Evaluation of Maintainability of Model-driven Persistency Techniques

Thomas Goldschmidt
Software Engineering
FZI Forschungszentrum Informatik
Karlsruhe, Germany
goldschmidt@fzi.de

Jochen Winzen
andrena objects ag
Karlsruhe, Germany
jochen.winzen@andrena.de

Ralf Reussner
Institute for Programming Structures and Data Organisation
Universität Karlsruhe (TH)
Karlsruhe, Germany
reussner@ipd.uka.de

Abstract

Although the original OMG Model-Driven Architecture Approach is not concerned with software evolution, model-driven techniques may be good candidates to ease software evolution. However, a systematic evaluation of the benefits and drawback of model-driven approaches compared to other approaches are lacking. Besides maintainability other quality attributes of the software are of interest, in particular performance metrics. One specific area where model driven approaches are established in the area of software evolution are the generation of adapters to persist modern object oriented business models with legacy software and databases. This paper presents a testbed and an evaluation process with specifically designed metrics to evaluate model-driven techniques regarding their maintainability and performance against established persistency frameworks.

1 Introduction

The MDA as defined by the OMG [10] is associated with a number of benefits for software engineering projects, such as flexibility, cost efficiency and platform independence. However, it was mainly intended to be used for developing new stand-alone applications. However, the software engineering reality is different: Efforts for software evolution supersede any other part of the software life cycle. Therefore, approaches to apply model-driven techniques to software evolution projects is an promising approach.

In the following we consider a software engineering approach as model-driven if it (a) includes meta-modelling and (b) makes use of well-defined transformations [9]. By that we intentionally broaden the definition of the OMG which defines various specific kind of models, such as computation independent models, platform independent models and platform specific models. However, this broader view eases the application of model-driven approaches to software evolution questions.

If one accepts this broader view on model-driven software engineering, adapter generation, as used for connecting new code with legacy applications or legacy data sources can be done in a model-driven manner. Using the information contained in the business object model and the relational database scheme a transformation should generate a formal mapping between both models. For example, the model-driven legacy integration tool suite by the Delta Software Technology Group [2] is such an approach, developed independently from OMG's MDA approach.

Given the platform independence and flexibility of such model-driven techniques, they are a promising approach for software migration projects, as they map novel object models to relational databases (or other data storage technologies, such as ISAM). For exactly the same reasons, model driven persistency layers sound attractive, as they explicitly decouple business logic from data storage technology. However, in this domain, other technologies are successfully used, in particular, so-called persistency frameworks which persist object models in relational databases. According to recent discussions [4] this topic is still an important issue. In addition, high-level interfaces to databases as provided by modern programming environments form a competing alternative. An example (although platform specific) is ADO.NET [1] in Microsoft's .NET environment

which provides flexibility on the query side and exchangeable databases at the back end. Given the number of software applications used for business and administrative tasks and their life-span which often lasts over decades, it becomes clear that software evolution issues are one major driving factor for the rather important selection of the right persistency approach (model-driven, framework based or high-level interfaces). Besides maintainability and flexibility most often performance is of major concern for such an persistency approach.

Comparisons of such persistency systems are mostly done by comparing their features [8, 6]. Unfortunately, until now there is no systematic evaluation of the model-driven persistency approach comparing the competing alternatives (like the other alternatives are also not systematically compared). It is clear, that such a systematic comparison should evaluate the three above mentioned approaches regarding their maintainability and performance.

The main contribution of this paper is the presentation of an evaluation method for model-driven persistency techniques against these other techniques. The evaluation method comprises a testbed and goal-driven defined metrics (according to the GQM approach [5]) for maintainability and performance of the competing solutions. By this, this article contributes to the empirical evaluation of model-driven techniques. In the long term, software developers should benefit from this work by more educated decisions on the deployment of model-driven techniques for persistence in particular. Besides, the method presented here can be generalised to a methodology for the systematic evaluation of model-driven techniques applied also to other software engineering concerns beyond data persistency.

