methodology of the UML modeling for GUI and Web applications can be useful for business application modeling. However, their approach deals primarily with navigation between views or layout presentation details and less on the composition of business objects, along with their structure and relations, derived from the customer requirements.

The use of UML for modeling relational database systems started later than modeling for software. The main rules of mapping OO to RDB are described in [1], together with detailed design patterns for enterprise systems (see also Sect. 3.2). In 2002, the UML profile for data modeling was proposed [5] and implemented as a Rational Rose add-in; it allows the generation of database scripts from Rose models and vice versa.

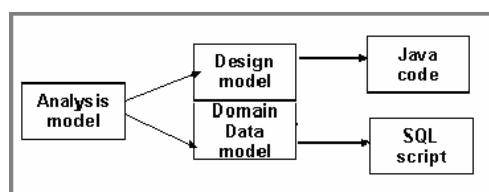
2.3 Current Development Process Pain Points

A significant problem with the existing development practices arises during application maintenance, when requirements may change and updates need to be made. Often the documentation is maintained in a text file or even on paper, and these are not kept up-to-date. Requirements changes are done informally and tests are both created and executed manually. This process can be improved using a model-driven approach, where the different artifacts, including documentation, code, and test cases, are automatically generated from the defined models.

3 UML Profiles and Metamodels for Business Applications

As noted above, there are several UML modeling directions for user interfaces and others for designing relational databases. Our intention is to combine them to provide a means for modeling the entire application as a business system, while paying attention to both its architecture tiers and the lifecycle phases of development.

To model business applications, we define metamodel and profiles implementing it while the described stereotypes follow the architecture described in Sect. 2.1. These profiles are used to create three models (see the figure on the right): analysis model, design model and domain data model. These models are tightly connected to present the same business logic. The model transformations can be applied to them to create design and domain elements from the higher level analysis model. However, the designer can also make changes later to each of these models so they include specific details of the implementation. These lower level models can be used directly



for generating code and SQL scripts or divided again into different levels to separate platform-specific features into additional models. In Sect. 4.2 on transformation and code generation, we describe how to combine manual and automated updates.

3.1 Analysis Model

The analysis model is a higher level abstraction of the application; it describes the customer view of the application, its requirements, use cases, the main flow, and the general structure of components and data objects. The goal of the analysis model is to help different stakeholders come to an understanding and agreement on how the application's high-level design corresponds to their requirements. The model can be used as basis for the generation of lower level models through transformations. The analysis model can also be used to create test cases for the views and behavior logic.

Figure 1 presents the classes of the analysis metamodel (a) and their relationships (b). The concrete classes are presented as stereotypes in the corresponding analysis profile. Requirement elements are included in this model, in keeping with the recently proposed SysML standard [14].

