Information and Software Technology 82 (2017) 1–18

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

# Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design

Alexandre Torres[a,*], Renata Galante[a], Marcelo S. Pimenta[a], Alexandre Jonatan B. Martins[b]

[a] Instituto de Informática - Universidade Federal do Rio Grande do Sul (UFRGS), Av. Bento Gonçalves, 9500, Porto Alegre, RS, 91501-970, Brazil
[b] TAGMATEC, Rodovia SC-401, 600, ParqTEC Alfa, CE Alfama, sala 405, Florianópolis, SC 88030-911, Brazil

ARTICLE INFO

*Article history:*
Received 30 December 2015
Revised 23 September 2016
Accepted 27 September 2016
Available online 28 September 2016

*Keywords:*
Object-relational mapping
Design patterns
Impedance mismatch problem
Enterprise patterns
Class models

ABSTRACT

*Context:* Almost twenty years after the first release of TopLink for Java, Object-Relational Mapping Solutions (ORMSs) are available at every popular development platform, providing useful tools for developers to deal with the impedance mismatch problem. However, no matter how ubiquitous these solutions are, this essential problem remains as challenging as ever. Different solutions, each with a particular vocabulary, are difficult to learn, and make the impedance problem looks deceptively simpler than it really is.

*Objective:* The objective of this paper is to identify, discuss, and organize the knowledge concerning ORMSs, helping designers towards making better informed decisions about designing and implementing their models, focusing at the static view of persistence mapping.

*Method:* This paper presents a survey with nine ORMSs, selected from the top ten development platforms in popularity. Each ORMS was assessed, by documentation review and experience, in relation to architectural and structural patterns, selected from literature, and its characteristics and implementation options, including platform specific particularities.

*Results:* We found out that all studied ORMSs followed architectural and structural patterns in the literature, but often with distinct nomenclature, and some singularities. Many decisions, depending on how patterns are implemented and configured, affect how class models should be adapted, in order to create practical mappings to the database.

*Conclusion:* This survey identified what structural patterns each ORMS followed, highlighting major structural decisions a designer must take, and its consequences, in order to turn analysis models into object oriented systems. It also offers a pattern based set of characteristics that developers can use as a baseline to make their own assessments of ORMSs.



## 1. Introduction

Relational Databases (RDBs) and Object-Oriented Programming Languages (OOPLs) are based upon distinct paradigms, containing technical, conceptual, and cultural incompatibilities. The obstacles of dealing with such incompatibilities are commonly referred to as the object-relational Impedance Mismatch Problem (IMP) [1–3].

If system analysis aims at the recognition of the business problems, sketching platform independent solutions, such as identifying and applying analysis patterns [4], the design activity is focused in fitting the analysis to the limitations of the chosen platform. At the design, the IMP can turn analysis models into something difficult to understand, maintain, evolve, and even track back to the original idea.

For example, two basic concepts on OO models are *inheritance* and *many-to-many* relationships, both extensively used on analysis models, such as analysis patterns [4], and both are absent on RDBs. Primary and foreign keys are important concepts for good database design, and both are absent on OOPLs. Bridging these conceptual mismatches, without losing the tracking among all artifacts, is the challenge of the IMP.

In many projects it is possible to avoid the IMP, by not adhering to Object-Oriented (OO) practices, or not using a RDB. Sometimes it is feasible to place all domain logic within stored procedures. Another way is using *NoSQL* persistence, such as Document-Oriented Databases, and deal with other kind of mis-

* Corresponding author.
*E-mail address:* atorres@inf.ufrgs.br (A. Torres).

http://dx.doi.org/10.1016/j.infsof.2016.09.009
0950-5849/

matches related to Object-Document Mapping [5]. Nevertheless, relational databases (RDBs) continue to be the backbone of information systems, and nobody knows if this will ever change [6].

From the developer standpoint, Object-Relational Mapping (ORM) encompasses solutions for mapping business objects to relational data, by separating persistence concerns on a *persistence layer* [7]. Contrary to popular belief, ORM is not trivial nor fully automatic: mappings are a kind of *updatable view*, and its automation is undecidable [8]. Moreover, ORM relies on experienced developers being able to trade-off between database design concerns, such as data normalization, primary keys, and relationships, and the forces of OO design, such as pursuing high cohesion and low coupling [3,9].

Design patterns are a great way to organize knowledge and teach good practices, and this also applies to ORM patterns [7,10–12]. For quite some time, Object-Relational Mapping Solutions (ORMSs) were expensive and limited, and these patterns were an important reference when developing the ORM from scratch. Nowadays ORMSs are ubiquitous: every OO platform has at least one ORMS, sometimes with an "official" API. This popularization made ORM looks simpler than it really is [13].