This work is part of the research project *Model Driven Integration of Business Information Systems (MINT)* [12]. MINT is supported by the German *Federal Ministry of Education and Research* in the scope of the *Forschungsoffensive Software Engineering 2006*.

The paper is structured as follows. In section 2 specific design guidelines for such a testbed are discussed and the architecture of the testbed is presented. Likewise, section 3 presents the process of deriving appropriate metrics and presents the metrics as such. Section 4 lists and briefly explains the persistency techniques that will be evaluated. Preliminary results of using the testbed with existing persistency technology are sketched in section 5. Section 6 concludes and discusses the planned future evaluation work with the testbed.

2 The Evaluation Testbed

2.1 Design Guidelines to Ensure Validity of Measurements

The design of the testbed to evaluate and compare persistency techniques highly influences the *internal* and *external* validity of the results [7]. While internal validity describes the quality of the results regarding the single empirical investigation (e.g., case study or experiment), the external validity is concerned with the extent to which the results generalised beyond the single case study or experiment. Often, these both kinds of validity counter each selves.

As a testbed, we selected a real-world multi-tier business application. In this application, the various persistency techniques are deployed. This forms the only variation point of the application. To support external validity, the architecture described below is intentionally a "typical application": From a technical viewpoint, its architecture could be used for many business applications beyond its specific application domain. To yield a high internal validity, one has to design the testbed in a way, that the specific persistency technique is deployed in its best specific way. For example, in EJB 1.x and 2.x several patterns had to be considered to yield a satisfying performance in certain situations. However, such patterns are specific to a persistency technique and hence not part of the testbed which has to be designed to allow the use and exchange of various persistency techniques. Therefore, one could question the internal validity of the results, by arguing that the way we use a specific technique in our general architecture is uncommon to its most optimised use in other situations. This would also lower the external validity. However, there are some arguments to counter:

- A technical concern, such as persistency, should not influence the design of the domain oriented business logic tiers anyhow. If the persistency technique requires such a specific design of the middle tier, then this is a shortcoming of this technology and should be made to be a pre-condition of its successful application. Much more, such a technology has to be evaluated and compared against competing techniques intentionally without realising any specific technology concern in the middle-tier.
- All approaches deployed here, are know to yield an acceptable performance without strong modifications of the business tier.

2.2 The Architecture of the Example Application

One of the first decisions is to select an suitable example application for the evaluation testbed. On one side it has

