

A Meta-Model for Enterprise Applications

Cristina Marinescu
LOOSE Research Group
“Politehnica” University of Timișoara
cristina.marinescu@cs.upt.ro

Ioan Jurca
Department of Computer Science
“Politehnica” University of Timișoara
ioan.jurca@cs.upt.ro

Abstract

In the last years, as object-oriented software systems became more and more complex, the need of performing automatically reverse engineering upon these systems has increased significantly. This applies also to enterprise applications, a novel category of software systems. As it is well known, one step toward a research infrastructure accelerating the progress of reverse engineering is the creation of an intermediate representation of software systems. This paper shows why existing intermediate representations of object-oriented software are not suitable for performing reverse engineering upon enterprise applications and proposes an intermediate representation (a model) for enterprise applications which facilitates the process of reverse engineering upon this type of applications. Based on an experimental study conducted on three enterprise applications, we prove the reliability of the introduced approach, discuss its benefits and touch the issues that need to be addressed in the future.

Keywords: reverse engineering, meta-models, models, enterprise applications

1. Introduction

In the last years, as object-oriented software systems became more and more complex, the need of performing automatically reverse engineering (*i.e.*, the part of the maintenance process that helps to understand the system in order to be able to make the appropriate changes [1]) upon such systems has increased significantly. This applies also to enterprise applications, a novel category of software systems which are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data [5]. Usually, enterprise applications involve two programming paradigms: the object-oriented and the relational paradigms, the last for ensuring the persistency of the involved data within the application.

As stated in [15], one step toward a research infrastructure accelerating the progress of reverse engineering is the creation of an intermediate representation of software systems. An intermediate representation for reverse engineering must support different levels of abstraction - from the code-structure level up to architectural level - to be suitable for all phases of reverse engineering [6].

Probably the best-known intermediate representation of a software system is his *model*. The model of a given software system contains specific information extracted from the source code based on a *meta-model*. Different meta-models for representing object-oriented software have been defined, *e.g.*, FAMIX [16] and MEMORIA [12]. The aforementioned meta-models specify the relevant entities (*e.g.*, classes, methods etc.) and their relevant properties and relations (*e.g.*, inheritance, method calls) found in an object-oriented system.

In this context a crucial question is: *Is it suitable to perform reverse engineering upon an enterprise application using a meta-model for representing a regular object-oriented system (i.e., a system which does not involve persistency)?* As we will see next, the answer to this question is negative.

In this paper we introduce DATES, a meta-model for representing enterprise applications. The proposed meta-model contains, on one hand, entities from regular object-oriented systems and, on the other hand, entities regarding the relational part of enterprise applications and the interactions among them.

The paper is structured as follows: in Section 2 we show why the answer to the above question is negative. Next (Section 3), we present our approach for modeling enterprise applications. This is followed by a brief description of the tool support (Section 4) that ensures the automation of the entire approach and by an experiment (Section 5) based on three case-studies. The conducted experiment is intended to reveal the applicability and the accuracy of the approach. The paper concludes with a discussion on related work (Section 6) and some final remarks towards the future work (Section 7).

2. Enterprise Applications and Intermediate Representations

Usually, an enterprise application involves a lot of persistent data (usually the persistency is supplied by a database, mostly relational database), its users manipulate the data concurrently and has a lot of user interface screens [5]. Its architecture is multi-layered, consisting of three primary layers namely *data source* (ensures the communication with the database), *domain* (encapsulates the logic of the system) and *presentation* (ensures the interaction between the user and the application) [4]. In an enterprise systems each design entity (e.g., class, method) belongs to a layer.

An intermediate representation (IR) of an enterprise application for reverse engineering tools has to be fulfilled, according to [7], the following requirements¹:

- (R2) “The semantics of the IR must be well-defined and it must exactly describe the constructs of the modeled programming languages; this is necessary for an exact analysis”.
- (R9) “IR should support different levels of granularity from fine-grained to coarse-grained”.
- (R12) “In a reverse engineering environment IR must also capture higher level abstractions”.
- (R13) “Not only does the IR have to have the ability to specify higher concepts, it also must provide means to express any relationships between these concepts”.

