ENORM: An Essential Notation for Object-Relational Mapping

Alexandre Torres
Instituto de Informática, UFRGS
Porto Alegre, Brazil
atorres@inf.ufrgs.br

Renata Galante
Instituto de Informática, UFRGS
Porto Alegre, Brazil
galante@inf.ufrgs.br

Marcelo Pimenta
Instituto de Informática, UFRGS
Porto Alegre, Brazil
mpimenta@inf.ufrgs.br

ABSTRACT

Despite the growing adoption of object-relational mapping frameworks, UML and its most widespread extensions do not represent these mappings in a platform independent way. Maintaining mappings scattered in the code is difficult and error prone, specially if the schema is large and serves several systems. This paper proposes ENORM, a notation that extends class models representing all the essential mappings. ENORM is platform independent, providing a meta-model based on design patterns employed by three frameworks of Java, Ruby, and Python languages. An empirical evaluation indicates that ENORM performs well in comparison to separated models.

1. INTRODUCTION

Relational databases (RDB) are the backbone of information systems, and nobody knows when (or if) this will change [3]. However, the Impedance Mismatch Problem (IMP) continues to haunt object oriented designs that tend to underestimate the Object-Relational Mapping (ORM) difficulties.

In the past decade we saw a growing adoption of ORM frameworks by information system developers of distinct platforms such as Java, C#, Python, and Ruby on Rails. These frameworks have most of their resources based upon established patterns [6, 11, 14], and its use spread a more standardized approach for the IMP. Nevertheless, mappings scattered in the code, annotations and/or XML files are difficult to read, understand, and reason about changes.

The Model Driven Architecture (MDA) proposes that models take on the main role on the system development process [4, 17]. For an effective MDA approach, the information represented by models should be coherent, integrated, and computable, so that automatic transformations could turn models into executable system [16]. The UML notation lacks a specific notation for persistence, or to map classes to database. The absence of mapping information poses a challenge for developing transformations.

This paper presents ENORM, a general purpose notation that represents the essential structural concepts of ORM by extending the UML class model with a profile, and offering a concise set of new visual elements specific for ORM designs. These essential concepts are based upon persistence patterns adopted by distinct ORM frameworks in the market. The goal of ENORM is to facilitate the design by the clear application of ORM patterns, document mappings with a platform independent notation, and be a repository for MDA transformations and code generation.

The focus of ENORM is designing with structural patterns within a domain modeling logic, with objects of the domain incorporating both behavior and data [11]. ENORM does not encompasses the design of queries or the use of dynamic diagrams.

A controlled experiment was performed to evaluate modeling using ENORM. The results indicates that using only models, ENORM has a lower mean of missed goals than separated models.

This paper is organized as follows: Section 2 presents related works; Section 3 presents the notation; Section 4 presents the meta-model; Section 5 presents examples using ORM tools; Section 6 presents limitations and special cases; Section 7 summarizes empirical evaluation; and section 8 has the conclusions.

2. RELATED WORK

The agile database modeling [1] is a well known proposal for database modeling using UML extensions. It is mainly based upon the class diagrams for representing data models with a set of stereotypes. The Object Management Group (OMG) has also an underway proposal for data modeling representation [18]. None of the two notations have the focus on ORM, ORM frameworks or patterns.

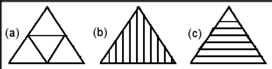
The Entity-Framework proposes the EDM model based on the EER notation [5]. EDM is focused on multipurpose conceptual modeling for distinct persistence mechanisms using the .NET platform. ENORM takes a distinct approach by encompassing general ORM design patterns, in a cross-platform way.

On a previous work, we proposed a notation based upon the Java Persistence API (*JPA*) standard named MD-JPA [20]. Although *JPA* is a standard, it is focused at the Java platform, including many concepts particular only to Java. It was not clear at that time what concepts are particular for *JPA*, and what was missing from other frameworks and platforms.