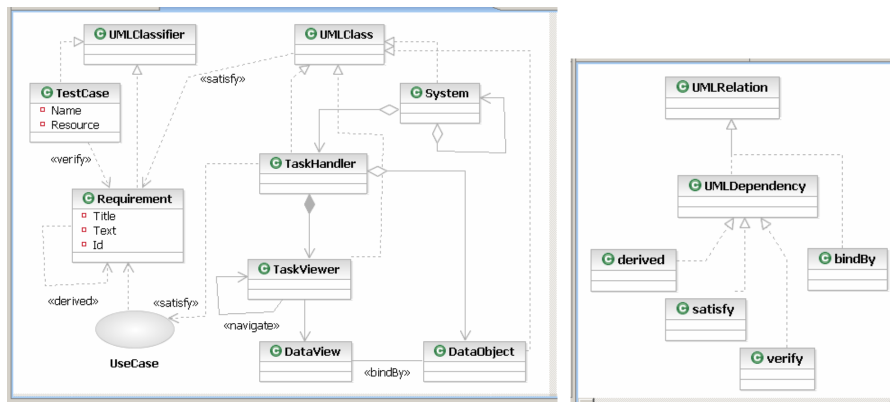


Fig. 1 (a, b). The metaclasses (a) and relations (b) of the analysis metamodel

The entire system is composed of smaller subsystems that contain data objects and handlers of tasks derived from the associated use cases. To model view aspects, we use stereotypes TaskViewer and DataView which are extensions of the general 'Viewer' metaclass in our metamodel. TaskViewer is analogous to a page or a window frame. One or several such viewers are associated with a class of stereotype TaskHandler; they are connected by naviation relashionships. In addition to view aspects, logic aspects are modeled by behavior diagrams that describe the control flow and navigation between viewers (see Sect. 4.2).

DataView stereotyped class is bound to DataObject class targeting to present its fields and actions. For business application modeling, we used two extensions of this metaclass: EditView presents a single instance of DataObject as a group of fields for editing its attributes; SelectView presents the same class as a set of instances in a table form. One or more of instances can be selected for the subsequent actions, like

editing, deleting, etc. Other specific views can be added by the designer for their particular needs.

3.2 Design Model

The design model represents the same business logics captured in the analysis model, but includes implementation details; it is used for code generation through one or more transformation procedures. Some elements of the design profile and metamodel are taken from the analysis metamodel, e.g., DataObject, EditView, SelectView; however, in the design model these elements have more implementation details, like methods with typed parameters. The design model can be very flexible and extended by designers for their specific implementations. Its goal is to keep the implementation with the right architectural approach using best practices patterns [3], and make it easy to understand and update at a later stage.

Following the MVC pattern mentioned above (see Sect. 2.1), class stereotypes are defined as view-related classes, and the associated classes of control and model stereotypes (Fig. 2). In the design model, view classes involve concrete view elements (associated by aggregation relation or as attributes), and operations corresponding to the user actions in the viewer, such as the Button or Menu actions stereotypes.

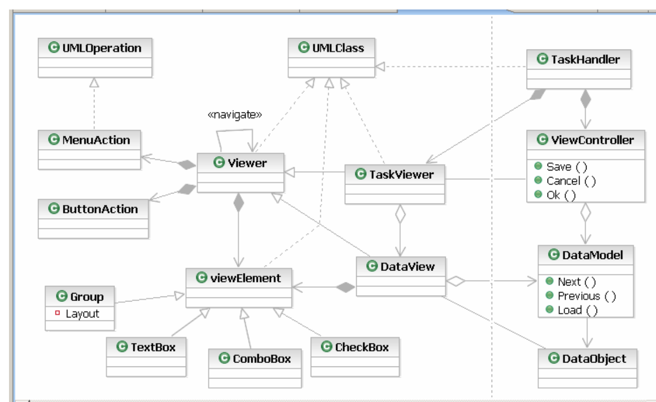


Fig. 2. The design metamodel

Elements stereotyped as 'Model' that are connectors to the data domain level, can be shared between views (Fig. 3, a) These elements are responsible for keeping a set of object instances retrieved from the relational tables by domain data model classes (see Sect. 3.3).

Navigation between views can be modeled by relationships with additional stereotypes (Fig. 3, b), where 'Page' navigation refers to the 'next-previous' navigation adopted by web pages. 'Child' creates a new view, keeping the old view for a later use when the child is closed. 'Tab' navigation is the notebook style where the user can switch views using several tabs.

Tagged values are additional attributes that are associated with the profile. For example, the stereotype 'Group' has attribute 'Layout' which is defined as a tagged

Fig. 3 (a, b). The hierarchy of view metaclasses (a) and navigation relations (b)

value set with two values, 'horizontal' and 'vertical'. Other tagged values can be added by designers who need to extend this profile.