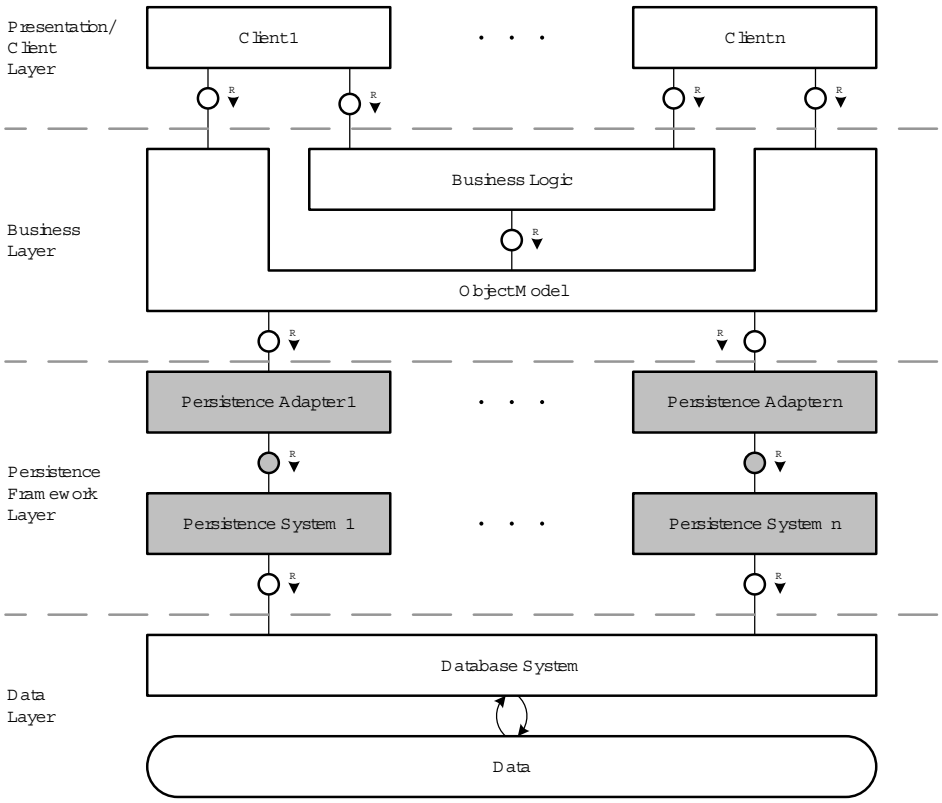

Figure 1. Testbed Architecture

to be simple enough to allow the implementation of a plug-in architecture for the different persistence systems and to measure all defined metrics (see section 3). On the other side it should be a *real-world* scenario that represents the vast number of legacy applications out there.

The current evaluation testbed is based on MESCOR, an extensive program suite for financial research. This program suite is a group of standalone applications using a common server middle tier to access the database. All application data is stored in one centralized Oracle database which contains more than 100 tables. The stored information includes very different aspects of the financial research domain like companies, shares, analysts, industrial sectors and all related data items.

MESCOR is in daily use by several customers for one decade now and maintained by andrena objects ag. Client and server software is written in Borland Delphi and communicates via DCOM and sockets. Since 2005 some parts of the system were migrated to Microsoft.NET (e.g. MESCOR webclient). In advance an in-depth analysis of all relevant use cases and the database structure was performed. So it was possible to choose some parts of the whole system as examples to evaluate the different techniques.

The first metrics will be measured with the *Chartproduction* application which creates charts for all share prices stored in the database. It is a simple use case but it requires a high-performance data access layer and therefore is a good example application to judge the performance characteristics of the competing solutions. The other applications in the evaluation testbed will focus on topics like complex read/write operations (object trees, updating collections etc.) or user interaction scenarios (short response time required).

The evaluation is restricted to compare the different solutions for persistency only. We designed a generalized testbed architecture (see figure 1) to provide a common approach to exchange the persistence system and gather the metrics. All testbed applications are structured using 4 layers, namely (bottom-up):

- The *data layer* is just the database which is the same for all applications.
- The *persistence framework layer* is the most important part of the testbed. This is the place where we inspect and measure the different persistence systems (e.g. ADO.NET, NHibernate [3], the MINT approach

[12]). Every system has to implement a common set of service interfaces that define the connection to the business layer. The mediation between these interfaces and the persistence system itself is accomplished by a individual persistence adapter for each system. We use the façade pattern for the interface definition and implementation in this layer.

- The *business* and *client layers* are strictly separated from the persistence framework layer and only know its common interface. So there's no need to change them when switching the persistence system.

All layers exchange information using a common object model which was modelled in UML and is tightly related to the existing database structure. So the object model mainly consists of simple data transfer objects (DTOs). Since the testbed is developed in C# with Microsoft.NET the objects are implemented as so called "plain old C# objects" (POCOs). The persistence framework layer has to return its query results using this object model, too. All layers only know the interface definitions of the object model. So any of the persistency techniques may use its own implementation of the object model if necessary.

In summary the evaluation testbed is a collection of sample applications that all share the generalized testbed architecture. The sample applications focus on different usage scenarios to evaluate the competing persistency techniques. All variation is bundled into the persistence framework layer which contains the particular persistency technique and a specific persistence adapter. The other three layers stay constant for every application. Every technique has to provide the same functionality and thus fulfil equal evaluation criteria.