3. ESSENTIAL NOTATION (ENORM)

The notation here proposed is a lightweight UML profile, represented by a set of graphical extensions for class models, encompassing the essential structural concepts of ORM. ENORM was designed to be easily understood by developers and rich enough for MDA tools, allowing the specification of the relevant persistence details or hiding what can be inferred.

Table 1. New visual elements and their meaning

Graphical Element	Description
class || *table1, table2, …*	Persistent class. Optional tables
property || *column def*	Definition of the column mapping
«PK»	Part of the primary key
«Embed»	Dependent or embedded
	Inheritance types: (a) Flat, (b) Vertical or (c) Horizontal. Discriminators can be specified.
join table = *table*	Association table name
join columns = *col1, col2, …*	Columns that implement association
«Override» *property path* || *column def*	Override an inherited or embedded property or association mapping
property or assoc. end θ	Transient feature should not be mapped
«Map»	Set key property of qualified associations

ENORM elements (Table 1) are derived from ORM patterns following the domain model pattern [11]. Besides, ENORM reflects common practices of various ORM frameworks, such as *activerecord* for *Ruby (AR)*, *JPA*, and *SQLAlchemy* (SA) for Python [2, 12, 19].

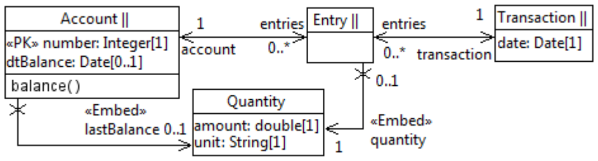


Figure 1: Simple Transaction example

A *Persistent* class (marked with "||") represents a class implemented as an *Active Record*, *Data Mapper,* or mapped in such a way by a framework. The class is persisted by a table with the same name; or one or more specified tables. Each property of a persistent class maps to a column, that can be detailed in the model. Associations between persistent classes are implemented with Foreign Keys (FKs) detailed by join columns and tables. Inheritance can be *flat* for single table pattern; *vertical*, for joined table pattern; or *horizontal* for the concrete table pattern. Non persistent classes can be persisted by associations marked as *embed* within *persistent* classes. A persistent class can have transient properties by using the transient symbol.

3.1. A simple example

Figure 1 shows a simple design for the *Accounting* patterns [10]. *Account*, *Entry*, and *Transaction* are persistent classes, each persisted by tables with the same name. *Account* has a meaningful Primary Key (PK) named *number*. *Entry* and *Transaction* will also have PKs, but they are not specified (inferred).

Quantity is not persistent and does not correspond to a table. However, each *Entry* instance refers to a *Quantity* with the *Embed* stereotype. Since the upper multiplicity is one, quantity association is persisted along the *Entry* table, by columns *amount* and *unit*. *Quantity* is similarly embedded by *Account*.

Finally, the associations between persistent classes are mapped as FKs connecting the PKs of each table. *Entry* will have a column referencing account *number* and a column referencing the PK of Transaction.

3.2. A not so simple example

Database information systems usually refer to centralized databases serving multiple systems, that must adapt to the existing schema. Often that means a break between nomenclature used by the system and the database, and a more complicated mapping.

Figure 2 introduces the *SummaryAccount* class, that aggregates accounts implementing multiple summary accounts [10]. Each account can be part of one or more summary accounts, and the *entries* of the summary are the union of all underlying *DetailAccount* instances.