Each ORMS has a different terminology, with different names to concepts that have similar, or even the *same,* meaning. For example, *embedded values* are also referred as *complex types, composite columns,* and *aggregated values,* all of them representing the same basic pattern. ORMSs named after patterns, such as *Data Mapper* and *Active Record*, in reality aggregates a combination of different pattern implementations, what may confuse developers about their propose and scope. Moreover, each OOPL has particularities that must be taken into account when dealing with ORM, such as if it has static or dynamic types, or if it supports multiple inheritance. This all contributes to categorizing IMP as the *Vietnam of computer science* [14].

This paper pursues answers to developers and designers about what compromises must be taken when dealing with the IMP, and how to assess the strengths and weakness of ORMSs. It presents a critical survey in the scenario of ORMSs, relating its characteristics with the established design/architectural patterns in the literature. In order to be relevant to the widest public, within a limited survey, the ORMSs were selected to represent the most popular OOPLs.

The scope of this study covers the characteristics of ORMSs related to the static view of modeling [15], including all data structure concerns, as well as the organization of operations on the data, describing behavioral entities as discrete modeling elements. This includes the presence of *Create, Read, Update and Delete* (CRUD) basic persistence operations, but not the design of their behavior; includes the way a relationship can be fetched and the impact of each decision, but not the way queries are designed to fetch related data. The design of queries in ORMSs, using SQL, or other proprietary object query languages, is out of the proposed scope. Internal implementation mechanisms of ORMSs were left out, such as *Unit of Work* and *Repository* patterns, because they are difficult to assess, and should not interfere with structural design.

The contribution of this paper is to provide a common terminology for the assessment of ORMSs, and a better understanding on ORM patterns and how they are applied across distinct platforms and frameworks. It summarizes a list of important decisions that a developer will have to make when using ORMSs. This information helps the designer in the anticipation of project limitations, the documentation of design decisions due to ORMS characteristics, and a better traceability between implementation artifacts and conceptual elements. This paper is not intended to point out what is the best of the ORMSs, but rather to help understand how to compare them.

The remaining of this paper is organized as follows: in Section 2 we discuss the related work that drives our study, with a brief historical introduction of the impedance mismatch problem; in Section 3 we present the methodology used in the survey; in Section 4 we present the survey of ORMSs relating patterns, implementations, and implications for the application design; in Section 5 we present conclusions and future work.

## 2. Historical background and related work

Initial research on the object-relational IMP may be traced back to the practical need of storing *Smalltalk* objects in persistent databases [1]. At that time, most studies were focused on the upcoming new generation of OO database systems, discussing issues later employed for ORM such as inheritance, associations, and polymorphism [16,17].

Patterns are recurring solutions identified by experience and documented as practical guides to software design [18]. In the 90s, due to the long takeoff of OO databases, impedance mismatch research shifted to ORM. The early impedance solutions were organized as an easy to access "buffet" of design patterns for system designers and developers. Patterns for each persistence layer component, and approach, covered several strategies to overcome the IMP [7,10,19].

*TopLink* released the first known commercial ORMS for *Smalltalk* in 1994, and two years later for *Java* [20]. The central patterns of interest in this paper (ORM patterns) are those focused on the mapping of objects to tables, since they establish the common ways of designing classes that represent database objects and *vice versa* [12].

In the twenty-first century the OO databases adoption was near stagnant [21]. The set of persistence patterns became part of a so called enterprise pattern set [11], as the impedance mismatch problem was recognized as much more than a technical concern, encompassing conceptual and cultural problems [3].

The dominance of OO languages and the inexpensive/free ORMSs disseminated the use of persistence layers and a *Domain Model* approach, where domain classes centralized behavior and data. Successful ORMSs such as *TopLink* (now *EclipseLink*) and *Hibernate* contributed to, and received contribution from, standards such as *JDO* and *JPA*, both influenced by the object data standard [22]. Failure in adopting ORMSs is often related to SQL and performance anti-patterns [23,24].

The impedance mismatch problem attracted some attention beyond the pattern scope in recent publications. Model management fits the IMP as one type of engineered mapping problem, solved by schema evolution operations [25]. Mappings are also subject of study for the general impedance mismatch case [26]. Both works deal with impedance mismatch beyond the OO domain, including XML, *Cobol* and Data warehousing mismatch problems.