In order to answer to the question *Is it suitable to perform reverse engineering upon an enterprise application using a meta-model for representing regular object-oriented system?* let us consider the example presented below: we have an enterprise application which contains the table called *books* with the structure presented in Figure 1 and the class *Book*, which is responsible for updating the aforementioned table – its implementation is shown in Figure 2.

```
create table books (
  ID int primary key, title varchar,
  author varchar, publisher varchar,
  year int)
```

Figure 1. Table books.

If we extract the model of the source code from Figure 2 based on a meta-model for representing regular object-oriented systems (e.g., FAMIX[16] or MEMORIA[12]),

¹ We omit to present the requirements which do not affect the reverse engineering process of an enterprise application, compared to a regular object-oriented application.

we will extract the entities: class *Book*, method *updateAuthor* and related entities strongly connected to method *updateAuthor* – library classes *String*, *Connection*, *PreparedStatement*, *int* and library methods *prepare*, *setString*, *setInt*, *execute*. Next, we are going to see if the requirements R2, R9, R12 and R13 for an intermediate representation for reverse engineering are fulfilled by meta-models for representing regular object-oriented systems.

```
class Book {
  public String updateAuthor(int id, String
    title, String publisher)
    throws Exception {
    ...
    Connection con = ... ; //initializations
    PreparedStatement updateStatement;

    String update;
    update = "UPDATE books SET title = ?, " +
      "publisher = ? WHERE ID = ?";

    updateStatement = con.prepare(update);
    updateStatement.setString(1, title);
    updateStatement.setString(2, publisher);
    updateStatement.setInt(3, id);

    updateStatement.execute();
  }
}
```

Figure 2. Class Book.

- **R2** – as we can notice, class *Book* beside constructs that are present in a regular object-oriented system, contains also a construct connected to the relational

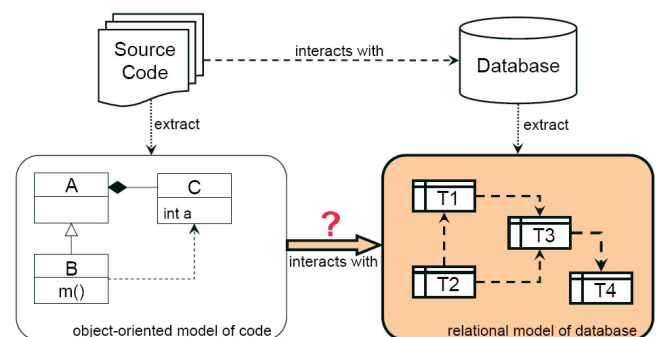


Figure 3. Design information must be extracted both from the source code and the database schema.

part of an enterprise application: a SQL statement embedded in a string that is executed by the method *executeQuery()*. This construct is not captured by a meta-model specific to a regular object-oriented system.

- **R9** – Regarding the different levels of granularity from fine-grained to coarse-grained IR should support, a meta-model specific to a regular object-oriented system like FAMIX[16] or MEMORIA[12] is not able to support fine-grained levels of granularity. For example, it can not store information about accessed tables from the body of a method, due to the lack of **R2**.
- **R12** – the condition regarding higher level abstractions is also broken. For example, within a regular object-oriented application there is not a *data source* layer and, consequently, it is not possible to map automatically entities from the source code into the data source layer. Due to the fact that usually meta-models support annotations, it is possible to annotate manually entities as belonging to the data-source layer. It is obvious that for large-scale enterprise application this is not a proper solution for reverse engineering. Moreover, annotating manually entities is an error-prone operation.
- **R13** – usually, enterprise applications involve two paradigms (*e.g.*, object-oriented, relational). Thus, in such applications we have, on one hand, object-oriented concepts like classes, methods, attributes and, on the other hand, relational concepts as tables, columns, primary and foreign keys. Consequently, a proper meta-model for reverse engineering for enterprise applications should contain, beside concepts from object-oriented paradigm, concepts related to the relational paradigm. Moreover, as R13 states, a proper meta-model for enterprise applications must provide means to express existing relationships between the involved concepts (*e.g.*, accessed tables from a class, classes that access a table).

