# UML2SQL—a Tool for Model-Driven Development of Data Access Layer

Leszek Siwik, Krzysztof Lewandowski, Adam Woś,
Rafał Dreżewski, Marek Kisiel-Dorohinicki

Department of Computer Science
AGH University of Science and Technology, Kraków, Poland
{siwik,doroh,drezew}@agh.edu.pl

**Abstract.** The article is a condensed journey over UML2SQL: a tool for model-driven development of data access layer. UML2SQL includes an object query language and allows for behavior modeling based on UML activity diagrams, effectively linking structural and behavioral aspects of the system development. From the general idea of UML2SQL and its origins, we go through the details of its architecture and beyond the processes and schemes which make UML2SQL a distinct tool in the data access domain. Finally, an example of developing an application using UML2SQL is given as an illustration of its practical usage.

## 1 Introduction

The realm of Model-Driven Development (MDD) or, in general, Model-Driven Architecture (MDA) introduces the idea of a complete model-based application generator. Such a concept seems really attractive: it would allow concentrating only on an application's design, driving it to perfection. The process of coding and testing the application would be as simple as clicking a button and getting through some wizard dialogs. It is obviously a simplification, but indeed Model-Driven Development can be seen in such a light.

Unfortunately the research on existing projects leads to the conclusion that actually there is no fully functional MDD tool to facilitate application development in the domain of data access layer. Nevertheless, there are some CASE solutions which provide their users with various benefits of MDD. Mainly, these tools let one design and develop applications of the real-time genre (examples include Telelogic Rhapsody [21] and IBM Rational Rose [10]). There are also tools like LLBLGen Pro [12] or OrindaBuild [15], which let users generate code to support data access. But none of those tools can be regarded as really MDD ones, since the connection between application model and its development in general is rather scarce, if any.

The aim of UML2SQL is to fulfil the need to have a tool for data layer model-driven development. The ambition of being an MDD tool is only one aspect of UML2SQL—the other, equally important, is to provide a possibility to use only one integrated application model during all system engineering phases.

The paper is organized as follows. First, in section 2 a short discussion of existing MDD tools is given. Then, section 3 presents basic concepts of UML2SQL, and section 4—selected implementation details of the tool. In section 5 the process of modeling of a simple application is presented, which serves as a proof-of-concept of UML2SQL.

## 2 A glance at existing MDD tools

There are many examples of tools that try to make use of Model-Driven Development ideas and approaches, ranging from simple code generation extensions in UML-based graphical tools, to advanced modeling, development, simulation, and testing platforms.

The first complete tools for MDD were introduced in the field of real-time systems, with the most widely known example of Telelogic Rhapsody. Rhapsody is targeted at embedded and real time systems, with its goal set to provide a fully generated application code. It is able to construct all necessary artifacts necessary to build the system from a platform-independent model by targeting C/C++, Java or Ada languages. Rhapsody uses, known since the 70s, the Shlaer-Mellor method [32], which utilizes state machines to model behavior. By creating UML state diagrams with transitions and states annotated with conditions and activities, a complex behavior of a system can be described. The ability of adding a custom code to all elements of the diagram, which is later included in generated code skeletons, allows for specifying a complete system in UML only.

The generation of a complete application code from an UML model was not unique for the real-time systems' domain for a long time. Nowadays many tools allow for creating code from a UML model, and the extent and range of this code depends on the tool in question. Basic functionality of generating high-level programming language skeletons of classes (packages, modules etc.) is currently present in almost all UML modeling tools. However, they can not be perceived as Model-Driven Development tools merely because of the fact that they are, after all, general purpose modeling tools with simple option allowing for transformation of UML model into its another representation—a programming language. Only the presence of integrated and embedded behavior modeling mechanisms means that a specific tool is actually a Model-Driven Development one.

There are numerous examples of tools that facilitate behavior modeling using UML models, and the most widely known are IBM Rational Rose, Blu Age or AndroMDA. All of them, apart from allowing for modeling complete logical models, support modeling business logic of an application. This is achieved by making the use of many Shlaer-Mellor method variations or by using activity and sequence diagrams. For instance, the Blu Age documentation states that the modeling is done with the use of three groups of components [5]:

- class diagrams integrate the business objects notions and the simple and complex business rules, such as the entities,
- services' modeling, the same as the controllers', is made by the means of activity and sequence diagrams,
- the roles and authorizations are expressed by means of use cases.

| Tool / Platform | Orientation | UML Oriented | Structural modelling | Structural modelling scope | Behavior modelling | Behavior modelling approach | Add custom code during modelling |
|---|---|---|---|---|---|---|---|
| Telelogic Rhapsody [16] | real-time systems | yes | yes | complete logical model | yes | Shlaer-Mellor state machines | yes |
| AndroMDA [1] | MDA framework | yes | yes (via cartridges) | depending on the cartridge | yes (via cartridges) | N/A | N/A |
| AndroMDA BPM4Struts [2] | business process | yes | yes | complete logical model | yes | activity diagrams | yes |
| OpenArchitecture Ware [14] | MDA framework | yes (integrated via Eclipse EMF) | yes | depending on plugins | yes | depending on plugins | N/A |
| Blu Age [5] | JavaEE/.NET | yes | yes | complete logical model | yes | activity/sequence diagrams | yes |
| Middlegen [13] | DB persistance layer | yes | yes | database persistance layer using EJB, JDO, Hibernate etc. | no | N/A | N/A |
| IBM Rational Rose Data Modeler [9] | database applications | yes | yes | object models, data models and data storage models; physical-logical mapping | no | N/A | N/A |
| IBM InfoSphere Data Architect [8] | data layer | yes | yes | logical, physical domain models for various RDMSes | no | N/A | N/A |
| LLBLGen Pro [12] | data layer | no | yes | database model | limited | can design queries using a complex query builder | can include linq expressions |
| UML2SQL | data layer | yes | yes | complete logical model | yes | stored procedures/triggers by activity diagrams | yes (using a query builder) |

**Table 1.** Selected features of the most important MDD tools and platforms

All these tools also offer the ability to insert user-defined code into the generated code skeletons, thus ensuring that a complete, working application can be generated solely from the model. The option of reverse engineering is also available, making them even more productive.