Another classification for data mappings was proposed in a general top-down way: a conceptual framework based on concerns such as paradigm, language, schema and instance; each concern influencing the subsequent choices [27]. Our classification has a bottom-up approach, and deals with a smaller set of representative frameworks, focusing more on language, schema and instance options for ORMSs.

## 3. Methodology

This survey is based on bibliographic research, aiming at the relationships between the ORMS information and the pattern literature. The sources of ORMSs information are obtained at the descriptions, online manuals, published papers (when they exist), books, and practical experience found on the internet validated by the personal experiences of the authors. The ORM patterns were

**Table 1**
Language popularity rankings consulted.

| Ranking | URL |
|---|---|
| A. IEEE Spectrum's 2014 Ranking | http://spectrum.ieee.org/static/interactive-the-top-programming-languages |
| B. TIOBE Index for December 2015 | http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html |
| C. PYPL, December 2015 | http://pypl.github.io/PYPL.html |
| D. RedMonk, June 2015 | http://redmonk.com/sogrady/2015/07/01/language-rankings–6–15/ |

**Table 2**
Selected ORMSs, its platforms, and rankings.

| Language Ranking | | | | Platform | ORMS | Version | Docs Date | ORMS URL |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | | | | | |
| 1° | 1° | 1° | 2° | Java | JPA/Hibernate | 2.1/4.3.11 | 08/15 | http://www.hibernate.org |
| | | | | | MyBatis | 3.3.0 | 05/15 | http://www.mybatis.org |
| | | | | | Cayenne | 3.1 | 09/14 | http://cayenne.apache.org |
| 3° | 3° | 5° | 6° | C++ | ODB | 2.4.0 | 02/15 | http://www.codesynthesis.com/products/odb/ |
| 4° | 4° | 2° | 4° | Python | SQLAlchemy | 1.0.9 | 12/15 | http://www.sqlalchemy.org |
| 5° | 5° | 4° | 5° | C# | Entity Framework | 6.1.3 | 12/15 | http://www.asp.net/entity-framework |
| 15° | 7° | 13° | 19° | VB.net | | | | |
| 6° | 6° | 3° | 3° | PHP | Doctrine | 2.5.2 | 12/15 | http://www.doctrine-project.org/ |
| 7° | 8° | 7° | 1° | Javascript | Bookshelf.js | 0.9.1 | 10/15 | http://bookshelfjs.org |
| 8° | 10° | 12° | 7° | Ruby | ActiveRecord | 4.2.4 | 08/15 | http://ar.rubyonrails.org |
| 11° | 9° | 15° | 11° | Perl | | | | Ruled out |
| 2° | 2° | 6° | 9° | C | | | | Ruled out |
| 9° | 18° | 10° | 13° | R | | | | Ruled out |
| 10° | 16° | 11° | 17° | Matlab | | | | Ruled out |
| 16° | 15° | 8° | 10° | Objective-C | | | | Ruled out |
| – | 14° | 9° | 18° | Swift | | | | Ruled out |
| – | – | – | 8° | CSS | | | | Ruled out |

selected among the set organized by Fowler [11], although we also consider original patterns published in the 90s [7,10,12,28]. The examples discussed in this paper are extracted from the analysis patterns literature [4], discussing possible designs and implementations by applying ORM patterns and resources available from the ORMSs.

### 3.1. ORMS selection

The first step was to select a small, but representative set of ORMSs. The goal was to represent the most popular OOPLs, picking ORMS that are both popular and representative for our study. We consulted four language popularity rankings on the internet, as shown in the Table 1, aiming at the most popular OOPLs. It is difficult to assess popularity between ORMS, therefore we sought to choose a set that best represented the persistent patterns, and presented an active community in discussion forums, bug tracking, and maintenance releases in 2015.

Table 2 lists the platforms we selected from the rankings, with its respective popularity index in each ranking of Table 1, and the chosen ORMSs for this survey. Table 2 also includes version information and the last known date for the consulted documentation. The *JPA* standard [29] was the first obvious choice encompassing a number of ORMSs for the *Java* platform. The *JPA* implementation of choice for out tests was *Hibernate.* The *MyBatis* (an active fork of *iBatis*) is a well-known *Java* framework that does not use *metadata* to perform ORM [30].

The *ODB* is the ORMS chosen for *C++*, and it is not a framework, relying in code generation [31]. The *Entity Framework (EF)* [32] is both a major persistence layer and an example of *model based* ORM tool with configurable code generation. The Apache *Cayenne* is a model/generation based framework that allows some degree of organized customization for *Java* [33].