The given discussion reveals that *in order to perform reverse engineering on enterprise applications we need a specific meta-model which contains, on one hand, entities from a regular object-oriented system and, on the other hand, entities regarding the relational part of the enterprise application and the interactions between the two paradigms (see Figure 3).*

3. DATES - A Meta-Model for Enterprise Applications

In this section we introduce DATES, the meta-model for representing enterprise applications we have developed in order to facilitate the process of reverse engineering upon

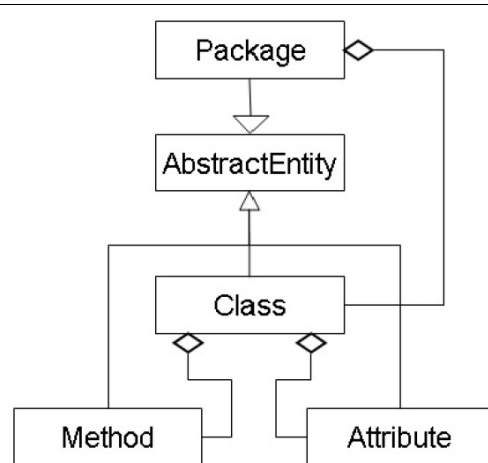


Figure 4. A Meta-Model for Object-Oriented Systems.

this type of applications. The meta-model for enterprise applications contains, as Figure 3 presents, entities specific to regular object-oriented systems, entities representing the relational database and the interactions among them. Next, for each type of design information from the proposed meta-model we dedicate a section.

3.1. Modeling Entities from Object-Oriented Systems

Entities from object-oriented systems can be modeled using an object-oriented language. In this case, the meta-model is represented as an interconnected set of data classes, usually one for each type of design entity. The fields of a class modeling a design entity are either elementary properties of that design entity or links to other related data structures. For example, a structure that models the *Class* design entity is expected to have a field of type *Method* that establishes its connection to the methods the class contains. In Figure 4 we show a simplified depiction of a meta-model for representing entities from object-oriented systems.

3.2. Modeling Entities from the Relational Database

A step further in building a meta-model for enterprise applications is to model the entities of the involved relational database.

According to the presentation of relational databases from [13], a relational database consists of one or more tables where each table has its own schema. A schema of a table consists of the name of the table, the name of each field

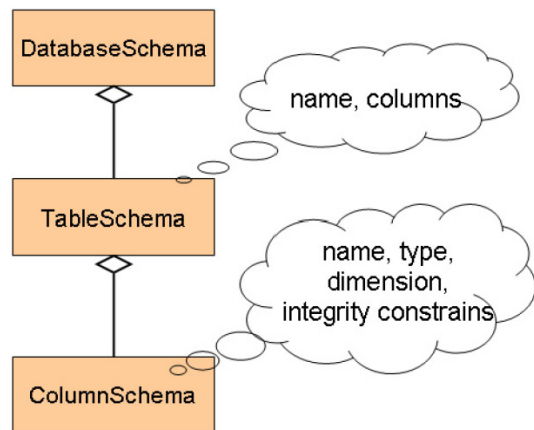


Figure 5. A Meta-Model for Relational Databases.

(or attribute or column) and the type of each field from the table. Additionally, upon a schema can be created integrity constrains. Thus, a meta-model for a relational database must contain, like in Figure 5, entities for modelling the tables and columns found in a relational database. Thus, the structure *TableSchema* that models the *table* design entity is expected to have a field of type *ColumnSchema* that establishes its connection to the columns the table contains.

3.3. Modeling the Interactions between the Object-Oriented and Relational Entities

As we mentioned before, the entities (*e.g.*, classes, methods) that ensure the communication with the relational database belong to a layer called *data source*. Consequently, between the object-oriented part of an enterprise application and the relational part there are interactions only within the data source layer. Thus, finding the interactions between the object-oriented and relational entities requires the identification of the entities (*e.g.*, classes and methods) that belong to the data source layer.

In order to do this, we take a simple approach to this issue, by taking into account the various usages of third-party libraries and/or frameworks that are specific for the data source layer.