The extensive and "rich" abilities of MDD tools for general application development are unfortunately insufficient in the field of data layer modeling. Even though, there are tools that allow for generating a database schema or database persistence layer code from a UML logical model, however the lack of support for modeling business logic in databases is obvious at first glance.

Such tools as IBM Rational Rose Data Modeler is able to convert a UML logical model expressed as class diagrams into a physical storage model in Relational Database Management Systems (RDMS), expressed in terms of tables and relations comprising a complete database schema. There are also tools, such as Middlegen, that are able to generate a database persistence layer code which facilitates using the generated schema from inside a programming language such as Java or C#. There is still no way, however, for modeling stored procedures, triggers or any other business logic that can be included in a database, which sets this scenario of MDD usage light-years behind the mainstream of "Java application" scenario. The ability, available in probably the most advanced data layer MDD tool—LLBLGen Pro, to design queries using a complex graphic query

builder and include linq queries [11] in the generated code is still nothing compared to the abilities of Telelogic Rhapsody or AndroUML.

A short comparison of tools mentioned above is presented in table 1. It is needless to say, that there is a need for a tool that in the context of data layer, apart from structure modeling, facilitates also behavior modeling. UML2SQL, which aims to be such a tool, is also included in the comparison, and will be presented in the following sections of this chapter.

## 3 UML2SQL concept

What would be the requirements like for a tool which could aid model-driven engineering? How do existing systems approach this problem? It is an obvious practice to base an MDD (CASE) tool on the Unified Modeling Language [10, 21]. Using this standard allows architect(s) to get accustomed with new functionality delivered by the tool quickly and painlessly. The power of UML [30, 27] itself is well known and we do not want to focus on it in this article. It has to be stated that UML2SQL is based on the UML standard too—as it is explained later (in section 4.1), the tool is designed as a plug-in to an existing UML modeling application.

Consequently, the process of designing system structure is similar for all types of CASE tools. We can model the application using different types of diagrams. Automating application code generation from its structure description is not a problem either. Although mapping of the object-oriented application model to the relational data model is not straightforward and requires some consideration regarding the choice of mapping strategy, it is well described in literature ([18, 22] and section 4.5) and many (if not all) UML modeling tools provide such functionality.

What is more, the structural model of an application can be used to produce method skeletons and CRUD-like (Create, Read, Update and Delete) operations which can be deduced from the model. Such model characteristics are also widely applied in various engineering systems. One may ask so, how does UML2SQL differ from the above-mentioned CASE tools? What is its added value?

### 3.1 Behavior modeling

The big issue starts when one wants to model the behavior of an application. It applies not only to UML itself, but foremost to the later possibilities of application code generation and general MDD tool operability. UML provides the designer with several types of diagrams (e.g. state diagrams, collaboration diagrams, sequence diagrams, etc.) which can be used to present system's behavior. However, even the most accurate diagram cannot be simply translated into source code. What is more, it is not the diagram's goal to be a complete template for code generation! To enable automatic application creation, some kind of an extension to UML has to be used, that would allow one to describe precisely the actions taken by the system.

The most popular approach nowadays is to model system behavior using the concept of state machines. The discussion on this idea dates back to the 1970s, when "Shlaer-Mellor Method" was presented [32]. It is widely used in real-time and embedded systems domain. Some other methods described in literature include:

- "Interaction Based" [20] modeling based on collaboration and sequence diagrams with an example of generating Java code from collaboration diagrams [24];
- "Contract Based" [20] method which abstracts from behavioral models and uses strongly the structural model, additionally equipped with some pre- and post-conditions imposed on operations and expressed in Object Constraint Language [29, 26];
- using Abstract State Machines with AsmL language to model application behavior and also to test and evaluate a running system [19];
- applying activity diagrams to represent application behavior, which is expressed by means of states and transitions [2]. Such an approach is used in AndroMDA powered BPM4Struts cartridge (Struts J2EE application generator) and it can be understood as some variation on state machines.

As a practical example of behavior modeling, Rhapsody by Telelogic [21] can be presented. The diagrams created to illustrate system functionality can be completed with Java or C++ code snippets which are later included in the generated code. In the case of modeling an "executable" application it is an acceptable approach. The high-level programming language code inserted into diagrams still holds the idea of developing the Platform Independent Model (PIM), which can be easily compiled to the platform specific application (PSM) [28]. But when it comes to model data access, things get complicated.

First of all, when accessing data there is no straightforward object collaboration or message passing that could be modeled by executing adequate methods. It is not a situation that can be described by changing application's or objects' states either. Moreover, it is hard to imagine having to insert handcrafted high-level programming language code, which would access the database into any type of diagram. Even inserting code based on object-relational mapping technology would be painful, as any model refactoring could make it obsolete and, additionally, translating such code into SQL could be difficult. The natural way of accessing data is SQL, but even if the MDD tool provided us with such a possibility, we would encounter another problem: the impedance mismatch [18] between object-oriented paradigm and relational databases. Finally, by using a dedicated SQL dialect we face the danger that changing the database vendor would require rewriting code which operates on and communicates with the database.

Proposed UML2SQL tool was designed in order to avoid the mentioned above problems. It lets one model procedures that access data with activity diagrams (see section 4.4). Additionally, a dedicated language (called *eOQL* for enhanced Object Query Language) was designed. eOQL is a simple extension of Hibernate Query Language (HQL) [6, Chap. 14] and can be understood as a middleware between the application's structural model and its behavioral model (more details may be found in section 4.3). With the assistance of a graphical editor one can create statements that are assigned to activity diagram nodes—each node has one statement—and later used to generate stored procedures in a required SQL dialect (described in details in section 4.5). The connection between structural and behavioral models as well as the whole system architecture will be elaborated in section 4. One thing worth of mentioning here is that when modeling procedures with the eOQL editor, one can focus on the structural (i.e. object) model only, and need not to worry about the low-level data model. In MDA terminology, one copes with PIM instead of PSM [28].