3.3 Domain Data Model

The domain data model can be created by the transformation of Data Objects from either the analysis or design models. The main rules of mapping OO to RDB are briefly as follows [1, 4, 5]:

- Entities become tables
- Attributes become columns
- Identifiers become primary keys

For mapping object relations such as inheritance, aggregation, association into table relations, there are several patterns, like 'Embedded Foreign Key', 'Associated Lookup Table' and others [1]. If Data Objects can not be mapped into single tables having more complex structure, then classes with stereotype 'TableGateway' are used (Fig. 4) as table mappers.

The domain data model may then be reviewed and modified, or transformed into an additional platform specific model (PSM) that reflects specific implementation or performance requirements. The benefit of the domain data model is that it can be reused for implementation of the same business model on different database platforms.

Fig. 4. The domain data metamodel

4 Example of a Business Application and Its Model

4.1 Application Overview

Our case study considers a typical business application called Verification Office Application. The company manages testing and the certification of various measurement devices – scales, speeds and distance meters, thermometers, and so forth. It also handles packaged goods and the quality of service at medical labs. Company employyees need a comprehensive database system to manage the customers, their devices, test results, certificates and their validity, as well as billing and reporting. There are three kinds of application users: Regular worker, Secretary, and Administrator.

The application has a two-tiered architecture. The first tier contains the GUI front-end implemented in Java, designed following the MVC paradigm. Each use case has its handler class, with its own view and navigation between them supported by a specific mechanism. The second tier contains functions for database manipulations. These tiers communicate through the domain data objects included in the first tier.

Fig. 5 presents examples of use case (a) and class diagrams (b, c, d) related to the analysis model. Fig. 5, b shows the requirements associated with the corresponding

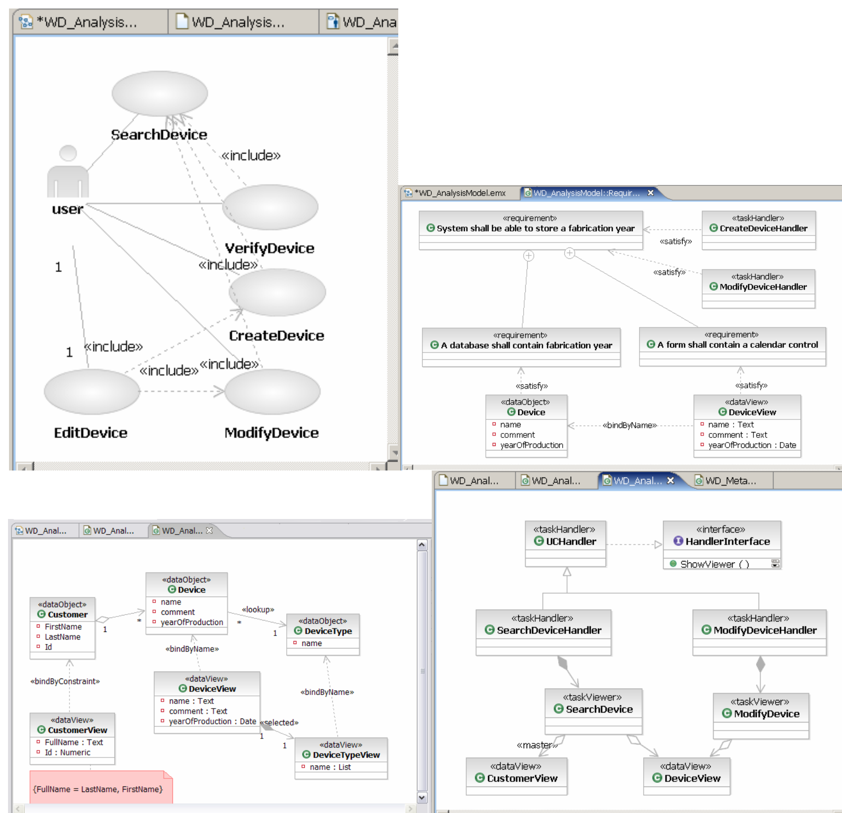


Fig. 5 (a, b, c, d). Examples of diagrams from the analysis model

design elements. Fig. 5, c shows relations between three data objects and their views. Fig. 4, d shows two use case handlers and the related view objects.