Mapping of Methods to the Data Source Layer. A method is considered to belong to the data source layer if it invokes one or more methods from a specific library that provides an API for accessing and processing data stored in a data source, usually a relational database (*e.g.*, for systems written in Java, the method invokes the `executeQuery()` method from the `java.sql` package).

Mapping of Classes to the Data Source Layer. A class containing one or more methods belonging to the data source layer will be mapped to the data source layer.

The communication between the object-oriented part of an enterprise application and the relational database is performed within the methods belonging to the *data source layer*, usually, by executing SQL commands as embedded strings from well-known methods as `executeQuery(String sql)`, `executeUpdate(String sql)`. Thus, the methods of classes are the primary entities that ensure the communication with the databases - in this context, the *Method* entity from the object-oriented meta-model has to be enriched with information regarding the operations performed upon a relational database. The proposed solution regarding this issue is presented in Figure 6. Class *Method-DATES* contains information regarding the operations upon the database the method performs: *e.g.*, delete, insert, select, update, each of these operations involving one or more table. Class *SQLStatement* contains an attribute that stores the accessed tables from the operation. The information regarding the accessed tables from an entity is spread from low-level entities (operations perform within the bodies of methods) to high-level entities according to the following rules:

- a method stores the set of the accessed tables from its body.
- a class stores the set of the accessed tables from its methods.
- a package stores the set of the accessed tables from its classes.

At this moment, finding the classes that access a particular table called *myTable* requires two embedded iterations, the first through the classes from the data source layer and the second through the accessed tables from the inspected class. This operation for large-scale enterprise applications might be time-consuming and for this reason we decided to introduce within the entity *TableSchema* also information regarding the entities that access the table (*e.g.*, classes). The classes that access a particular table are store within the *TableSchema* in a field *accessed-by*.

4. Tool Support

In this section we present the tool called also DATES² we have developed in order to model (*i.e.*, obtain an intermediate representation) enterprise applications. DATES can be used for modeling enterprise applications written in Java where the persistency is provided by SQL relational

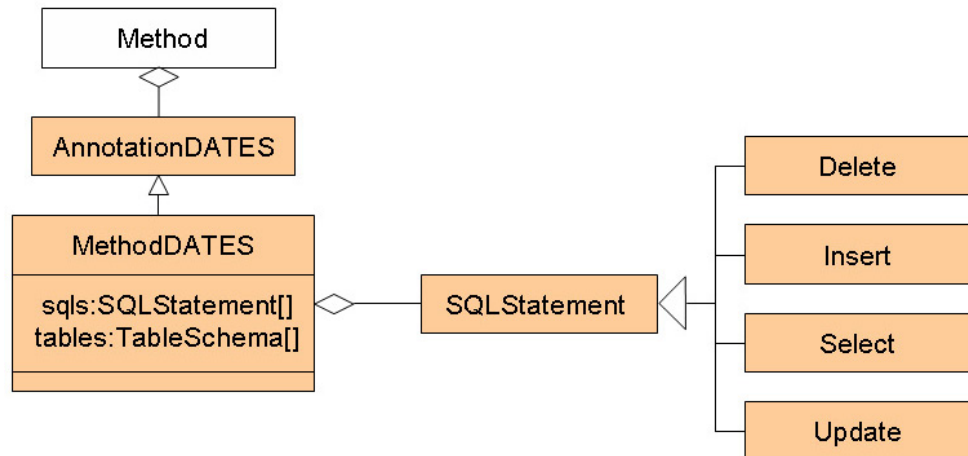


Figure 6. Entity Method.

databases, in particular MySQL and Microsoft Access. The communication with the relational database is performed by executing SQL commands as embedded strings from well-known methods as *executeQuery(String sql)*, *executeUpdate(String sql)*. Due to the fact that the meta-model for enterprise applications contains entities specific to regular object-oriented systems, we decide to not build DATES from the scratch – we built it on the top of the MEMORIA [12] meta-model.