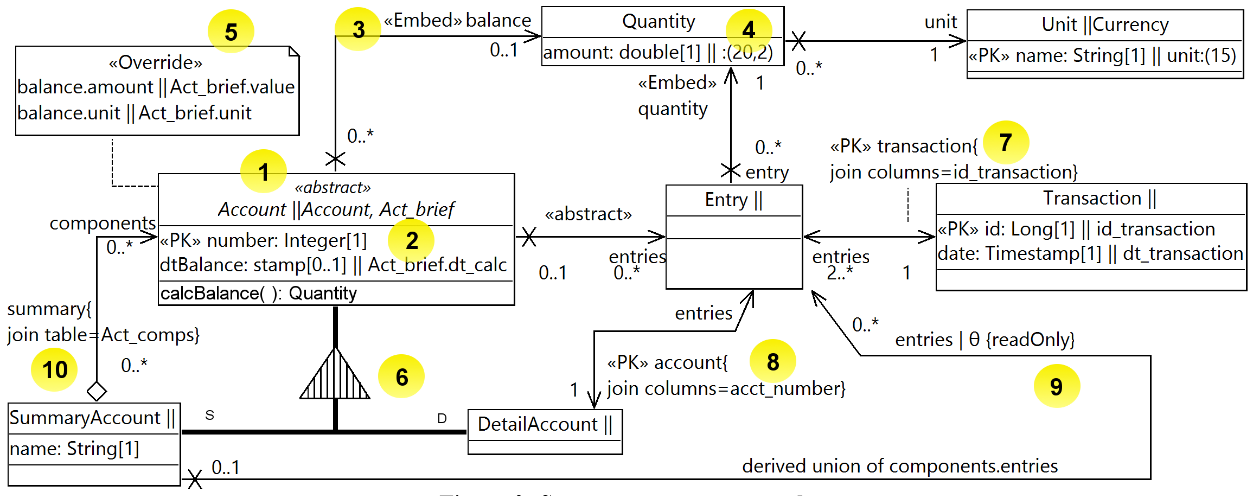


Figure 2: Summary account example

The *Unit* class now replaces the free text unit property. Several changes were introduced in the mapping, and Figure 3 presents the database derived from the model:

1. *Account* is mapped to two tables joined by the PK. The table *Act_brief* has an FK to *Account*.

2. Property *dtBalance* is mapped to column *dt_calc* on table *Act_brief*.

3. *Quantity* now refers to a *Unit* persisted by the *Currency* table. When *Account* references a *Quantity*, it stores a reference (FK) to the *Currency* table.

4. Property *amount* with default SQL precision/scale of *(20,2)*.

5. *Account* overrides the *quantity*: *amount* is persisted by the column *value* of table *Act_brief*; the association end *unit* is stored by the column *unit* in table *Act_brief*, that references the table *Currency*. By default, all columns would be stored along the primary table *Account*.

6. The account inheritance tree is persisted with the joined table pattern. Each class has its own tables, and each PK of the specializations refers to the *Account* PK. The discriminator column can assume 'S' or 'D'.

7-8. *Entry* refers to *Transaction* with a column named *id_transaction*, and refers to *DetailAccount* with a column named *acct_number*, setting the PK of *Entry*.

9. *Account* defines the association *entries* as *abstract*. *DetailAccount* implements *entries* by an FK, but for *SummaryAccount* this association is derived from its components. The transient symbol tells that this association should not be stored by an FK column.

10. The *components* association is many-to-many, and therefore is mapped by an association table. The *join table* specifies that this table is *Acct_Comps*. By default it will have FK columns referring to *Account* and *SummaryAccount*.

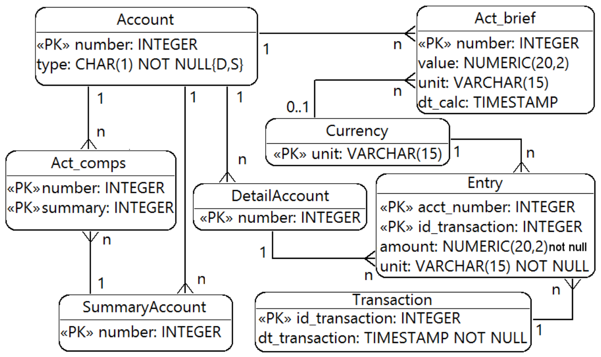


Figure 3: Database model of account example

3.3. Maps

UML allows the specification of *Qualified Associations* that represents partitions in the association between two classes. When the qualified property has an upper value of one, the association represents what is commonly referred as *Map* or *Dictionary* by object-oriented languages [22].