For behavior design we used UML activity diagrams. These activity diagrams (see Fig. 6) describe the task handlers' behavior as a specification of the detailed control flow, where the task handlers are responsible for fulfilling the functionality of the use cases. The following are examples of such diagrams where each lower level diagram contains details (Fig. 6, b) of an activity related to the higher level diagram (Fig. 6, a).

Fig. 7 presents an example of a class diagram from design model for the 'Create device' Task Handler, whose design follows the MVC pattern.

Fig. 8 shows an example of the domain model diagram presenting the relationship between the model and table elements.

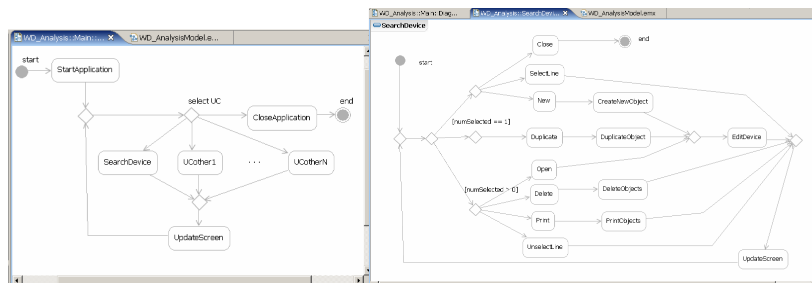


Fig. 6 (a, b). Examples of activity diagrams for a task handler from the analysis model

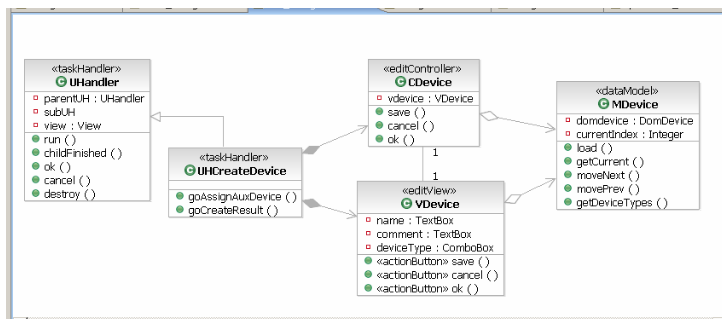


Fig. 7. Class diagram for a task handler from the application design model

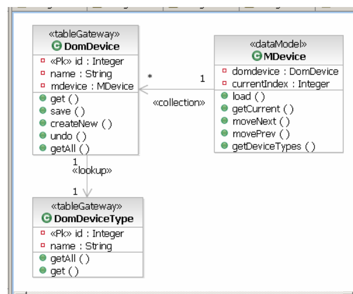


Fig. 8. The relationship between a model and table mappers

5 Using UML Modeling for Code Generation and Behavior Validation

Once the application models are created as a collection of structure and behavior elements, they can be used for several purposes for maintenance of the application. Our scenario for updating the application is taken from a real-life situation where some business object attributes are added or removed.

5.1 Model-to-Model Transformation

The reason for transforming a high-level model to a lower level model is to automate maintenance updates and to avoid manual changes whenever possible. The lower level model includes implementation-specific details of the application itself and of the middleware platform and hardware.

The transformation to domain data model is based on the OO-RDB mapping described in Sect. 2.3. The model contains table objects derived from the business objects and their relations. These table objects can be used directly in SQL scripts as single tables or grouped into more complex tables. In specific cases, additional lookup tables can be added to the model. This model can be derived from the data objects of either the analysis or design model.

In our study we use the specific mapping of the analysis model elements that reflects the existing application implementation design for the transformation to design model. The design model contains a set of classes related to the core of the framework, stereotyped as <<General>>; these classes are not changed during updates. Other stereotyped classes inherit from these core elements (see Fig. 9 a, b). This approach helps to support the code generation in a flexible way (see Sect. 4.2).