At this point, it has to be underlined also that proposed approach is one of many possibilities when it comes to generate code that operates on data. Stored procedures are taken under consideration since using them lets preserve the consistency of the whole generated system and integrates easily with other components which emerges as outcome of running UML2SQL generators. It can be imagined that instead of stored procedures for example high level programming language code is generated which connects to the database, queries data and runs according to the model data flow.

Questions may arise whether UML2SQL is something more than the already mentioned tools like LLBLGen Pro [12] or OrindaBuild [15]. These tools provide functionality of object-relational mapping and generation of database access code. Below it is argued that such functionalities are merely an outcome of UML2SQL, and not its essence.

### 3.2 Is this endeavor worth it?

"MDA is another small step on the long road to turning our craft into an engineering discipline." [28] Since designing a new system is often a challenge of adopting a new environment and a new technology, fast prototyping techniques are priceless. Still, the refactoring activities are very important from the quality point of view. Linking the application design directly with its code means the process of keeping both of them up-to-date is easy and can be done by clicking the proverbial button. It contributes not only to the quality of the code, but facilitates later system maintenance.

Apart from the purely engineering motives, the second equally important issue connected with UML2SQL is the aforementioned impedance mismatch between object-oriented approach and modeling data in a relational fashion. Relational databases are independent from programming languages and can be easily shared among multiple systems and users. It is also important that behind the relational model there is a highly reliable and mathematically proven theory, tested for a very long time. Moreover, the biggest database management systems vendors invested huge assets in their products and it is obvious that they will not easily give the stage to competitors providing other solution for managing data (e.g. object-oriented databases [25]). Even though coping with relational database engines introduces some problems in application development, it will not be abandoned in the near future. Consequently, the concept of keeping one common model describing relational data and object-oriented structure of the application seems crucial. The simplest solution is UML [30], but UML itself is not enough. Some kind of a connector between relational and object-oriented paradigms is necessary to enable using one model in both aspects.

## 4 UML2SQL deep dive

From the concept presented in the previous section, a standard usage scenario was created. It perfectly underlines the goal of UML2SQL. The list of steps to go through is as follows:

1. The user designs the application structural model in an UML modeling application.

2. The user also creates abstract procedures accessing data using activity diagrams (also in the modeling application).
3. Activity diagram nodes are populated by the user with eOQL statements, which are directly based on the model designed in the first step (see sections 4.3 and 4.4).
4. The tool generates Hibernate mappings, application source code, database schema and stored procedures (section 4.5).
5. Generated code is compiled against the database if necessary.

### 4.1 Background

As it was already mentioned, the ambition of UML2SQL is to become a complete MDD tool for data access layer development. Thus, the first thing to consider was the choice between delivering a proprietary UML editor or using an existing one, extending its functionality with plug-ins. We have chosen the latter. This choice was dictated by the potential to reach a wider audience (i.e. users of an existing tool), and of course by our limited resources. After thorough research we decided to use Visual Paradigm Standard Edition [17].

The second, even more important decision regarded the approach to the object-relational mapping (ORM). To achieve relational and object models consistency and ensure that UML2SQL user could focus on PIM [28] design, the following issues were taken under consideration:

- usage of a custom ORM implementation vs. usage of an existing ORM engine,
- providing a simple way to access designed and generated stored procedures,
- usage of a query language which would allow operating on object model,
- building a graphical query editor vs. parsing user input.

Our experience with first prototypes of UML2SQL let us decide to use an existing ORM engine. Hibernate [6] was determined to be a perfect solution for us. It not only supplies a recipe for mapping object model to relational model and the interface to use it, but it also provides a way to use stored procedures compiled against the database as any other query. This way the developer, equipped with specific Hibernate mappings generated directly from the application design by UML2SQL, can easily access data stored in the database and call procedures generated from the model.

As far as the query language is concerned, its origin is associated with the chosen ORM engine. The simple to learn and use Hibernate Query Language (HQL) [6, Chap. 14] was adapted for UML2SQL. In order to enable an architect to model the behavior precisely, HQL was extended to give birth to *enhanced Object Query Language* (eOQL), which is discussed in section 4.3.

### 4.2 UML2SQL architecture

In Fig. 1 a component view of UML2SQL is presented. Two main components can be distinguished: “UML2SQL Core” and “UML2SQL Plugin”. The main reason why presented system is split into such parts is maximal independence from the UML editor and its interface. As it was already described, the first two steps of the scenario presented

