

GignoMDA - MDA Approach for Applications in the Database Domain

Sebastian Richly
Dresden University of Technology
Software Technology Group
sebastian.richly@inf.tu-dresden.de

Dirk Habich, Wolfgang Lehner
Dresden University of Technology
Database Technology Group
{dirk.habich,lehner}@inf.tu-dresden.de

Abstract

Database Systems are often used as persistent layer for applications. This implies that database schemas are generated out of transient programming class descriptions. The basic idea of the MDA approach generalizes this principle by providing a framework to generate program code (and database schemas) for different programming platforms. Within our GignoMDA-project [1], we extended the classic concept of the MDA. That means, our approach provides a single point of information describing all aspects of database applications (e.g. database schema, user interfaces, business logic and project documentation, ...) with a great potential of cross-layer optimization. In addition to the full automatic generation of an complete multi-tier database applications we implemented in the GignoMDA-project, our new cross-layer optimization hints are a novel way for the challenging global optimization issue for database domain-specific applications.

1 Introduction

Relational database systems are often used as persistent layer for a vast range of applications, especially for multi-tier applications. A huge number of such small to mid-size database applications require similar tools for data management activities which are often implemented for each project. On the one hand, writing such code is neither challenging nor free of errors. On the other hand, the global optimization of such applications is a very difficult task because required knowledge is normally hidden within the individual components of multi-tier applications.

The utilization of models has a long tradition in the software technology and is the standard procedure since the definition of UML. The *Model Driven Architecture* (MDA) approach [2, 3], coined by the Object Management Group, places the UML models in the mid-point of the development process of applications. One of the main goals of MDA is to separate software design from architecture and

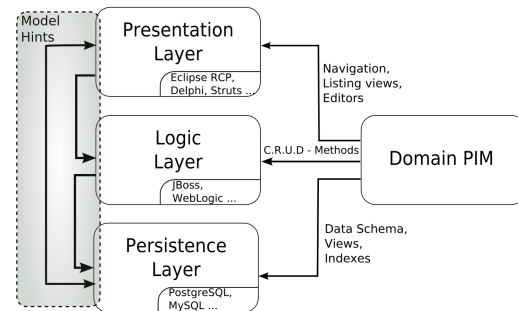


Figure 1. GignoMDA Approach

realization technologies allowing the design and architecture to be modified independently. The design addresses only the functional requirements while the architecture provides the infrastructure through which non-functional requirements like scalability, reliability and performance are realized. From the architecture independent UML model, various architecture specific models and program codes can be derived.

Our approach is based on the Model-Driven Architecture approach and extends classic methods for code generation through targeted control based on a central application specification. Not only does our GignoMDA approach enable a central and initially implementation- and architecture-invariant description of database applications, we also attempt to allow for an optimized implementation. In order to deal with the challenging global optimization issue of multi-tier database applications, we have extended the MDA concept for the description of functional dependencies and specification techniques for the implicit and explicit modeling of optimization hints as non-functional properties. These non-functional properties provide a novel way for cross-layer optimization steps, which is not possible within regular software development process.

To put it in a nutshell, we present our GignoMDA approach which is illustrated in Figure 1 by providing contributions in the following areas: (i) New domain ori-

ented annotations for UML models, so that these models are the mid-point of database application development addressing software design and optimization issues of multiple layers, (ii) explicit and implicit hints as the foundation for cross-layer-optimization techniques, especially for deriving physical database design decisions. These concepts are supported by our prototypical realization of a full automatic generation of a complete multi-tier database applications based on the AndromDA framework (<http://www.andromda.org>).

The rest of the paper is organized as follows: In Section 2 we give an brief overview of our UML profile extensions. Our approach of modelling and generation of the whole application is described in Section 3. The novel optimization hints are presented in Section 4. Finally, the paper concludes with a discussion on further research aspects.

2 Domain specific PIM

Since our GignoMDA project is based on the idea of the MDA approach [2, 3], we start with a very brief overview of MDA concepts. The MDA approach introduces a *Platform-Independent Model* (PIM), which is an abstract model of the software system that does not incorporate any implementation choice, mostly used to describe the business logic. Furthermore, the PIM can be extended by application designers to a "marked PIM" where the model elements are marked with (1) stereotypes to define their functionality within the application, and (2) tagged values to add additional information for the code generation process. The *Platform-Specific Model* (PSM), consisting of the target application platform, is derived from this PIM. That means, the UML model as PIM is the mid-point within the MDA world and the vision is to generate full-operating application for different platforms from this single UML based specification.