The *SQLAlchemy (SA)* is a well-known framework for the Python platform and presents a hybrid approach to the coupling problem [34]. The *Ruby ActiveRecord* (*RAR*) is the major ORMS for the *Ruby* platform [35]. *Doctrine* is an ORMS for *PHP* [36]. *Bookshelf (BS)* is a ORMS for *Javascript* running at the *Node.js* platform [37].

From the rankings we ruled out the *C, R,* and *CSS* languages because they were not OO; *Perl and Matlab* were at the top ten in only one ranking. *Objective-C,* and its successor, *Swift,* have the official *Core Data* framework, primarily aimed at mobile devices, with limited mappings to the *SQLite* database [38], and were also ruled out. VB.net was not selected by popularity, but the Entity Framework ORMS is also used by C#.

## 4. The survey

This survey is organized in sub-sections, each representing a common problem treated by one or more ORM patterns. For example, the *transparency and coupling* problem refers to how the data model is specified in relation to the ORMS, either following the *Active Record* or the *Data Mapper* pattern. The first two sub-sections discuss architectural patterns, and the following are focused at the structural patterns.

In order to organize the results in relation to the ORMS, each problem is taken as a criterion for our classification, and each criterion identifies one or more common characteristics relating ORM patterns and frameworks, such as what patterns are used or available, and what options are offered. For example, the *inheritance mapping* criterion discusses the problem of how inheritance can be mapped to relational databases. Each distinct pattern for the problem is a characteristic that may be available for each ORMS. At the end of the survey, Table 5 summarizes these findings.

Furthermore, while presenting the problems, patterns, and ORMSs implementations, we also identify key design questions related to creating object-relational mappings for analysis models. Each question is numbered with the notation *Q <number>*. Table 3 presents a summary of these questions, organized by UML element, such as models, classes, attributes, associations and inheritance, helping designers to document their decisions when mapping a model.

**Table 3**
Summary of design questions based upon an ORMS.

| Domain level | | What to observe/question during project design |
|---|---|---|
| Model | | Q1. If the framework has a visual data-model that controls persistence, how does it work with object-oriented domain models? Can we specify and implement operations? |
| Class | All domain classes | Q2. Is inheritance to ORM framework classes mandatory?<br>Q3. Is there some dependency to ORMS classes on domain classes? Can it be decoupled? |
| | If persistent | Q4. Which table(s) (or what SQL statements) are mapped to that class?<br>Q5. Is the identity mapping defined? With just one or many attributes?<br>Q6. How identity will be assigned? Will generation parameters for the identity be necessary? (such as table of ids, sequences, auto-columns…) |
| | If embeddable | Q7. Is it necessary to distinguish a class as embeddable? Is there a contract to follow?<br>Q8. Is it possible to define preferred mapping for attributes?<br>Q9. Is it used as identity for persistent classes? If so, should the class follow specific rules required by the ORMS?<br>Q10. How relationships *from* embeddable values to other domain classes work, if needed? |
| Attribute (persistent) | | Q11. Does it match a database column type or should it be an embeddable value?<br>Q12. If it is part of a composite Identity, does it represent an embeddable value class containing PK, or each attribute of the identity is an individual attribute of the class?<br>Q13. Type parameters, such as length and precision, were defined? Will Dates and LOBs need specific parametrization? Is the cardinality matching NULL/NOT NULL constraints? |
| Association (persistent) | zero/one-to-many | Q14. Should a collection type be defined? How to deal with element ordering?<br>Q15. Can/Should the fetch strategy be specified? |
| | many-to-zero/one bidirectional | Q16. Will the relationship attribute be loaded by a proxy?<br>Q17. The collection must have a reverse attribute, or collection that maintains, in the opposite class, a bidirectional relationship. Is it defined and documented? |
| | to-one | Q18. The FK that implements the relationship may be a class attribute or may be hidden by the ORMS. The relationship may be represented only by a reference to the related object. Is the relation Attribute-Collection-FK clear and well documented? |
| | many-to-many | Q19. Is a join table clearly defined, with FKs to the tables mapped to the related classes?<br>Q20. Do your ORMS transparently support Join Tables? In what cases a join table must be explicitly implemented as an association class? |
| | other | Q21. Maps and element collections are supported by few ORMSs. Can we use them? |
| Inheritance | | Q22. What strategy will be employed (among those available in the chosen framework)?<br>Q23. How to deal with inherited associations to other persistent classes?<br>Q24. Is a discriminator column necessary? Are the values documented?<br>Q25. Does the ORMS supports classes in an inheritance hierarchy with distinct PKs?<br>Q26. What are the consequences of my inheritance choices? Will it perform alright?<br>Q27. Does the ORMS supports mixing strategies? At what circumstances? |

### 4.1. Transparency and coupling on Active Record and Data Mappers

Coupling is a measure of interconnection among modules in a software. Software with low coupling levels tends to be easier to understand and less prone to the propagation of changes [39]. *Transparency* is commonly referred to as the ability to keep a *loose coupling* between application and the persistence library/framework, mainly by keeping the domain level classes unaware of the persistence framework [40]. *Transparency* is usually achieved by having ORM information isolated in *external configuration* files or annotations. The *transparency/coupling* criterion examines the problem of how the application is coupled to the ORMS.