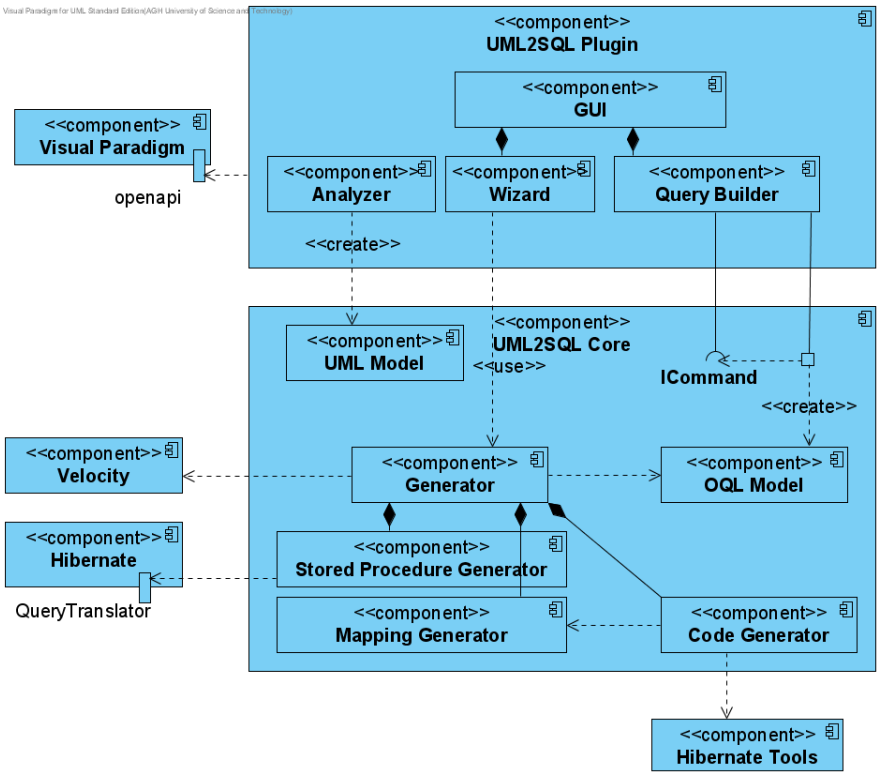


**Fig. 1.** UML2SQL component view

at this section's beginning are done using Visual Paradigm, but it can be imagined (and it was taken into consideration) that UML2SQL is plugged into another CASE tool when the "Plugin" component is replaced.

When the structural model of an application is ready, we can start playing with UML2SQL. When a request to build an eOQL statement for an activity diagram node is detected "Analyzer" is launched. It gathers the data from the UML editor and creates an internal UML model. Providing no errors or incompleteness of the UML model are detected, the "Query Builder" starts. This is a tool, an editor, which give an opportunity to build eOQL statements. From a software engineering point of view, it is worth to mention that the command design pattern was used here ([23]).

When activity diagrams are ready (i.e. populated with eOQL statements), application code can be generated. Using the "Wizard", the generation process can be customized in order to comply with application requirements. With the support of Velocity templates [4] and Hibernate [6] and using various generator types the application code is manufactured. The process is described in details in section 4.5.

### 4.3 eOQL—enhanced Object Query Language

The question which for sure arises is whether it was necessary to introduce a new language to satisfy the undertaken requirements. We have already mentioned that a language to perform queries in an object-like fashion does exist! What is more, HQL [6, Chap. 14] is not the only one language to do that. There are also other languages like JPQL and OQL. When using object query languages it is possible to benefit from the power of their declarative approach known from SQL language and to stay in the realm of object-oriented programming simultaneously. Therefore, the problem of impedance mismatch is avoided as one does not have to be aware of relational model when writing queries. So, why have we decided to go for the eOQL then?

Again, we should look at UML2SQL from MDA perspective. "Fully-specified platform-independent models (including behavior) can enable intellectual property to move away from technology-specific code, helping to insulate business applications from technology evolution and further enable interoperability." [31] The procedure that accesses data is not only a collection of queries that select, update or delete it. To specify precisely the behavior of such a procedure some additional functions have to be used. For example, it may be necessary to create new objects (which can be understood as inserting data into a database), specify conditions deciding about the execution path, call other procedures or use helper variables or transactions.

Some of those things are not purely object-oriented, but one has to remember that the aim of such modeling is to generate SQL stored procedure's code. Its nature however is not object oriented, but conforms to a procedural programming paradigm. The essence of eOQL is therefore to provide well-known techniques for querying data in an object-like manner, at the same time giving an opportunity to use other types of statements to model application behavior (business logic) as accurately as it is possible. eOQL is not an extension of OQL language itself. It should be treated as an extension of object query language concept in general.

One more important issue regarding eOQL is that UML2SQL's user does not have to be aware of its concept or presence. From the user's perspective, eOQL is a middleware between structural and behavioral models of the application. eOQL is hidden behind the graphical editor which provides the interface to build eOQL statements. Consequently, the knowledge of any query language and some rudiments of computer programming are perfectly enough to build eOQL statements and use UML2SQL.

Types of eOQL statements and their relation to SQL expressions are described in the following sections (4.4 and 4.5). The eOQL "grammar" also includes instructions which are not available to the end user, but are used during code generation. These are mainly flow control statements and declarations.

### 4.4 Behavior modeling

Since UML2SQL strongly depends on the provided UML model, the diagrams describing the application structure have to be adequately prepared. UML diagrams that are used for code generation must have some restrictions imposed on them. The following sections describe them in details.

**Class diagrams preparation.** In the case of class diagrams, our goal was to ensure maximum support for existing diagrams. Therefore, UML2SQL tool supports all class diagram elements by using the graceful degradation approach. That means that all unrecognized elements (that is, unnecessary as far as UML2SQL is concerned) of UML class diagrams are ignored. This is in contrast with the treatment of activity diagrams, which is described in the next section.

Bearing in mind that UML models can be constructed on different levels of abstraction, we decided to require an explicit indication of classes that should be included in the data model generated by UML2SQL. This allows the user to use an existing UML class model (with all the implementation-specific or utility classes) and only indicate to UML2SQL which classes should be included in the data model. In current implementation (under Visual Paradigm), this is achieved by adding the "ORM Persistable" stereotype to the class. Moreover, if system requirements demand having relations that keep the order of entities, one can use the "ordered" stereotype to mark the relations and have them generated appropriately later.

Apart from these requirements, there remain only a few rules. In general, a few details which are optional in UML class diagrams, should be defined. Among those are field names and association multiplicities. Without these details the engine would not be able to properly interpret class diagrams.

**Building activity diagrams.** Activity diagrams are treated in an entirely different way from class diagrams. The reason becomes apparent when one takes a look at a sample activity diagram presented in Fig. 2.