Nowadays, most database applications typically consist of three layers: (i) presentation layer based either on web technology or on a rich client platform (RCP), (ii) business logic layer implementing the structure and the behavior of business objects, and (iii) the persistence layer implemented by a standard (mostly relational) database system. The explanatory power of a typical MDA-PIM is not sufficient to describe all three layers for database applications and therefore we extend the PIM with the following concepts:

1. New domain-specific stereotypes for all three layers, especially we integrate the modeling of the presentation layer in the whole process, which is totally missing in many approaches. With our extensions we are able to design complete database applications (including the user interface) and automatically generate full-operating applications from the model.
2. Moreover, we introduce novel stereotypes enabling

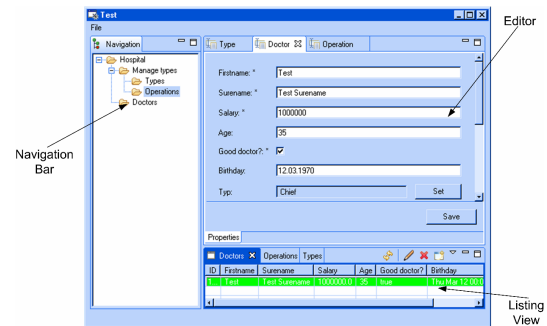


Figure 2. Fully Generated Eclipse-based Sample Application

cross-layer optimization for database applications. These stereotypes are considered in the generation process and the resulting application is optimized with respect to all layers.

3. New tagged values to annotate the UML model which are used in the UML2Code transformation. These new tagged values specify the utilization of the new stereotypes.

These additions, provided through an UML profile, do not change the platform independent character of the model. They do not contain specialisation to any platform but they transform the PIM into a domain specific model. The explanatory power is now limited to the database application's domain. Out of this domain model, it is possible to generate a complete database application, as we show in the following section.

3 Modelling the PIM

In this section we illustrate the new GignoMDA design process and the subsequent generation process with a database application, records doctors and their assigned surgeries within a hospital environment. This example application requires typical data management activities as other database application in this domain and we do not consider further extended business logic. In this example, the presentation layer is an Eclipse-based (<http://www.eclipse.org/rcp>) user-frontend (see Figure 2, which communicates via RMI with JBoss as business logic layer). The persistence layer is modeled via EJB supporting any relational database system. On this example database application we evaluate several cross-layer optimization hints, which are described in Section 4.

We start with the frontend modeling where the application frontend as depicted in Figure 2 is divided into three major parts: *Navigation bar*, *Listing View* and *Editor*. The

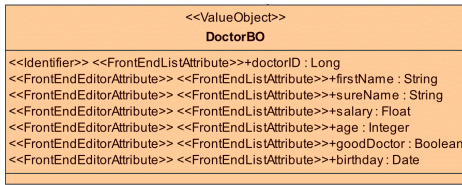


Figure 3. Sample Presentation Object

navigation bar illustrates the overall structure of the application and provides the entry points to the listing views and individual editors for data management. The navigation tree is defined by an UML use-case diagram. The hierarchy of the navigation node can be modeled by dependency connections between the use cases, annotated with a new stereotype.

The listing view comprises the result set of a database query, specified by an OCL constraint which will be transformed to either OQL (Object Query Language) or SQL. The tabular view displays only attributes of the underlying object which are annotated with the new stereotype `<<FrontEndListAttribute>>` (see Figure 3) or included in the tagged value `@client.view.table.columns`.

The editor view provides a mechanism to view and manipulate individual records shown in the listing view. The appearance and the rules for updates of the attributes and all participating associations are controlled by the stereotype `<<FrontEndEditorAttribute>>` (see Figure 3). Additional information—e.g. the label caption—are specified by tagged values. Moreover, the user’s input can be validated by given OCL constraints.

Besides the design of the frontend application we also have to define the set of interactions between the different parts of the application by a UML activity graph. Figure 4 shows a sample interaction pattern. With the activity state `RefreshDoctorView`, the activation of a navigation node opens the listing view `DoctorView` by calling the method `getAllDoctors`. As illustrated in Figure 5, this `<<FrontEndRefreshAction>>`-method is part of the set of C.R.U.D. (acronym for the life cycle operations Create, Retrieve, Update, and Delete) methods associated with each business object manager element (i.e. a class with the stereotype `<<Service>>`). These methods are either provided with default semantics (e.g. get all instances of the underlying business object for `getAllXXX`-methods) or replaced by methods with corresponding application-specific semantics. The presentation type of an activity state is set using the stereotypes `<<FrontEndView>>` and `<<FrontEndEditor>>`. Actions, like the opening of an editor or the deletion of an individual object, are modeled as transitions with the `<<FrontEndAction>>` stereotype.

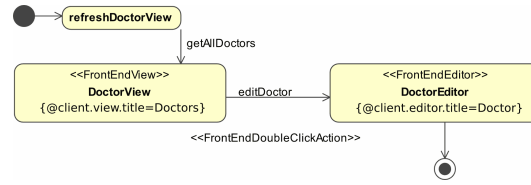


Figure 4. Activity Diagram

Additionally, Figure 4 shows the user interaction path from a listing view to the editor view by issuing a double click on a selected data record in the list. The `DoctorEditor` appears by calling the frontend-action `editDoctor`.