We take the *Domain Model* pattern as our context and the *Active Record* pattern as the starting point for our discussion of an ORM solution [11]. Each class of the domain is responsible for retrieving and maintaining its data in the database and each instance of the domain class represents one row (or record) in the database. This frequently means implementing an interface of common CRUD [41] methods, or extending an abstract super class, responsible for encapsulating the common services among all persistent classes.

The *RAR* can be used as an example of *Active Record* implementation for ORM [35]. The persistence framework provides a base abstract class named *ActiveRecord::Base* that implements (or bridges) most of the SQL conversation with the database. By extending *ActiveRecord,* the classes of the domain will be automatically persisted according to the mapping contract defined within the framework.

Fig. 1 depicts an *Active Record* example on *Ruby* consisting of the *Account to Entry* relationship. Each class of the domain must

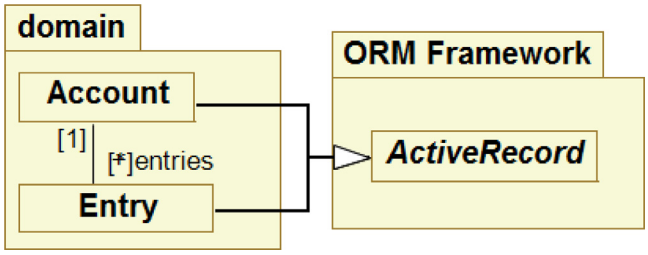


**Fig. 1.** Active Record example.

specialize *ActiveRecord::Base,* and optionally override its methods and properties, to represent persistent objects. *Ruby* is interpreted, dynamic typed, and highly reflective, characteristics that help with adopting conventions that represent mapping constructions, without losing the ability to override most of these conventions. For example, the Primary Key (PK) in *Ruby* is, by convention, a column named "ID", and the name of the table is the plural of the name of the class. However, the developer can override the super-class replacing the convention for one or more classes.

In order to allow further flexibility for the *Active Record*, one solution is to introduce an intermediary abstract class. The intermediary class generalizes each domain class, encapsulating the database access details. This intermediary class can then specialize the *ActiveRecord* base class that keeps the common database logic and acts as a *template method* class for the domain [18].

It is common to have the intermediary classes automatically generated by the framework, from models or configuration files for instance. Fig. 2 presents the *Account* domain example in this