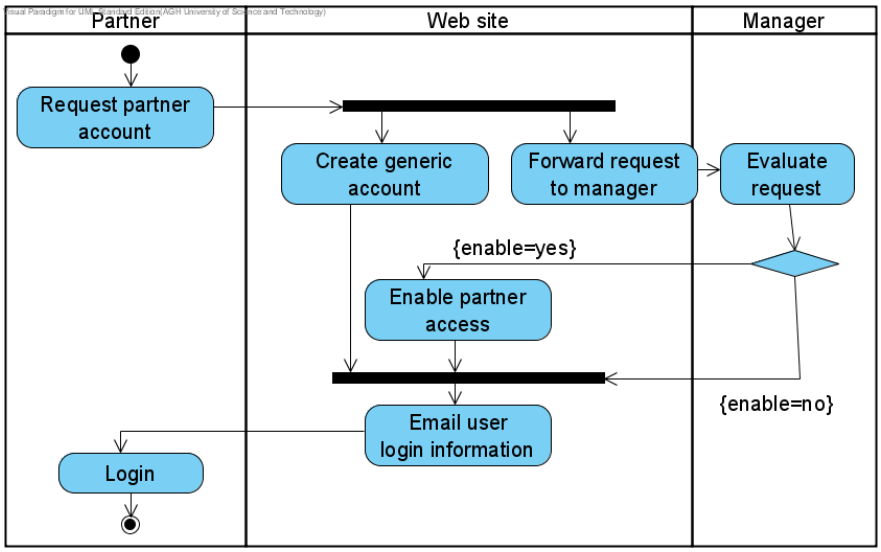


**Fig. 2.** An example of an activity diagram

From UML2SQL's point of view, there are several problems with such a diagram. First of all, several modeling constructs could not be translated to stored procedures because of lack of support in a declarative language that is SQL. These include, among others, elements related to object flow, constraints, synchronization elements (stored procedures are by design sequential and database engines do not provide any way of parallel execution inside a stored procedure) and exception handlers (some of the dialects provide exception handlers for stored procedures, but this is still far from the complexity that can be modeled in UML).

Moreover, even after restricting the structure of a diagram problems still remain mainly with the contents of the diagram. Human-friendly, textual information has little meaning for the procedure translation engine. Therefore UML2SQL requires that contents of activity diagrams are edited using an eOQL editor (see section 4.4).

To sum up, an activity diagram that can be translated by UML2SQL into SQL should meet the following criteria:

- it should contain only initial and final nodes, action nodes, decision nodes and loop nodes,
- each action, decision and loop node should have a statement assigned to it,
- each statement must be build using the eOQL editor (4.4).

**Using eOQL editor.** As it was already mentioned, the eOQL editor is a graphical tool to define eOQL statements which are assigned to activity diagram nodes. The classification of eOQL statements and their correspondence to the activity diagram nodes is presented in Table 2. The eOQL editor also provides a possibility to associate initial node with a list of variables which will be treated as procedure's arguments and to assign a variable to final node which would indicate that the user wants the procedure to return a value.

| General statement genre | Description |
|---|---|
| Query | Well known query types (e.g. "SELECT", "DELETE") have their direct counterparts in eOQL, whose semantics are just the same. "Pure" queries can be assigned to action nodes and additionally "SELECT" query can be used to define subqueries in other statements (including logical expression attached to decision node) and to define an iterable collection for loop node. |
| References | There are two statements which allow modeling an action of adding and removing an object to/from a collection (which corresponds to a relation modelled on class diagram). An example can be seen in section 5. |
| Condition | Decision nodes have to be populated with a statement modeling the logical expression determining the execution path. |
| Other | There is also a possibility to assign values to variables, execute other procedures and handle transactions. |

**Table 2.** eOQL statements types

Since eOQL operates on object model only, eOQL editor provides user with the concepts defined on the class diagrams, i.e. classes and their attributes. One has no awareness of how the object model is mapped to the relational model in the process of SQL generation. Described editor delivers some handy, wizard-like panels which assist user in query building in a fashion known from MS Access interface.

It has to be stressed once again that the main concept of introducing eOQL together with its editor is to set an additional layer of abstraction on the top of the object-relational mapping process. In that way one can model the behavior of the application operating on object model only preserving one of the fundamental assumptions that the whole process of modeling is based on one model, i.e. object model.

### 4.5 Code generation

Finally we can look at the output generated by UML2SQL. This is also a section which gives a basis for regarding UML2SQL as a complete MDD tool in the field of data access. After thorough design preparations, including building application structural and behavioral models and specifying concrete actions to be undertaken by created activity diagrams using eOQL editor, the code generation process can be launched and the delivered components can be used within an application.

**Generation plan.** The whole generation process is designed in a pipeline manner [23]. The initial input consists of the application class model, activity diagrams with assigned eOQL statements and database information (gathered by the "Wizard" mentioned in section 4.2). The process can be divided into three main stages, i.e.:

1. Hibernate mappings and configuration generation;
2. database schema and application source code generation and compilation (at the moment Java code generation is supported);
3. assembling stored procedures from activity diagrams and SQL code generation.

**Hibernate mapping files.** The first stage is done independently by UML2SQL itself. Using Velocity templates [4] the following Hibernate XML files are generated:

- configuration file (e.g. storing database connection data) [6, Chap. 3],
- mapping of persistent classes [6, Chap. 5-9],
- mapping of non-persistent classes,
- mapping of stored procedures [6, Chap. 16].

Two class mapping files are generated in order to distinguish classes from which SQL schema has to be generated. Both mappings are additionally equipped with some meta data describing the methods modeled on class diagrams and any other characteristics according to the Hibernate specification [6, Chap. 5-9]. As far as the object-relational mapping strategy is concerned, UML2SQL generates code which complies with "mapping each class to its own table" method [18]. Selected strategy provides intuitive operability at the object level and does not introduce any redundancy. However, performing operations at database level is sometimes a little bit complicated, especially when it comes to updating or deleting data.

**Hibernate tools in action.** During the second stage the outcomes of the first, presented above, are used intensively. Taking the advantage of Ant [3] and Hibernate tools [7], Java code for persistent and non-persistent classes is generated and compiled. The database schema and a Java interface with constants describing created procedures and their arguments are also created.