The extraction of design information from enterprise applications requires the following steps:

- Construct the *model of the system* based on MEMORIA [12] meta-model. At this phase, the model of the system is not enriched with specific information regarding the interaction between the object-oriented and relational paradigms found in enterprise systems.
- Load the DATES tool in the INSIDER [10] front-end. At this moment the model of the system is enriched with specific information of enterprise applications' design, both for the relational paradigm (tables, columns, primary and foreign keys – obtained from the schema of the database by an extractor we have developed) as well as for the object-oriented one (*e.g.*, accesses to database tables from the bodies of methods within the data source layer, obtained also by the extractor).

In practice, for large-scale software enterprise applications, the extraction of the models would be probably useless without a proper tool support that enables browsing through the design entities of the software applications. DATES was integrated within the IPLASMA [10] environment in order to

- browse through the design entities.

- assess automatically the quality of the design of enterprise applications.

The quality of the design of enterprise applications is assessed using regular object-oriented assessment techniques (*e.g.*, software metrics, detection strategies [11]) and also specific assessment techniques like the ones from [9].

5. Evaluation of the Approach

In order to evaluate the approach we have conducted different experiments on a suite of enterprise applications. In this section we present the results obtained by applying the Tool Support described in Section 4 upon three enterprise applications. The size characteristics of the systems are summarized in Table 1. Within the **SALARY** application the persistency is provided by a MySQL relational database while within **KITTA** and **TRS** the persistency is provided by relational databases created in Microsoft Access.

System	Size(bytes)	Classes	Methods	Tables
KITTA	212,553	37	254	10
TRS	537,058	54	500	10
SALARY	780,871	115	580	12

Table 1. Characteristics of the case studies.

The purpose of the experiments was to answer questions regarding the reliability of the introduced approach:

- Does the model of the relational database capture all the entities from the relational database (*e.g.*, tables, columns)?

- Are the entities from the data source layer correctly classified?
- Are the accesses from the object-oriented design entities to relational tables well-captured?
- Are the *accessed-by* relations regarding the tables from the relational database well-captured?

5.1. Capturing the entities from the relational database

For each application, we performed a manual investigation in order to find out if the model captures all the entities from the involved relational database in the analyzed enterprise software system. The manual investigation consists of

- checking if the number of tables from the model (last column from Table 1) is equal to the number of tables from the involved database.
- checking, for each table, if the captured name and columns in the model correspond to the existing name and columns in the table from the database. We consider a column from the model corresponding to a column from a relational table if they have the same name and the same characteristics (*e.g.*, type, dimension, integrity constraints).

The performed investigation reveals that the model captures all the existing entities from the relational database within the enterprise application.

5.2. Classifying entities from the data source layer

Classifying correctly the entities from the data source layer plays an important role due to the fact that only entities classified as belonging to the data source layer interact with the relational database (*i.e.*, the accesses between an object-oriented entity and a relational entity are established only if the object-oriented entity belongs to the data source layer). In this context, a first step in validating the correctness of our approach regarding the existing relations between the two involved paradigms is to check if the object-oriented entities that interact with relational entities were classified correctly as belonging to the data source layer.

Classifying correctly the entities from the data source layer in an enterprise applications means that:

- among the entities that were classified as belonging to the data source layer there are no false positives (*e.g.*, entities erroneously identified by the approach as belonging to the data source layer).

- among the entities that were not classified as belonging to the data source layer there are no false negatives (*e.g.*, entities which were not identified by the approach as belonging to the data source layer).

For each analyzed enterprise application the number of object-oriented entities that were classified by our approach as belonging to the data source layer is presented in Table 2.

	KITTA	TRS	SALARY
Data Source Classes	9	10	16
Data Source Methods	25	24	74

Table 2. Size of the data source layer.

In order to find out whether design entities were correctly classified we performed, like in the previous case, a manual investigation. During the manual investigation, we browsed through the source code of each enterprise application in order to classify every class as belonging/not belonging to the data source layer. For each class identified as belonging to the data source layer we counted the number of its methods that ensures the communication with the data source layer.