3 Derivation of Metrics

3.1 The Goal-Question-Metric Approach

We defined the quality plan, which we will use to evaluate the persistency techniques using the Goal/Question/Metric (GQM) approach. The GQM approach [5] is a systematic method to find and define tailored metrics for a particular environment. In contrast to the collection of metrics that are chosen just because they can be measured, the GQM approach helps to identify the reasons why particular metrics are chosen. It also helps to interpret the values resulting from the collection of these metrics. The GQM approach is a top-down methodology that consists of three steps:

1. Starting from the definition of *goals* that should be achieved by the conducted measurements. A goal is defined using a template which consists of the following parts:

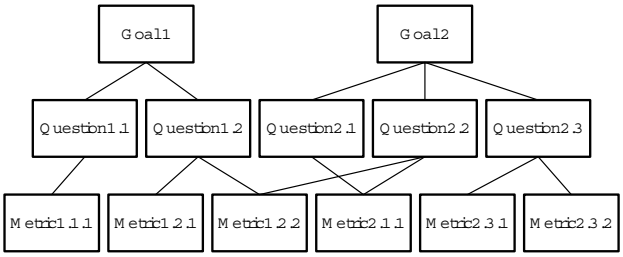

Figure 2. GQM structure as defined in [5]

   Purpose What should be achieved by the measurement?
   Issue Which characteristics should be measured?
   Object Which artefact will be assessed (this may be a product, a process or a resource)?
   Viewpoint From which perspective is the goal defined (e.g. the end user or the development team)?

2. The next step is to define *questions* that will, when answered, provide information that will help to find a solution to the goal.

3. To answer these questions quantitatively every question is associated with a set of metrics. It has to be considered that not only objective metrics can be collected here. Also metrics that are subjective to the viewpoint of the goal can be listed here.

Figure 2 depicts the three levels of the GQM approach. The goals are defined on the conceptional level.

3.2 GQM-Plan for Maintainability

Maintainability is hard to measure with empirical metrics because many volatile factors (such as developer experience, development paradigm, etc.) have great influence on it. That is the main reason why it is hard to validate analytical metrics. Unlike other maintainability evaluations that rely on direct analysis of source code (e.g. as published in [11]) a comparison between model-driven and conventional development has to focus on more abstract metrics. Maintenance operations on models versus maintenance operations on source code can not be compared by metrics that are based on the source code. In a model-driven development environment a maintenance task will mostly be solved by changing either the application model, or in some cases by changing the generator. On the other hand changes in non-model-driven environments are performed by changing the source code directly. In the model-driven environment the code is being generated, therefore the code complexity does not influence the maintainability. Hence metrics such as code complexity or size cannot be reasonably applied here.

A solution that facilitates this comparison is to define project specific, empirical metrics. These metrics can be found by using the GQM method and creating questions that involve specific scenarios and tasks that can be found in the particular project or domain. The major disadvantage of these metrics is however that they are consequently only valid in the narrow domain for which they where defined. Instead of being universally valid they only provide results for the particular goals of the evaluation plan.

To measure the maintainability of a persistency technique we defined our first measurement goal according to the GQM method as follows:

Goal 1: Purpose: Comparison
Object: Different persistency techniques
Issue: Maintainability
Viewpoints: The software development and maintenance team

The following questions were elaborated to cover the first goal:
Some of the questions are of rather general nature as they ask for common development efforts.

Question 1.1: How big is the initial effort to understand the technology and to create a design how the technology can be implemented into the system?

In order to answer this question we will gather the following metrics:

M1.1.1: *Person-days* to design and implement the mappings that are needed for a special application (MESCOR Chartproduction). This includes the time that is needed to understand the technique and do initial work, such as implementing a generator for a model-driven technique.
M1.1.2: *Amount of aspects* that have to be implemented manually (such as transaction handling, caching, queries and referential integrity).
M1.1.3: *Amount of workarounds* that were needed to implement the technology in the system.
M1.1.4: Amount of time, measured in *person-days*, spent to implement the workarounds that were needed to implement the technology in the system.

As our example application consists of several more or less independent parts we want to measure the initial development effort for each of them.