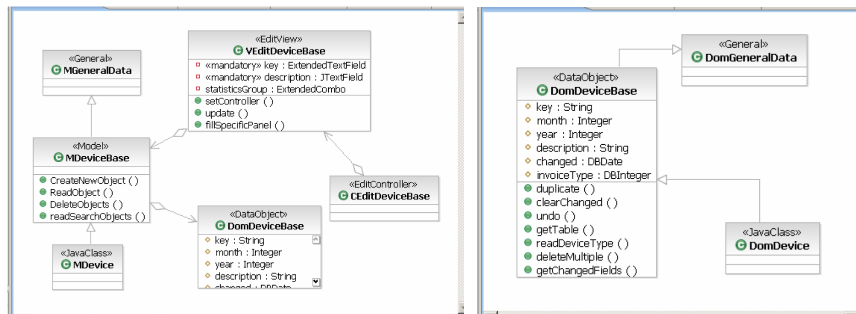


Fig. 9 (a, b). Class diagrams with stereotypes used for model-to-code transformation

The mapping of some of the profile elements is presented in Table 1. In addition to the stereotype mapping, a naming convention was used adding a prefix and a suffix to the original class names.

The model transformations were implemented using the Atlas Transformation Language (ATL) tool [15], which transforms an input model to another model using

Table 1. Part of the mapping between analysis and design model stereotypes

Analysis Model Stereotype	Design Model Stereotypes
<<dataObject>>	<<dataObject>>, <<model>>
<<editView>>	<<editView>>, <<editController>>
<<selectView>>	<<searchView>>, <<searchController>>
<<taskHandler>>	<<taskHandler>>,<<taskViewer>>, <<taskController>>

rules written in ATL and metamodels for both input and output models. In our study, the input and the output models have the same metamodel, which is the EMF/UML2 metamodel [16].

5.2 Model-to-Code Transformation

In our maintenance scenario the application code had already been implemented and some updates were carried out to the existing application design. To facilitate partial generation of the code, the design model needs to maintain consistency with the existing design and naming conventions.

During design model creation, each class that is a candidate for updates was divided into two: one of which is the target for code generation (named with a suffix) and the other is a subclass of the first for making manual code extensions and modifications. The stereotype "General" was used for the application classes that are in the stable part of the application and are not regenerated during model-to-code transformation.

In summary, we introduced three kinds of classes as input for the transformation: stable, automated and manual.

The code generation was implemented using the MOFScript tool [17]. The main transformation procedure was built from a set of libraries containing sub-procedures for specific stereotypes targeted for code generation, including for example: 'Data Object', 'Model', and 'Edit View'. The following is an example of this procedure:

```
uml.Class::mapClass() {
     var stereotype : string = ""
     stereotype = self.getStereotype();
     if (stereotype <> "General") {
            if (self.getStereotype() = data_stereotype )
                   self.dataMapClass()
            else if (self.getStereotype() = view_stereotype )
                   self.viewMapClass()
            else
                   self.standardMapClass()
     }}
```

The body of these procedures includes parts of the real code, which is usually cut-pasted with manual updates while eventual programmer's mistakes can introduce

bugs. Given class attributes and operations, the code is created by a loop procedure on each attribute as follows:

```
uml.Operation::generatePropertyLoopMethod () {
     self.standardMethodSignature()
     m_name = self.name
     names = self.owner.getPropertyNames()
     if (names != null) {
            names->forEach(n : String) {
                   a_name = n.firstToUpper()
            tab(1) a_name <%.%> m_name <%();%>
            nl(1) }}
     self.standardMethodEnd()
}
```

The following is the result of another generated method:

```
protected List getChangedFields () {
 // Generated body of getChangedFields method is here
       List var = new ArrayList();
       if (key.hasChanged()) var.add(key);
       if (month.hasChanged()) var.add(month);
       if (year.hasChanged()) var.add(year);
       if (description.hasChanged()) var.add(description);
       if (invoiceType.hasChanged()) var.add(invoiceType);
       return var;
}
```

The stereotype "JavaClass" was used for manual extensions, being a subclass of the automatically generated class. A template is generated for these classes; it includes only calls to the parent methods that are generated automatically. They may be used ‘as is’ or modified. This provides the flexibility for manually changing objects with specific needs. In addition, a library of utility procedures was created for the common use by other stereotype-specific libraries.