The classification obtained by performing the manual investigation of **KITTA** and **TRS** enterprise applications coincides with the one performed by our approach (*i.e.*, there are no false positives and no false negatives). But within the **SALARY** application the manual investigation classifies 18 classes as belonging to the data source layer, instead of 16 which were automatically classified. Among the 16 classes which were classified automatically as belonging to the data source layer there are no false positives. As well, within the classified methods of the 16 classes there are no false positives.

In order to find out why within the model of the **SALARY** application there are two false negatives (*i.e.*, the two classes which were manually identified as belonging to the data source layer) we checked again manually the two classes and we discovered that they contain code from Java 1.5 (templates) and, unfortunately, at the moment the MEMORIA meta-model can store only design information for systems written in Java 1.4 or previous versions. This is the reason that the two classes are missing from the model and, consequently, from the data source layer too.

5.3. Capturing the accesses between the object-oriented and the relational paradigms

The goal of this section is to answer the last two questions formulated at the beginning of this chapter regarding the reliability of the introduced approach.

At this phase, for each enterprise application, we performed a manual investigation in order to find out all the accesses from the classes belonging to the data source layer to the tables from the relational database and we compared the results with the ones provided by the introduced model of the applications. The goal of the comparison is, like in the previous case, to discover if there are false positives (*i.e.*, accesses from a class to tables erroneously identified by the approach) and false negatives (*i.e.*, accesses from a class to tables which were not identified by the approach) among the accesses from classes to tables.

Within the **TRS** application we did not find any false positive and negative regarding the accesses from classes to tables. This applies also to the *accessed-by* relations captured by the model and stored into the *TableSchema* class.

Within the **KITTA** application we found several false negatives and a false positive. Analyzing again manually the accesses from the classes which are on the list of false negatives we discovered that those classes access as embedded strings tables which do not exist in the relational database of the application. The introduced extractor of design information from enterprise application retains only the accesses to existing tables and, consequently, in the model we will not find accesses to tables which are not part of the involved database. Regarding the false positive (an access to a table from a class which does not appear in the source code) we discover that it is in the model due to the fact that the access is present in a comment from a method of the class. But this is not a serious problem because removing comments from the source code is a trivial task (for example, the approach presented in [17] removes comments before searching for duplications in the source code).

Within the **SALARY** application we found also several false negatives which are due to the reason as within the **KITTA** enterprise application.

Conclusive Remarks. In order to validate the reliability of our model for representing enterprise applications we performed several experiments on three case-studies. The performed experiments

- brings in front one of the major problem of enterprise applications: the source code of enterprise application might contain sql statements embedded as strings which access entities from the relational database which do not exist, most of the time these errors being discovered at runtime. With a minor en-

hancement, our approach would allow to discover the aforementioned type of accesses.

- reveals that in order to minimize the false positives in the model it is mandatory to remove the comments from the source code before constructing the system's model.
- shows that it is possible to obtain the model of an enterprise application written in Java 1.5, but those entities that use Java 1.5 facilities (*e.g.*, templates) will not be captured by the model. This is a limitation of the MEMORIA meta-model for representing object-oriented systems, a limitation which is a temporary one.

Although the size of the analyzed systems makes them appealing, as it allowed us to inspect them also manually the systems are too small and it is mandatory to conduct further case-studies in order to validate the scalability of the introduced approach.

6. Related Work

We dedicate this section to several representative solutions that fall in (or are closely related with) the process of reverse engineering for enterprise applications.

Increasing the level of understanding in software systems by identifying features³, locations in the source code that implement a specific functional or nonfunctional request, is not a new technique. In [8] some case studies which conduct to the following result are presented: Feature location is also needed for the maintenance of object-oriented code and object-oriented structuring does not facilitate feature location in the source code [8]. Consequently, there is a need for feature location methodologies suitable for object-oriented systems. In [14] is proposed a methodology for finding and describing concerns using structural program dependencies. FEAST, the tool that implements the proposed methodology, uses a structural program model which consists of three types of entities: classes, fields and methods. This structural model allows locating, for example, fragments in the source code that handle Java anonymous classes. But locating fragments in the source code that, for example, insert a row in a specific table is not possible due to the lack of information from the meta-model. Thus, our approach makes possible the identification of a complementary set of features specific for enterprise applications.