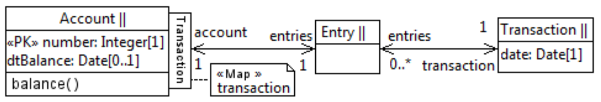


Figure 4: Map with key reference

Figure 4 presents an example where the association end of *Account* is a map with a <*Transaction, Entry*> form, where the qualified variable of type *Transaction* is the key. The *Map* stereotype allows the specification that the key is in fact the transaction property of entry, what is common on ORM. The goal is that when the user adds a pair <*tx*, *ey*> to the map, it will associate *ey* both with the account and the *tx* transaction.

The property key can also be user defined, derived from a complex operation. In such cases, it can be a read-only map. Qualified associations without a property key are also allowed. In the transaction example, the map would be persisted in a separate many-to-many table, instead of using the association between entries and transactions. Qualifier properties can also assume non-persistent and scalar types.

4. ENORM METAMODEL

Backing up the visual notation there is a profile providing compatibility between ENORM and UML implementations, such as the Eclipse UML2 package. Figure 5 summarizes the stereotypes, the extended UML elements, meta-classes, and its properties and relationships detailed by this section.

The *Persistent* Stereotype is applied to a class marking the class with the double bars (||) of Table 1. The *source* property allows the direct definition of one *Table*, a reference to an already defined table by *TableRef*, or a *JoinedSource* comprising two or more tables connected by *JoinColumn* objects. The use of *Table* or *TableRef* determines the class that "owns" the table definition, preventing duplicate specification of tables. If *source* is unspecified, the class is persisted by a table with the same name of the class.

Properties owned by a persistent class are, by default, persisted, and scalar values are stored as columns. The *ColumnMapping* Stereotype allows the definition of these columns, informing column name, if it accept nulls, length, precision, scale, unique constraint, database type and so on. The column can be owned by a *Table*, but the table may be inferred if the *Persistent* class does not define a table, or if the class is not persistent. Again a *ColumnDefinition* can be a *Column* owned by the property or a *ColumnRef* that references a *Column*. A property without mapping will have a column with an inferred definition.

The *Embedded* stereotype is applied to association ends or simple properties whose types are not persistent classes. This means that this class is persisted as a dependent table (if to-many) or embedded in the table (if to-one). Properties of non persistent classes can have the *ColumnMapping* stereotype applied in order to specify how is its preferred way of being persisted, such as *length*, *precision* and so on. These definitions will not have a table.