**Stored procedures generation.** The last stage is the most complicated one. The activity diagram itself is a graph describing possible execution paths. However, we do not cope directly with constructs known from programming languages. The naive approach to building procedures would include labelling every single statement and adding goto instructions when a "jump" is detected. Obviously, it is not an acceptable solution to have code full of the "hated" goto instruction. In UML2SQL, each diagram is subject to the loop detection algorithm as described in [33, Chap. 20], after which eOQL flow control instructions (like *DO-WHILE*, *WHILE-DO*) are added. *FOREACH* loop is generated for "Loop" nodes, "Decision" nodes are translated into *IF-THEN* statements. The variables that are used inside procedure's statements and are not its arguments are gathered to be listed on declaration list. After the procedures are assembled from the abstract eQOL statements, the SQL generator is launched. The generation process uses the following components:

- dialect specific *template provider*, whose responsibility is to feed the generator with an adequate Velocity template for a given eQOL statement;
- *Velocity templates* [4], which describe how given eOQL statement is translated into SQL code and, for statements other than queries, provide the recipe for direct generation;
- Hibernate "QueryTranslator" [6] which is used to transform "SELECT" queries;
- Hibernate "Executor" [6] which stores the execution plan for queries that modify data, i.e. "DELETEs" and "UPDATEs";
- compiled Java binaries (generated during the previous stages) which are used by components taking their origin in Hibernate.

All of the components mentioned above are dialect dependent and the generated code may differ when various SQL dialects ([6, Chap. 3]) are chosen. Finally, when the SQL code is ready it can be complied against the database if requested. There is also a possibility to create the database itself. The example of generated code can be found in section 5.

At the moment UML2SQL is equipped with generators which let one generate whole database system (i.e. database schema together with stored procedures) in MySQL and T-SQL. However, only code generated for MySQL is well tested. As far as database schema generation is concerned, all dialects supported by Hibernate library can be handled.

## 5 UML2SQL test case

This section serves as an illustration of the ideas introduced in the paper—an example of making a use of UML2SQL is presented. The goal is to show how UML2SQL can

be applied in the subsequent stages of the application development process. Generally speaking, such a process can be divided into some well-defined phases, and UML2SQL covers some parts of the application structural and behavioral design, development, testing and post-deployment maintenance.

The proposed example is based on an application that can be described as a computer aided register of tourist routes or as an *ElectronicRoadAtlas*. After pointing out how UML2SQL supports development of *ElectronicRoadAtlas*, a short discussion presents what has to be conducted to create a similar system without UML2SQL's help.

### 5.1 Definition of requirements

The first step of application development engaging fully software engineers is the process of defining and describing requirements for a created system. In figure 3 an use-case diagram is presented which conveys the requirements set for the application. Analyzing

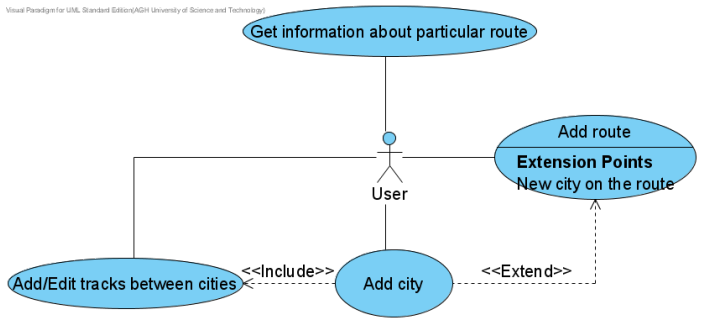


**Fig. 3.** Use case diagram describing requirements for the designed system

the diagram the following requirements can be distinguished:

1. Adding and editing routes between cities.
2. Defining a new city together with creating new links between existing cities and the one defined.
3. Adding new routes, which can require adding non-existent cities, which are present on the added route.
4. Retrieving the information about existing routes.

If the scope of the UML2SQL is taken under consideration, the requirements should be seen more from the data access layer domain perspective. Consequently, the problem of user interface will not be described here since the main stress is put on the way how selected functionalities are exposed in the programming interface.

### 5.2 Structural modeling

Having the requirements' definition, one can focus on architectural-oriented activities. The first step is the preparation of a structural model. The form of the model itself is

strongly determined by the requirements for the final application. All requirements have to be satisfied, therefore the structural model cannot restrict the development of some areas, which can make final product unacceptable. An exemplary model, which can be understood as a database of information regarding tourist routes, is presented on the diagram 4.

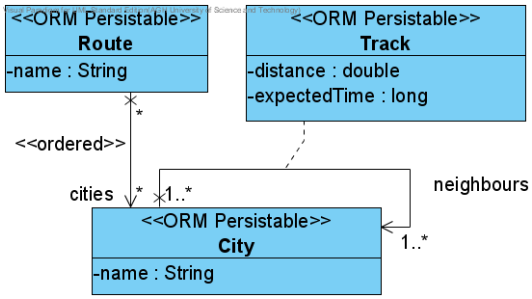


**Fig. 4.** Computer aided route database system data model

A particular tourist route (described by the `Route` class) is represented by a list of subsequent cities which appear on it. The association class `Track` describes data characterizing the link between two cities. The relation of self-aggregation for the `City` class can be understood as an extended representation of distance table which can be commonly found in the road atlas books. It is user responsibility to keep the data correct, i.e. the distance between cities A and B should be the same as the distance between B and A.

### 5.3 Business logic modeling

Taking into account the typical UML2SQL usage scenario the next step is the modeling of business logic, which can cover some of the application requirements. Some of the requirements will be satisfied thanks to the applying Hibernate library with delivered (generated) by the UML2SQL tool configuration and object-relational mapping files only. The requirement which would demand delivering a stored procedure is “Getting information about particular route”. A model of an algorithm, which computes a distance and an expected travel time (two figures are returned as a result) is shown on diagram 5. The UML2SQL procedure model is constructed as an activity diagram. The algorithm computing mentioned data is based on iteration process, which step by step sums up the distances and expected travel times assigned to the subsequent cities’ links appearing on the considered route.

### 5.4 Application code generation

Referring to the typical UML2SQL usage scenario again, the next step to take after having all modeling activities finished is to launch application code generation process.