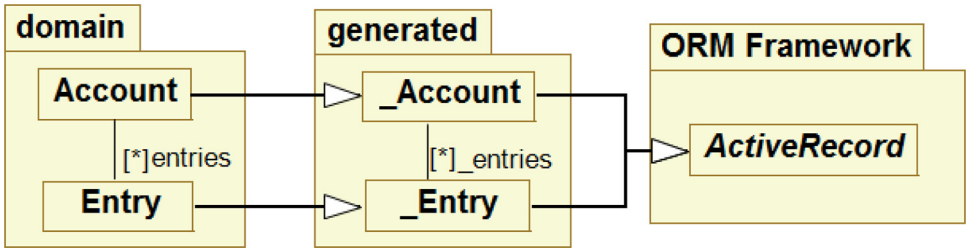


**Fig. 2.** Inheritance to generated classes.

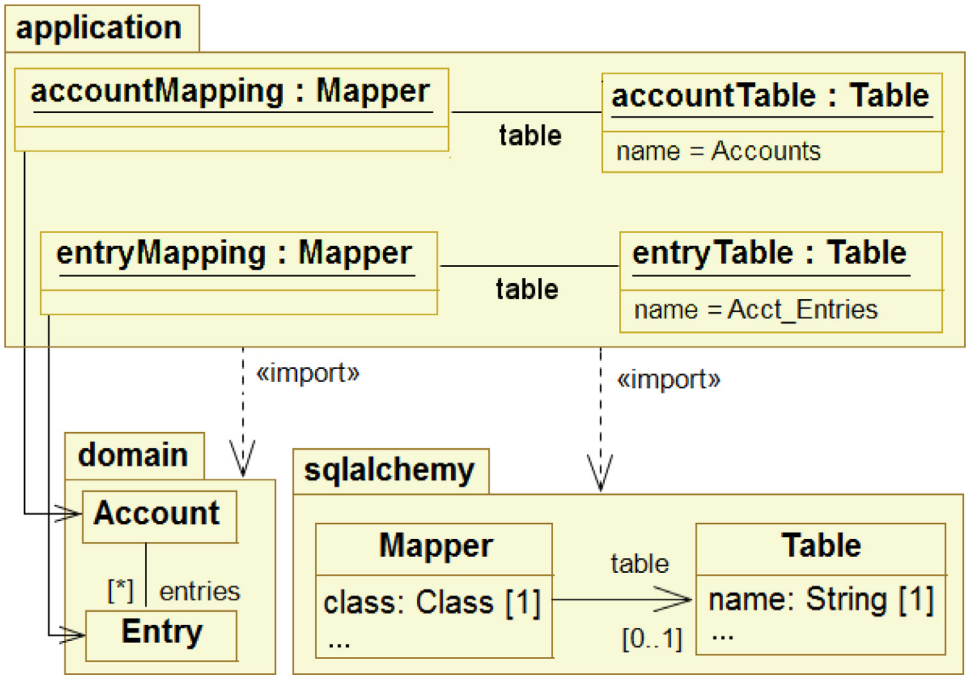


**Fig. 3.** Instantiated mapping example for *SQLAlchemy loose coupling* approach.

scenario. The *Cayenne* framework is an example of this approach, in which the *ActiveRecord* class is called *CayenneDataObject*. The generated intermediary abstract classes (with their name preceded by an underscore), contains the implementation of the mapping between the domain and the database accessing the persistence layer, including the properties and its *accessors* methods. The domain classes may then override these properties and implement the domain logic.

Both approaches to the *Active Record* pattern raised concerns about the high coupling to the persistence framework, because the inheritance relationship imposes a *strong coupling* to the general classes [39]. The *Data Mapper* pattern proposes a solution where mapper objects point out to the domain classes, encapsulating the database access, keeping the domain classes *loosely coupled* to the ORMS.

The *SQLAlchemy* persistence framework for the *Python* platform presents a hybrid example of the previous *Active Record* examples, and a *loosely coupled* solution based on the *Data Mapper/Mediator* pattern. The developer can choose among inheritance from base class, configuring a *strong/tight coupling,* or the *loose coupled,* dynamic instantiation of mapping objects that link domain classes to table definition objects.

Fig. 3 shows a simplified example of the transparent mapping, achieved in *SQLAlchemy*. The framework provides classes to specify the table (with attributes such as *name, column, column types*…) and mappings (*Mapper* class) between a *Table* and any class. In the application package, some class will instantiate the tables with its *metadata*, and the mappers binding each *Table* to the corresponding domain class. For example, the instance *accountMapping* is a *Mapper* that refers to the instance *accountTable* as its *Table* and to the domain class *Account* as its mapped class. The *accountTable* contains information about the database table, such as the name of the table being *Accounts*, along with all *metadata* necessary to create the table and its constraints (not shown below, for the sake of simplicity). The *Entry* class is mapped in a similar way, by the *entryMapping* instance referring to *entryTable,* with the table named *Acct_Entries*. The framework can then use the *Mapper* instances to access the database and factory instances of the domain classes, based on application requests. This kind of *loose coupling* by instantiating mapping classes is known as *Instantiated mapping.*

A variation of the *Instantiated mapping* approach has become part of the *JPA* standard, as illustrated by Fig. 4: instead of instantiating the mapping on some application class, as with *SQLAlchemy*, the mapping is done on independent files and/or code annotations, achieving a better separation of concerns. As with *SQLAlchemy*, the domain knows nothing about its persistence, and the persistence can be applied to any domain package, even without its original source code. *Doctrine* is also a *Data Mapper* that uses external configuration, but on code comments, to achieve *loose coupling*.