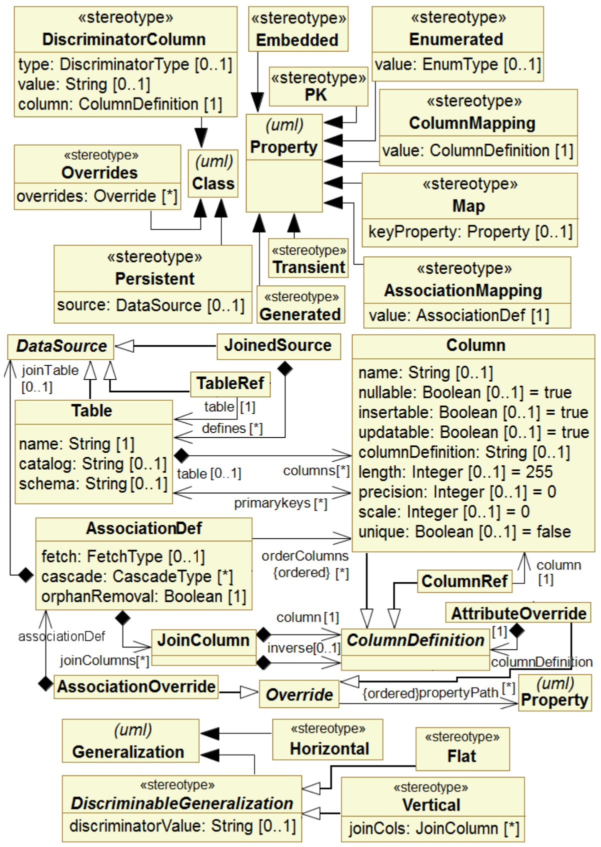


Figure 5: Profile of ENORM

The *AssociationMapping* stereotype allows the definition of mapping details for one association by the application in one of the association ends. The *AssociationDef* class allows the definition of fetch strategies, cascade delete, orphan removal policy, columns used by an order by clause, join columns, and a join table. The *JoinColumn* class defines the FK column in the detail side, and optionally the corresponding PK in the master side (for multiple PK, or ad-hoc joins). The *joinTable* is usually defined on many-to-many situations to specify the table(s) that implement the relationship.

The *PK* stereotype marks a property as part of the PK of some persistent class. It can be applied on association ends, meaning that the FK(s) columns are also part of the PK. *PK* can be combined with *ColumnMapping*, *AssociationMapping* and so on. *Generated* marks a column with generated values.

Horizontal, *Flat*, and *Vertical* stereotypes can be applied to a generalization to specify which pattern will be used to emulate inheritance on the database. With *Flat*, all columns necessary to represent the inheritance tree are stored in the same table. Usually, the instance type is determined by a discriminator column, that can be defined by applying the *DiscriminatorColumn* stereotype at the general class, and filling the property *discriminatorValue* for each generalization with the *Flat* application.

The *Vertical* stereotype stores each class along its properties in a distinct table, that is by default joined by a common PK. It is possible to specify what columns perform the join by the *joinCols* property. It is also possible to define a discriminator. Finally the *Horizontal* stereotype stores each concrete class independently, and the origin table determines the type.

A class may specify an inheritance pattern even when it inherits from a non persistent class. In this situation (and only this), the properties and associations of the general class will be persisted along the persistent specializations. The *Overrides* stereotype allows a class to override such properties (*Attribute Override*) and associations (*Association Override*), defining the *columns*, *join columns*, *join tables*, among other details.

A class may also override properties and associations of embedded/dependent classes. The tricky part here is that one class can embed a class that embed another class. The property path of embedded overrides is stored by the ordered association *propertyPath*. In the example of Figure 2 the path "*balance.amount*" refers to the sequence {*Account.balance, Quantity.amount*}. This allows the override to differentiate when the class has more than one relationship to the same class.

The *Enumerated* stereotype allows the definition of how enumerations are mapped (string or ordinal values). The *Transient* stereotype marks a property or association end to be ignored on persistence mapping.

5. ENORM AND ORM FRAMEWORKS

The way *JPA*, *SA*, and *AR* implements each ORM pattern is distinct. *AR* separates database from class definitions on migration files, where each table is specified with its columns and references. *JPA*, in the other hand, infers much of the database structure from annotations placed before each class (or XML), but does not have a central place where the database is defined. In the middle ground, *SA* allows the definition of tables, classes, and its mappings separately (classical) or together (declarative), but the table definitions are clearly separated at *runtime*.

JPA advanced a lot in the field of embedded and dependent mapping, providing several resources to automate complex collections of elements and embedded classes. *SA* has a simple mechanism called *composites* that deals with embedded objects, but not with dependent objects. *AR* also has a similar mechanism named *composed_of*. *JPA* allows the partial mapping of plain classes (such as *Quantity*) and a later override by the container classes. *SA* and *AR* does not have this resource.

SA and *JPA* supports all three inheritance patterns. AR only supports the *Flat* strategy, and other strategies can at best be emulated with a simple relationship.

SA allows the definition of queries based on polymorphism for inheritance or class mapping. *JPA* relies on one-to-one relationship between tables for mappings and multi-table inheritance. *AR* does not allows a class mapped to multiple tables. The implementation of *Account,* with *AR, JPA*, and *SA*, is available at the web site of our modeling tool [9].