When we extracted the design information specific to the relational part of the application we presumed that the information regarding semantics of attributes, primary keys and foreign keys in the database tables is complete [3] and can be derived directly from the database tables. Recently, Yeh

³ also known as concepts [8], concerns [14].

and Li state in [18] that sometimes this may not be the case and they propose an approach where various procedures, like field comparison, data analysis, code analysis are applied in order to determine the semantics of attributes, followed by the identifications of primary keys, foreign keys and cardinality constraints.

A tool specifically designed for database reengineering is proposed in [2]. The proposed approach builds an instance of *Database*, a class containing a meta-model for representing relational databases with no dependence of the vendor. The main difference between this approach and our approach resides in the fact that our approach provides a meta-model for representing relational databases and also the interactions between an object-oriented programming language (at this moment, Java) and the databases.

7. Conclusions. Future Work

This paper is a step forward in modeling enterprise applications by taking into account their particularities that make them distinct from regular object-oriented systems. The introduced meta-model for representing enterprise application helps the reverse engineer by providing him with additional design information which is not present in a meta-model which captures design entities from regular object-oriented systems. The proposed meta-model is also used for assessing the design quality of enterprise applications [9].

In the future, we will focus our work on the next fronts:

- We intend to continue the evaluation of the introduced approach against other enterprise applications.
- Based on the introduced meta-model, we intend to define and automatize a comprehensive suite of specific quality assessment analyses for enterprise applications (e.g., analyses related to the duplication of SQL statements within the object-oriented source code).
- We intend to extend the applicability of the approach among enterprise applications written in other object-oriented programming languages (e.g., C++, C#).

Acknowledgments This work is supported by the Romanian Ministry of Education and Research under Project CNCISIS-TD No. 2739/19.05.2006.

References

- [1] E.J. Chikofsky and J.H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [2] I. de Guzman, M. Polo, and M. Piattini. An integrated environment for reengineering. In *Proc. IEEE International Conference on Software Maintenance*, 2005.
- [3] M.L. Pedro de Jesus and P.M.A. Sousa. Selection of reverse engineering methods for relational databases. In *Proc. European Conference on Software Maintenance and Reengineering*, 1999.
- [4] K. Brown et al. *Enterprise Java Programming with IBM Websphere*. Addison-Wesley, 2001.
- [5] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- [6] R. Kazman, S.G. Woods, and S.J. Carriere. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proc. IEEE Working Conference on Reverse Engineering*, 1998.
- [7] R. Koschke, J.F. Girard, and M. Wurthner. An intermediate representation for reverse engineering analyses. In *Proc. IEEE Working Conference on Reverse Engineering*, 1998.
- [8] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Proc. International Workshop on Program Comprehension*, 2005.
- [9] C. Marinescu. Identification of design roles for the assessment of design quality in enterprise applications. In *Proc. IEEE International Conference on Program Comprehension*, 2006.
- [10] C. Marinescu, R. Marinescu, P.F. Mihancea, D. Rațiu, and R. Wetzel. iplasma: An integrated platform for quality assessment of object-oriented design. In *Proc. IEEE International Conference on Software Maintenance (Industrial and Tool Volume)*, 2005.
- [11] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proc. IEEE International Conference on Software Maintenance*, 2004.
- [12] D. Rațiu. *Memoria: A Unified Meta-Model for Java and C++*. Master Thesis, Politehnica University of Timișoara, 2004.
- [13] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw Hill, second edition, 2002.
- [14] M.P. Robillard and G.C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. International Conference on Software Engineering*, 2002.
- [15] S. Rugaber and L.M. Wills. Creating a research infrastructure for reengineering. In *Proc. IEEE Working Conference on Reverse Engineering*, 1996.
- [16] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, Institute of Informatics and Applied Mathematics, University of Bern, 2001.
- [17] R. Wetzel and R. Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Proc. International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASCO5)*, 2005.
- [18] D. Yeh and Y. Li. Extracting entity relationship diagram from a table-based legacy database. In *Proc. European Conference on Software Maintenance and Reengineering*, 2005.