Question 1.2: What is the effort to implement further parts of the system?

M1.2.1: *Person-days* to implement the mappings that are needed for each application.
M1.2.2: Time (in *hours*) spent for testing and debugging.

To be able to make fine-grained statements about the maintainability of each persistency technique we identified the most important and most frequently conducted tasks. As the persistence framework layer is a glue-layer between database and object model changes made to either of those will presumably result in changes to the persistence framework layer. Apart from small, trivial adjustments a non-trivial change is primarily either an addition, change or removal of a persistent class or relation. The resulting questions for the GQM plan are:

Question 1.3: How big is the effort to extend the persistency layer with a new persistent class?

These metrics will help to answer question 1.3:

M1.3.1: Time to conduct the change in *hours*.
M1.3.2: *Amount of files and/or models* that need to be touched.
M1.3.3: *Amount of test and debug runs* that were needed to pass all tests after the change.

Correspondingly to the effort of adding persistent classes we defined a question for persistent relations.

Question 1.4: How big is the effort to extend the persistency layer with a new persistent relation?

Metrics that will help to answer this question are:

M1.4.1: Time to conduct the change in *hours*.
M1.4.2: *Amount of files and/or models* that need to be touched.
M1.4.3: *Amount of test and debug runs* that were needed to pass all tests after the change.

Not only extensions of the static parts of the application will be measured, also extensions to the business logic have to be considered:

Question 1.5: How big is the effort to extend the business façade with new functionality?

The same metrics as for question 1.3 and 1.4 apply here as well:

M1.5.1: Time to conduct the change in *hours*.
M1.5.2: *Amount of files and/or models* that need to be touched.

M1.5.3: *Amount of test and debug runs* that were needed to pass all tests after the change.

For questions 1.3 and 1.4 we defined several change scenarios that necessitate these changes. To comply with the external validity of our evaluation these change scenarios were elaborated based on the experiences made with the legacy system.

Depending on the project, e.g. whether it is a newly developed application or if an already existing database should be integrated into a object-oriented environment, different abilities of the mapping technology are crucial. In a migration project the ability of a persistency technique to bridge large differences between database and object model layout may be of capital importance. Whereas in a project which has no need to respect legacy applications this aspect may be less important. To reflect these considerations in the evaluation we elaborated three different scenarios in which we will execute our measurements:

1. Definition of a new database schema according to the object model.
2. Mapping the newly created object model to the legacy database.
3. Migrating the mapping from the legacy database to a newly defined database schema.

In each of these scenarios we will collect the presented metrics. This will lead to a differentiation which persistency technique is most suitable for a particular scenario.

3.3 GQM-Plan for Performance

Apart from maintenance aspects the most often the performance of a persistency technique is one of the major concerns. To measure the performance of these techniques we defined our second measurement goal according to the GQM method as follows:

Goal 2: Purpose: Comparison
Object: Different persistency techniques
Issue: Performance
Viewpoints: The software development and maintenance team

The questions that will cover the second goal can be parted into two categories. The first contains questions that are of a general nature. Those can be asked for all client applications that come with the example system. The latter covers questions that are specific to actions that can be performed in special applications. The application specific questions and metrics are tailored according to the functionality of each application. Because it would require specific knowledge of the applications on order to understand them reasonably only the general questions and metrics will be mentioned here. The general questions are:

Question 2.1: Which impact does the choice of the persistency technique have on the time that is needed to start the application?

To measure this question we derived the following metrics:

M2.1.1: Time to initialize the application measured in *milliseconds*. This metric is measured from the entry of an special `init`-method until its end. Within this method initializations components such as database connections and caches are executed.

The interpretation of the results of this metric will be based on the assumption that a short initialization time is better than a long one. However it is hard to say whether a vast or a narrow distribution where the values are rather small respectively large is more convenient. Anyhow we derive several dependent metrics from M2.1.1 including statistical values such as minimum, maximum, mean value or standard deviation. In this way, it is possible to distinguish each persistency type according to the current requirements of an application.