An additional advantage is that the *JPA* acts as an abstract factory [18] to one among various *JPA* frameworks, making it possible to reduce or eliminate the coupling between the application and the implementing framework. In *Java*, domain classes with no cou-

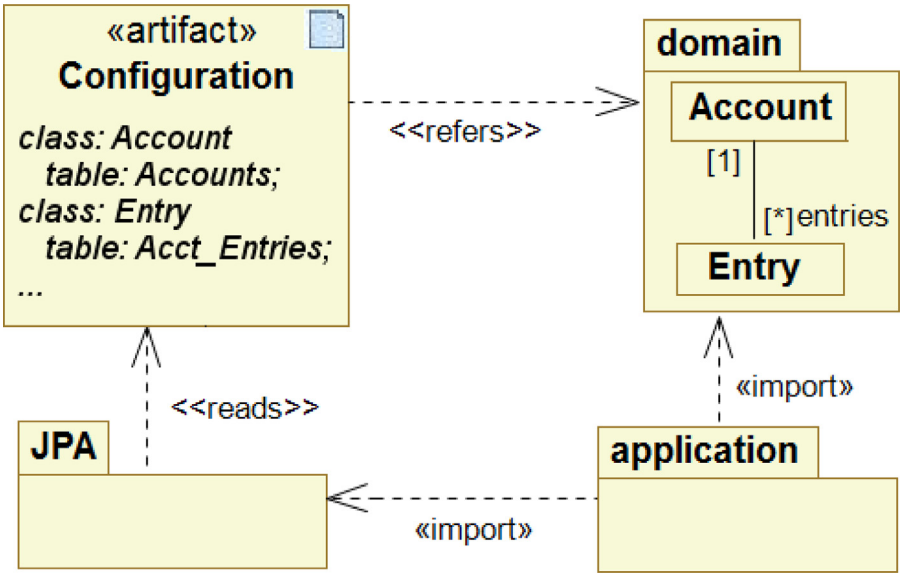


**Fig. 4.** *JPA loose coupling* overview.

pling to the persistence framework are commonly referred as POJO (Plain Old Java Object) classes [42].

The *EF* uses an *Active Record* coupled approach, like the one shown in Fig. 1, combined with a configuration as its default framework [32]. The domain classes specialize the *EntityObject* abstract class [43]. There is also an alternative of using the POCO (Plain Old CLR Objects) classes, which avoids the inheritance coupling on the domain classes [44]. *ODB* uses the *Data Mapper* approach, but the coupling is very tight, because the *ODB* base library must be declared as friend of each domain class [31].

*Bookshelf* claims to be a *Data Mapper* [37], although we could not find the separation between domain objects and data mappers. CRUD operations are issued directly at the domain objects that extends a *Model* prototype of *Bookshelf*, and these are characteristic of the *Active Record* pattern. Because of that, *Bookshelf* is not loosely coupled.

Finally, *MyBatis* also employs *external configurations* for ORM. It achieves a *loose coupling* between the domain and the framework, encapsulating framework access at the application level.

#### 4.1.1. Discussion

The key reason to choose the *Data Mapper* pattern is decoupling the domain classes and the ORMS. With *Active Record*, domain classes inherit from classes of the framework, or classes generated by the ORMS that inherit from ORMS libraries, whereas in *Data Mapper*, mapper classes unidirectionally reference domain classes, and only these mapper classes are coupled to, and/or generated by, the ORMS (raises Q2).

A framework that requires subclass *coupling* will introduce "alien" elements from the framework into the domain classes. If the platform does not support multiple inheritance, it will be impossible to have inheritance between persistent and transient domain classes, given that each domain class already specializes its persistent counterpart and cannot inherit from another class. Notice that other kinds of strong coupling may be introduced, for example, when domain classes declare variables with types provided by the ORMS (raises Q3).

On the other hand, an ORMS that allows a loosely coupled domain model still has restrictions, mapping requirements, and limitations. The naive assumption that *any* model can be persisted may lead to a domain model that cannot be implemented, or is unpractical, with the chosen framework or technology. If legacy databases are present, and they often are, chances are great that the design freedom on the domain model will be limited.

It is also worth noticing that the *Data Mapper* pattern included finder interfaces at the domain model, implemented at the mapper layer [11]. The only ORMS that effectively implements finders in that way is *MyBatis*. Most ORMS exposes a generic API that implements the data mapper and a bunch of other patterns, offering a single finder by PK, and a finder by general criteria. It is up to the developers, and usually a good practice, to implement object oriented specific finders, encapsulating all query related code for future maintenance.

If the ORMS employs external configuration, in the form of files, or annotations, it may avoid coupling, but introduce configuration management issues. For example, a validation step may be required to assure that the mapping configuration is still valid for the domain model in question to prevent run-time errors. In that case, whether or not ORMSs offer some mechanism to help validate and manage mapping configurations would be another issue to consider.

### 4.2. Metadata and schema

The core of any ORMS is the mapping of database data to/from objects in memory. The *metadata and schema* criterion examines the problem of how distinct ORMSs represent these class-table mappings, what artifacts should be maintained or generated, and how they affect the original domain model.