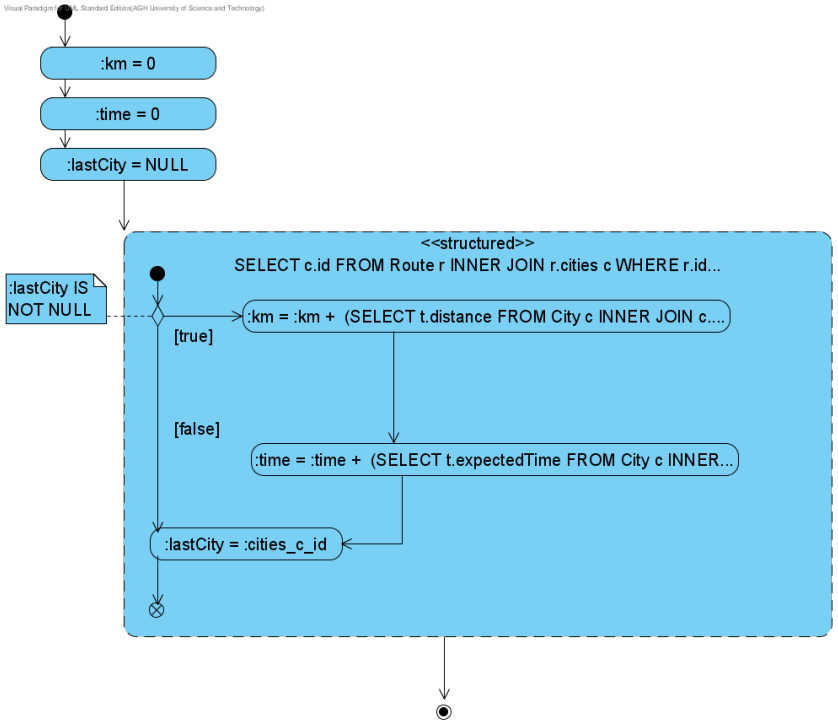


**Fig. 5.** Model of a procedure computing expected time and travel time of chosen route

As a result UML2SQL gives the following artifacts which can be used in various ways at the subsequent development stages:

- Hibernate configuration files,
- Hibernate object-relational mapping files,
- database schema (in a chosen SQL dialect),
- SQL stored procedures code (in a chosen SQL dialect),
- Java code of structural model and some additional interfaces facilitating further development.

The code of a stored procedure based on the activity diagram shown in fig. 5 is presented below:

```
CREATE PROCEDURE SP_getExpectedTimeAndDistance ( route bigint )
BEGIN
 DECLARE km double precision ;
 DECLARE time double precision ;
 DECLARE lastCity bigint ;
 DECLARE cities_c_id bigint ;
 DECLARE _fetchFailed integer DEFAULT 0 ;
```

```
DECLARE cities CURSOR FOR select city2_.id as col_0_0_ from route
 route0_ inner join routeCities cities1_ on
 route0_.id=cities1_.routeId inner join city city2_
 on cities1_.cityId=city2_.id where route0_.id=route
 order by cities1_._index;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET _fetchFailed = 1;

SET km = ( 0 ) ;
SET time = ( 0 ) ;
SET lastCity = ( NULL ) ;
OPEN cities ;
FETCH cities INTO cities_c_id ;
WHILE ( NOT _fetchFailed )
 DO
 IF ( lastCity IS NOT NULL ) THEN
  SET km = ( ( km +
       ( select neighbours1_.distance as col_0_0_ from city city0_
    inner join track neighbours1_ on city0_.id=neighbours1_.cityId
    inner join city city2_ on neighbours1_.targetCityId=city2_.id
    where city0_.id= cities_c_id and city2_.id=lastCity )));
  SET time = ( time +
    ( select neighbours1_.expectedTime as col_0_0_ from city city0_
    inner join track neighbours1_ on city0_.id=neighbours1_.cityId
    inner join city city2_ on neighbours1_.targetCityId=city2_.id
    where city0_.id= cities_c_id and city2_.id=lastCity )) ;
 END IF;
 SET lastCity = ( cities_c_id ) ;
 FETCH cities INTO cities_c_id ;
END WHILE;
CLOSE cities ;
SELECT km result1, time result2 ;
END
```

Finally, a question has to be raised how the generated application code conforms to the requirements assumed. The generated procedure's code can be compared with an algorithm described in the previous section.

### 5.5 Applying generated code

The generated source code for data model and associated object-relational mapping files provide an easy and comfortable way to manage the persistence layer. Entity adding, editing or deleting is done simply by creating Java objects and invoking adequate Hibernate session methods. For example, the addition of a new city to a database is as simple as `City` class object creation and Hibernate session save method invocation. Summing up, Hibernate engine gives an access to persistence layer management functions in very flexible way. Additionally, the generated stored procedures make using application business logic possible, which in a consequence gives an entire well-defined interface to the layer implementing all requirements for a given system.

A Java code snippet is presented below, which applies generated by UML2SQL and complied against database stored procedure to sort a list of all existing routes retrieved from the database:

```
SessionFactory sessionFactory = new Configuration().configure(
        "hibernate-config-sql.xml").buildSessionFactory();
final Session session = sessionFactory.openSession();
List<Route> routes = (List<Route>) session.createQuery("from Route").list();

final Map<Route, Double> lengths = new HashMap<Route, Double>();
Collections.sort(routes, new Comparator<Route>() {

 @Override
 public int compare(Route route1, Route route2) {
  double d1 = getDistance(route1);
  double d2 = getDistance(route2);
  return Double.compare(d1, d2);
 }

 private double getDistance(Route route) {
  if (!lengths.containsKey(route)) {
    Query query = session
    .getNamedQuery(ProceduresRepository.SP_GETEXPECTEDTIMEANDDISTANCE)
    .setLong(ProceduresRepository.SP_GETEXPECTEDTIMEANDDISTANCE_ARGUMENT_1,
    route.getId());
    List<Object> result = query.list();
    if (result.size() == 1 && result.get(0) instanceof Object[]) {
        lengths.put(route, (Double) ((Object[]) result.get(0))[0]);
        } else {
                lengths.put(route, new Double(0));
        }
 }
 return lengths.get(route);
}
});
for (Route r : routes) {
    System.out.println(r.getName() + ": " + lengths.get(r));
}
session.close();
```

At first, Hibernate library is configured and initialized with an XML file generated by UML2SQL too. Then a list of all routes is loaded from a database using an HQL query. Finally, a list of routes is sorted using an anonymous `Comparator` object in which the stored procedure is called to compute a total distance of a particular route.