As already pointed out, the interaction patterns rely on methods which are performing the application logic, i.e. retrieving the underlying objects in the simplest case. For the ongoing example, Figure 5 shows all necessary model elements which are part of the logic layer for a specific entity, i.e. a Doctor object.

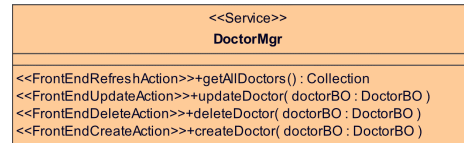


Figure 5. Sample Service

All C.R.U.D.-methods of an entity used by the editor or listing views are supposed to be defined within an object marked as `<<Service>>`-object. These objects represent the business logic and they can be easily extended. For example, operations with the stereotype `<<FrontEndCreateAction>>` will be implemented as a create method with standardized behavior according to the EJB specification. Furthermore, every retrieve operation must match a `<<FinderMethod>>` method of the—via `<<EntityRef>>`—referenced entity. The specific semantics of this method, e.g. the corresponding query, can be defined by an OCL constraint, or using the new tagged value `@persistence.operation.query`.

In addition to frontend and application patterns, the underlying database objects have to be modeled. The major aspects of data modeling are already provided by UML and the classes with the stereotype `<<Entity>>` corresponding to database objects. Additional aspects are (1) check constraints or triggers, that are modeled through OCL constraints and (2) database indexes for several attributes through tagged values.

The generated application features only the functionality described in the UML model which normally enfolds functions for data maintenance. Moreover, the GignomDA prototype provides options for projects with advanced claims on their applications, because almost every generated (Java)

class implements a generated interface. Using the behavioral pattern `template method` or `strategy` it is possible to integrate own code or change the behavior of existing functions. Therefore, GignoMDA is starting framework for ongoing database application development.

4 PSM - Model Hints

In contrast to the original MDA approach, the described PIM can be used for the code generation. It contains enough informations because of its domain specific character. The PSM is not necessary but within our GignoMDA project, we exploit the fact that the design and the potential content of the database have an impact on the presentation layer and vice versa by introducing the concept of *model hints*. For example, the application designer—usually supported by the domain expert—may specify the number of expected instances of objects or attributes already during the design phase. Such hints result in changes of the default behavior of the application (e.g. prompting for a search dialog to avoid mass loading when activating the listing view) and in additional DDL (Data Definition Language) operations with respect to underlying database system (e.g. enabling partitioning). Hints, in general, are therefore a central mechanism for cross-layer optimizations in large applications. Within the GignoMDA project, we distinguish two kinds of model hints:

- **Explicit Hints:** Explicit hints are added by the application designer via stereotypes or tagged values. An explicit hint annotates an model element a specific role or trait. For example, the stereotype `<<LookupEntity>>` tells the underlying system that the data are mostly read-only. The other extreme of the expected behavior can be annotated by adding the stereotype `<<UpdateEntity>>` telling the code generator to optionally add database parameter adjustments for extensive logging and locking.
- **Implicit Hints:** Implicit hints are derived by the generator from the specified structure and behavior and cannot be added by the application designer. For example, every association between two objects holds a tagged value `@client.association.displaytype` telling the code generator whether the association should be displayed as a set of check boxes or using a tabular list. By indicating the "check box-style", the designer implicitly denotes that the associated table will contain only a few objects. This information, can be used for example to exploit object-relational functionality of the underlying database system and create a schema holding the associated objects with a nested table.

Hints therefore play a general role in enabling cross-layer optimization out of the central specification. Furthermore, we are able to derive different optimization strategies for different architectures from the hints in the MDA development process in the code generation process. In contrast to the domain specific additions described above, hints have platform specific character. The PIM plus Hints construct the PSM which is the better base for the code generation because it contains more information for the UML2Code transformation. However, in the most cases, these optimizations are necessary for a well working application.

5 Open Questions and Conclusion

In this paper, we have presented our GignoMDA approach to model and automatically generate applications in the database domain. Furthermore, we have described our novel optimization hints for the challenging global optimization issue in this specific domain. Moreover, we believe that building of large database applications will definitely be based on an extensive portion of generated code in the near future. To realize this vision, research activities in the software as well as the database direction are necessary. In detail, our next research steps are:

- In one optimization-driven research activity we will extend our model hints technique. In this research direction further all kinds of hints will be investigated and defined for various architecture technologies.
- Another research activity is to enrich the functionality of GignoMDA to software and database evolution aspects, so that the GignoMDA approach can be used for the whole software development life cycle process. In this part, we will investigate how to model the software and database evolution.
- Yet another point of interest is, how more complex OCL constraints can be used. The advantage of the usage of OCL constraints is their domain independent character and they are not limited to a specific profile.

References

- [1] GignoMDA. <http://www.gigno.de.vu/>.
- [2] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.
- [3] Dave Thomas and Brian M. Barry. Model driven development: the case for domain oriented programming. In *OOP-SLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 2–7, New York, NY, USA, 2003. ACM Press.