6. LIMITATIONS AND SPECIAL CASES

This section enumerates some known limitations and special use cases of ENORM.

- **Flexible data sources.** Currently, the profile only supports the mapping of one class to many tables if each table has a one-to-one relationship to the first table. This is an easy way to specify the data source without caring about checking how a complex mapping would be persisted. A more flexible rule for data sources would be equivalent to a side effect free *updatable view* [7].
- **Qualified associations.** Qualified associations can have more than one qualifier properties. This kind of construct would need keys with *tuples* of objects, what can be quite complicated to implement using ORM tools. Qualified properties with upper cardinality over one is a special case, representing a map of collection elements, where each key can have more than one associated value.
- **Multiple Inheritance, multiple types.** The profile does not include resources to deal with the persistent specialization of more than one persistent class, and the resulting mapping would be unknown. However, a class can specialize any number of other classes as long as it only inherits persistent information from one tree branch. Single relation with multiple type attributes [8] was not included in ENORM.
- **Association class and "n-ary".** The profile does not have any specific mapping for the *Association Class* element of UML, it is as any other class. ENORM does not yet support persistent associations with more than two classes. These associations must be separated on binary associations.
- **Generics and Template parameters.** Mechanisms such as generics can be specified using template parameters on UML [20], and they are useful for strong typed languages such as Java and C#. We did not identify any additional extension necessary to the use of template parameters.

7. EMPIRICAL EVALUATION

The goal of the empirical evaluation is to check if ENORM had a greater rate of success in the activity of changing models, regardless of any impact related to implementation. Changing models was our choice because it is more common than creating new models, and captures both comprehension and application of the notation.

Controlled experiments comparing the use of ENORM and separated UML/Relational models were performed to test our hypothesis. In this paper we summarizes the results of an experiment performed in 2012 with 69 students[1].

The tasks were designed as modeling activities, showing models based upon Analysis Patterns and asking the participants to apply a set of modifications, creating an output model. These models and instructions were extracted from the Analysis Pattern literature [10], in order to reduce the artificiality of the tasks, and augmented with ORM details. Each task was as objective as possible, avoiding misinterpretations. One of the tasks was similar to the evolution of the *accounting* models (Figures 1 and 2), the other tasks related to *accountability* and *planning domains*.

The subjects were senior undergraduate students and graduate students, selected among those already approved on the basic database, object oriented development, and software engineering courses. Each participant received a training in the format of a tutorial with videos, and a small scale task just like the experiment itself.

The experiment had a *within-subjects* design, in which each treatment was applied to each subject, and the starting order was randomized (*counterbalancing*) so that the same number of subjects started with each treatment [13, 15]. The treatment (*method*) is the main independent variable assuming ***A*** (not using ENORM) or ***B*** (using only ENORM).

The dependent variable is the number or missed goals (*misses*) based on expected model. The time to execute each task was fixed due to external constraints, and was not evaluated in this experiment.

Other factors were controlled as follows: both hardware and software used on the experiment was the same to all participants; a specially developed modeling tool was employed to guarantee a similar environment, and detect the number of missed goals.

The Analysis of Variance (ANOVA) was employed to compare the treatments, making it possible to verify the residual effect in the *sequence* of activities. In other words, it checks if there are significant difference in the sequences *AB* or *BA*, an indication of learning effect.

[1] Full technical report available at [21].

Table 2 presents the least square means of *misses,* and *p-value* results analyzing *method* and *sequence (seq.)* on each of the four tasks (*T*). The other variables are the number of participants (*P*), and time in minutes available to perform each task (*Tim*).

Table 2. ANOVA results, per task.