### 5.6 Developing a system without MDD/MDA tool support

How would the application development process look like without the support of the tool such as UML2SQL? The question should be seen from the perspective of the implementation process comfort and the clarity and integrity of the code. Assuming that

the system model is the same, the problem of providing a separate physical data model occurs. That model has to be used to create database schema. Since there are many-to-many relations in object model—prepared data model would have to introduce some additional elements (i.e. association tables) which would cover those issues. So, already at the stage of modeling, models become incompatible and introduce additional difficulties associated by the impedance mismatch problem.

As usually, such an incompatibility introduced at the very beginning of the application development process returns with double power in the following stages. To name only a few, the implementation of data access methods would have to find a solution for translating SQL tables' rows into objects. Mixing SQL code with high-level programming language constructs introduces unclarity and make any code refactoring almost impossible. Mentioned issues can be handled by a programmer but the problem would arise if the application model changes. In such a case, all native SQL code has to be at least double checked, and what is more probable—fixed, so that it conforms to new database schema. The lack of automation in refactoring code responsible for data access layer management and the lack of separation of SQL code handling business logic can coerce into spending long hours on analyzing and fixing existing SQL code.

## 6 Conclusion

"The aim (of MDA) is to build computationally complete PIMs" [31] where the term "computationally complete" means capable to be executed. As presented in the previous section, using UML2SQL it is possible to create the model of an application, which after translation into particular source code can be used as a component of a "production-ready" system. The activity diagrams enriched with eOQL statements processed by UML2SQL code generators produce SQL stored procedures that can be executed completely independently. It can be said that in the domain of developing data access layer UML2SQL deserves the title of an MDD tool.

Once more we can go back to the advantages of using MDD tools. What is the most frustrating activity for developers? From our experience we would point out coding the same thing countless times and getting to know a new technology without any supportive examples as the most irritating. But having an MDD tool one may overcome those issues quite easily. Instead of writing the same thing again and again, one may simply generate it from the application model. In this way the developer's productivity and satisfaction will certainly increase meaningfully. As an MDD tool for building data access layer UML2SQL can be also used in such aspects.

One may argue that MDA as an idealistic concept which has no rights to exist in reality. However, an example of UML2SQL shows, that in a domain of data access layer code generation, tool based on MDA concept can be quite useful, and what is more, its usage does not require any additional, technology-specific, knowledge. The idea of a system modeled once—deployed anywhere gains its full meaning here.

UML2SQL tool is available at http://uml-2-sql.sourceforge.net/.

## References

1. AndroMDA. http://andromda.org/.

2. AndroMDA BPM4Struts. http://galaxy.andromda.org/docs/andromda-cartridges/andromda-bpm4struts-cartridge/index.html.
3. Apache Ant 1.7.0 Manual. http://ant.apache.org/manual/index.html.
4. The Apache Velocity Project. http://velocity.apache.org/engine/releases/velocity-1.5/.
5. Blue Age tools. http://www.bluage.com/index.php.
6. Hibernate Reference Documentation. http://www.hibernate.org/hibdocs/v3/reference/en/html/.
7. Hibernate Tools - Reference Guide. http://www.hibernate.org/hibdocs/tools/reference/en/html/.
8. Ibm InfoSphere Data Architect. http://www-01.ibm.com/software/data/studio/data-architect/.
9. Ibm Rational Rose Data Modeler. http://www-01.ibm.com/software/awdtools/developer/datamodeler/.
10. Ibm rational rose realtime. IBM.
11. Linq: .NET Language-Integrated Query. http://msdn.microsoft.com/en-us/library/bb308959.aspx.
12. LLBLGen Pro. http://www.llblgen.com/defaultgeneric.aspx.
13. Middlegen, Boss. http://boss.bekk.no/boss/middlegen/index.html.
14. OpenArchitectureWare. http://www.openarchitectureware.org/.
15. Orinda Build. http://www.orindasoft.com/public/features.php4.
16. Rhapsody, Telelogic. http://www.telelogic.com/products/rhapsody/index.cfm.
17. Visual Paradigm. http://www.visual-paradigm.com/documentation/.
18. S. W. Ambler. *Agile Database Techniques*. John Wiley and Sons, 2003-2007.
19. A. Andrzejak. Modelling system behaviour by abstract state machines. Technical report, ZIB Berlin, 2004.
20. N. S. Ashley McNeile. Methods of behaviour modelling - a commentary on behaviour modelling techniques for mda.
21. M. B. Charles W. Krueger. Leveraging the model driven development and software product line engineering synergy for success. Technical report, Telelogic, 2008.
22. J. L. Dave Minter. *Pro Hibernate 3*. Apress, 2005.
23. R. J. Erich Gamma, Richard Helm and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
24. S. S. G. Engels, R. Hucking and A. Wagner. Uml collaboration diagrams and their transformation to java, 1999.
25. J. L. Harrington. *Object-Oriented Database Design*. Academic Press, 200.
26. W. J. and K. A. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.
27. I. J. James Rumbaugh and G. Nooch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
28. J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. OMG, 2003.
29. OMG. *Object Constraint Language*. OMG, 2.0 edition, 2006.
30. OMG. *Unified Modeling Language: Superstructure*. OMG, 2.1.1 edition, 2007.
31. O. Sims. Mda: The real value, 2002. www.omg.org/mda/presentations.htm .
32. M. J. B. Stephen J. Mellor. *Executable UML*. Addison-Wesley, 2002.
33. R. L. R. T. H. Cormen, C. E. Leiserson and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, 2001.