Another important issue is to identify where the bottlenecks of a persistency technology are. Some may be faster executing queries but at the cost of a larger cache and therefore a higher memory consumption. To find these bottlenecks we derived the following question:

Question 2.2: Which layers (database, persistence, business logic) and resources (CPU, memory, network) are bottlenecks for the application?

The metrics that will help to answer this questions measure the resource utilisation per layer. E.g. CPU load for the database server or memory usage of the persistence layer. Each combination of layer, resource and business façade method will be measured separately.

M2.2.1: CPU utilization of business layer. Measured in *milliseconds* that are spent in the classes of the business logic.

M2.2.2: Memory utilization of business layer. Memory consumption by classes of the business logic. This metric is split into (a) the maximum memory (in *megabytes*) used during the test run and (b) the integral of the memory usage over the time of the test run (in *megabytes over seconds*).

M2.2.3: CPU utilization of persistence layer. Measured in *milliseconds* that are spent in the classes of the persistence layer.

M2.2.4: Memory utilization of persistence layer. Memory consumption by classes of the persistence layer. This metric is split into (a) the maximum memory (in *megabytes*) used during the test run and (b) the integral of the memory usage over the time of the test run (in *megabytes over seconds*).

M2.2.5: CPU utilization of database layer. Measured in *milliseconds* that are spent for the database operations.

M2.2.6: Memory utilization of database layer. Memory consumption of the database. This metric is split into (a) the maximum memory in *megabytes*) used during the test run and (b) the integral of the memory usage over the time of the test run (in *megabytes over seconds*).

Almost all applications that come with the example system have direct interactions with the user. Hence it is important for the performance of the system how responsive it is.

Question 2.3: How good is the responsiveness of the system from the client's view?

All operations that can be performed within one of the applications are bundled in the corresponding business logic façades. Therefore the responsiveness of each of these methods will reflect the performance of the user interaction.

M2.3.1: Response time in *milliseconds* of the methods of the business logic façades. Measured for each method using different parameter assignments.

M2.3.2: *Frequency distribution* and *standard deviation* of the response time of each method. Measured for using different parameter assignments.

Another very important requirement for a persistency technique is their scalability with an increasing number of requests. Consequently the following question will be asked:

Question 2.4: How big is the impact of an increasing load on questions 2.2 and 2.3?

M2.4.1: To find out the scalability behaviour of each persistency technique the measurements of questions 2.2 and 2.3 will be performed using an increasing amount of clients doing different operations.

Depending on the usage scenario there are different distribution possibilities for each layer. E.g. the database and the persistence framework layer may be run on different servers. To identify the impact of the client-server communication on the overall performance of the persistency technique we added another question:

Question 2.5: How big is the part of the client-server communication in the response time?

M2.5.1: Perform measurements for questions 2.2 and 2.3 on one server as well as distributed over two different servers. Measure the difference for each operation.

4 Assessed Persistency Techniques

The following persistency techniques will be assessed during our evaluation:

- **Manually implemented persistence layer:** High-level interfaces to databases such as such as ADO.NET give developers flexibility on the query side and exchangeable databases at the back end. Based on these interfaces an object relational mapping will be implemented.
- **Persistence framework:** Persistence frameworks (in our case NHibernate) provide an automatic mapping of objects to database rows using annotations or external mapping specifications. Such frameworks also include a vast range of features including caches and transaction handling.
- **Generated adapters:** These techniques can be used to generate adapter classes tailored for specific domains.
- **Combination of generator and persistence framework:** Generators can be used to create mapping definitions and supporting classes for a persistence framework. We will evaluate two different kinds of this combination:
  - **Out-of-the-box generators:** Existing generators that can produce mapping files for persistence frameworks.
  - **Project specific generators:** A project specific generator is created for the use in a specific environment and to generate exactly the output that is needed for the project.