T	P	Tim	P-value		Misses	
			Method	Seq.	A	B
1	69	10	<0.001	0.56	16.6	11.6
2	69	20	<0.001	0.42	11.2	6.5
3	35	20	0.006	0.29	13.1	10.7
4	35	23	<0.001	0.25	7.5	2.4

Assuming results as statistically relevant at $\alpha = 0.05$, there is a significant difference ($p<0.05$) between *methods **A*** and ***B***, with *method **B*** presenting a lower mean of misses at all tasks. The sample evidence does not confirm the presence of residual effect, given the absence of statistical significance for the effect of *sequence* ($p>0.05$ at all tasks).

8. CONCLUSION

Despite the growing popularization of ORM patterns by the adoption of persistence frameworks, the mappings between objects and database are dispersed in the code. Distinct frameworks employ distinct ways of presenting these mappings, despite following the same patterns.

This paper proposes ENORM, a new notation implemented as an extension of UML class models that allows the design of database based systems, providing the essential patterns of ORM in a platform independent way. ENORM unifies classes and mappings focused on the structural aspects of persistence, with the necessary detail for MDA tools. Our controlled experiment indicated that ENORM had a lower mean of missed goals when improving models, in comparison to separated models.

9. REFERENCES

[1] A UML Profile for Data Modeling: 2003. *http://www.agiledata.org/essays/umlDataModelingProfile.html*. Accessed: 2013-10-01.

[2] Active Record - Object-relation mapping put on rails: 2012. *http://ar.rubyonrails.org/*. Accessed: 2013-10-01.

[3] Atzeni, P. et al. 2013. The relational model is dead, SQL is dead, and I don't feel so good myself. *SIGMOD Rec.* 42, 1 (Jul. 2013), 64–68.

[4] Beydeda, S. et al. 2005. *Model-Driven Software Development*. Springer.

[5] Blakeley, J.A. et al. 2006. The ADO.NET entity framework: making the conceptual level real. *SIGMOD Rec.* 35, 4 (Dec. 2006), 32–39.

[6] Brown, K. and Whitenack, B.G. 1996. Crossing Chasms: a pattern language for object-RDBMS integration: the static patterns. *Pattern languages of program design 2*. Addison-Wesley Longman Publishing Co., Inc. 227–238.

[7] Dayal, U. and Bernstein, P.A. 1982. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.* 7, 3 (Sep. 1982), 381–416.

[8] Elmasri, R. and Navathe, S.B. 2003. *Fundamentals of Database Systems*. Addison Wesley.

[9] Essential ORM Modeler: 2013. *http://sourceforge.net/projects/eorm/*. Accessed: 2013-12-05.

[10] Fowler, M. 1996. *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional.

[11] Fowler, M. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc.

[12] JSR-000317 Java Persistence 2.0 - Final Release: 2009. *http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html*. Accessed: 2013-10-01.

[13] Juristo, N. and Moreno, A.M. 2001. *Basics of Software Engineering Experimentation*. Springer.

[14] Keller, W. 1997. Mapping Objects to Tables - A Pattern Language. *Proceedings of the 1997 European Pattern Languages of Programming Conference* (Irrsee, Germany, 1997).

[15] Ko, A.J. et al. 2013. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*. (Sep. 2013), 1–32.

[16] Mellor, S.J. et al. 2004. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley.

[17] OMG 2001. OMG's Model Driven Architecture.

[18] OMG 2005. Request For Proposal Information Management Metamodel (IMM).

[19] SQLAlchemy - The Database Toolkit for Python: 2012. *http://www.sqlalchemy.org/*. Accessed: 2013-10-01.

[20] Torres, A. et al. 2011. A synergistic model-driven approach for persistence modeling with UML. *Journal of Systems and Software*. 84, 6 (Jun. 2011), 942–957.

[21] Torres, A. et al. 2013. Technical Report - Comparing ENORM and separated modeling using Relational and UML class models: a within-subjects experimental study. *http://www.inf.ufrgs.br/~atorres/sigmod2013/*.

[22] UML 2.4.1 Superstructure: 2011. *http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/*. Accessed: 2013-10-01.