Analogous transformations may be created for other purposes, such as generating different kinds of documentation and test cases.

5.3 Behavior Simulation and Validation by Model Execution

The behavior diagrams presented in (3.3) can be used to validate the application logic. Presenting the logic in a graphical form enables developers to agree on the flow with analysts and stakeholders before starting the detailed design. A model execution tool was used to simulate and validate the specified behavior.

6 Conclusions

In this paper we describe the methodology and UML profiles for modeling business applications that access databases. The case study context was a business application with a specific maintenance scenario. This case study goal was to illustrate the benefits of modeling and code generation using a set of UML profiles. These benefits

include: traceability between high-level design changes and the corresponding implementation updates; code generation for periodic updates; and behavior validation using model simulation.

We also introduced a technique for creating model-to-code transformation that supports the generation of code for maintenance updates.

Our future activities will focus on enhancing the current metamodel and profile with additional semantics. We are investigating ways to improve traceability between customer requirements, model entities, and the application code to allow broader lifecycle support through model-driven development.

References

1. M. Fouler: "Patterns of Enterprise Application Architecture", Addison-Wesley (2003)
2. H.E. Eriksson, M. Penker: "Business Modeling with UML: Business Patterns at work", Wiley & Sons, Fall 1999.
3. D. Alur, et al.: "Core J2EE Patterns: Best Practices and Design Strategies", (2001)
4. A.M. Keller, R. Jensen, S. Agarwal: "Persistence Software: Bridging Object-Oriented Programming and Relational Databases", SIGMOD, May, 1993
5. "Mapping Object to Data Models with the UML", Rational Software Whitepapers, October, 2002.
6. CT Arrington: "Enterprise Java with UML", Willey & Sons, 2001
7. P. Coad, E. LeFebvre, J. DeLuca: "Java Modeling in Color with UML", PH 1999
8. J. Conallen: "Building Web Applications with UML". Addison-Wesley (1999).
9. D. Anderson: "Using MVC Pattern in Web Interactions", UIDesign.net,2000 [weblink: http://www.uidesign.net/Articles/Papers/UsingMVCPatterninWebInter.html]
10. R. Hennicker, N. Koch: "Modeling the User Interface of Web Applications with UML", Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists, 2001
11. N. Guell, D. Schwabe, P. Vilain: "Modeling Interactions and Navigation in Web Applications", Lecture Notes In Computer Science; Vol. 1921, Pages: 115 – 127, 2000
12. R.S De Giorgis, M.Joui: "Towards a UML Profile for Modeling WAP Applications", Journal of Computer Science and Technology, Vol.5(4), December,2005
13. K. Blankenhorn, Mario Jeckle: "A UML Profile for GUI Layout", Lecture Notes in Computer Science, Net.ObjectDays 2004: 110-121
14. SysML Specification, 2005, [weblink: http://www.sysml.org/]
15. ATL tool [weblink: http://www.sciences.univ-nantes.fr/lina/atl/]
16. J. Oldevik, et al.: "Toward Standardized Model to Text Transformations", proc. ECMDA-FA 2005, Nuremberg, Nov, 2005
17. EMF/Eclipse UML2 metamodel [weblink: http://dev.eclipse.org/viewcvs/ indextools.cgi/ org.eclipse.uml2/plugins/org.eclipse.uml2/model/UML2.ecore]