5 Preliminary Results

In this paper we presented a testbed architecture we intend to use for the evaluation of different model-driven and non-model-driven persistency techniques. With defining the

testbed and elaborating the GQM plans for maintenance and performance we laid the headstone for this comparison. As for now we started to implement the persistence adapters for ADO.NET and NHibernate for the parts of the object model and business façade that are necessary to complete the aforementioned Chartproduction application. The model-driven persistency techniques will be implemented as a next step.

The first test runs to collect the performance metrics, which we conducted in prior to the actual evaluation showed that both techniques are almost equal concerning the performance of the Chartproducion tool. However, a critical aspect seemed to be the caching mechanism of NHibernate. The Chartproduction tool has a special batch run mode during which it creates charts for all shares in the repository. During this run the time for creating a single chart increased linearly, resulting in an overall runtime being ten to fifteen times higher than the respective ADO.NET implementation. After changing the NHibernate adapter to clear the session cache after the creation of a chart the production times were similar for both technologies. It is important to mention that the performance of those first implementations was only nearly equal for this particular use-case. Further tests revealed that the performance differs in other scenarios.

The maintainability measures have not yet started, however there is one metric (M1.1.1) for which we already have some results: Even though it is difficult to measure the initial skill adaptation another preliminary result was the time we needed to implement this first part of the adapters. The NHibernate adapter took us 9 days until all our tests were successful, where the ADO.NET adapter took us 12 days to complete.

6 Conclusions and Future Work

As mentioned, the results presented are preliminary. The actual contribution is the presentation of a systematic approach to validate the benefits of model-driven techniques compared to other approaches for object persistency. From a more general viewpoint, such a systematic approach to evaluate and compare the benefits of model-driven development help to better understand when its specific advantages justify its additional effort of meta-modelling and the definition of transformations.

In our specific MINT project the next step is to compare project-specific generators against out-of-the-box generators for this domain. By doing this we hope to find out if the overhead of implementing a very specific generator pays off by lowering the development and maintenance efforts of the actual application. We are currently working on the implementation of a project-specific generator that will create an NHibernate mapping including all necessary classes, such as the POCOs and manager classes.

References

[1] ADO.NET. http://msdn2.microsoft.com/en-us/library/e80y5yhx.aspx. Last Access: 26-01-2007.

[2] Delta Software Technology. http://www.d-s-t-g.com. Last Access: 26-01-2007.

[3] NHibernate for .NET. http://www.hibernate.org/. Last Access: 26-01-2007.

[4] Objects and Databases: State of the Union 2006, A panel discussion at OOPSLA 2006. http://www.ddj.com/dept/database/194400088?pgno=1, November 2006. Last Access: 26-01-2007.

[5] V. Basili, G. Caldeira, and H. D. Rombach. *Encyclopedia of Software Engineering*, chapter The Goal Question Metric Approach. Wiley, 1994.

[6] Cunningham & Cunningham Inc. Object Relational Tool Comparison Dot Net. http://c2.com/cgi/wiki?ObjectRelationalToolComparisonDotNet, November 2006. Last Access: 26-01-2007.

[7] B. Freimut, T. Punter, S. Biffl, and M. Ciolkowski. State-of-the-art in empirical studies. Technical Report ViSEK/007/E, ViSEK, University of Kaiserslautern, 2002.

[8] F. Marguerie. Choosing an object-relational mapping tool. http://madgeek.com/Articles/ORMapping/EN/mapping.htm. Last Access: 26-01-2007.

[9] S. J. Mellor, A. N. Clark, and T. Futagami. Model-driven development. *IEEE Software*, 20:14–18, 2003.

[10] O. M. G. (OMG). Model driven architecture - specifications, 2006. Last Access: 26-01-2007.

[11] C. S. Ramos, K. M. Oliveira, and N. Anquetil. Legacy software evaluation model for outsourced maintainer. In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004*, pages 48–57, Mar 2004.

[12] N. Streekmann, U. Steffens, C. Möbus, and H. Garbe. Model-driven integration of business information systems. In *Softwaretechnik-Trends*, volume 26, pages 9–13, November 2006.