There are two basic alternatives, the first is to let the developer write the mapping for each CRUD operation of each domain class; and the second is based on data specifying the mappings between objects and tables, known as *metadata*, and let the ORMS assemble the SQL from this. The latter follows the *Metadata Mapper* pattern [11].

Fig. 5 illustrates how the *MyBatis* works without following the *Metadata Mapper* pattern. The ORMS reads the configuration containing SQL statements, prepared by the developer, for each mapping situation of each class in the domain. For example, *AccountMapper.xml* defines a query that is responsible for the construction of each *Account* instance; and *EntryMapper.xml* declares a query that returns a set of *Entry* instances related to one *Account*. The developer implements the *domain* classes, and *mapper* inter-

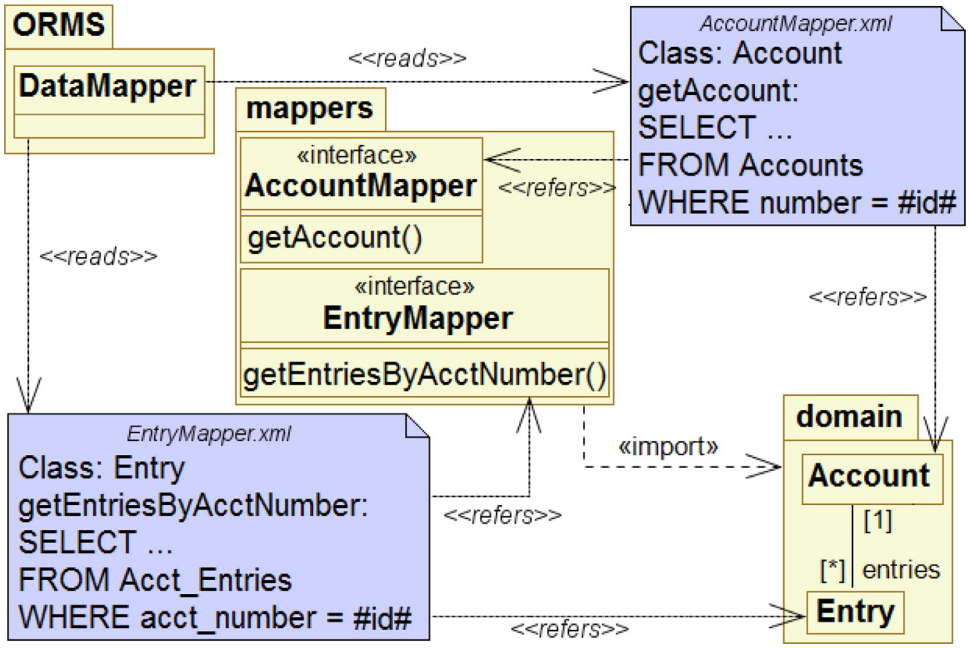


**Fig. 5.** Non *Metadata Mapper*, such as in *MyBatis*.

faces. The actual mapper implementations are generated by *MyBatis* at run-time, using the SQL in the configuration files.

The *Metadata Mapper* is represented by the *Mapper* class in Fig. 3. Instead of storing developer written SQL statements, it stores information about the tables, columns and mapping options of the developer [11]. The mapping options depends on the attribute and column types, including cardinality, length and precision, but may require some special configuration for Large Binary Objects (LOBs) and dates (raises Q13). This *metadata* is then used to dynamically assemble a SQL statement for each operation.

A *Metadata Mapper* framework may have *metadata* informed by the developer, using configuration artifacts (instantiating mappings, external files, annotations, or preprocessor directives), and/or by introspection. Another technique to obtain *metadata* is to *extract* it from the database itself and combine with data informed by the developer. In such case, the database schema itself is a configuration artifact, complemented by the developer according the framework rules.

*RAR* uses the *metadata extraction* approach to perform all column mappings. The developers do not need to declare properties representing the database columns. All they need to do is to declare the class mapped to the table and the framework will, at run-time, query the database to obtain column *metadata* and dynamically provide properties to the classes. One drawback of this approach is that without a database connection, the developer may not know what properties a persistent class have. *Ruby* developers usually maintain migrations files for *metadata* reference.

#### 4.2.1. Mapping classes to many tables

Usually, each class is mapped to one table, but sometimes it is useful to map a class to many tables. A naive approach to mapping classes to multiple tables on ORMSs is to create SQL views, and then try to create an algorithm that automatically maps changes from any view to its depending tables. However, depending on how the view is defined, propagating changes to the database may introduce side effects in other unrelated instances. Ultimately, the *ideal* mapping implementation would have to translate the operations affecting the tables, and these operations should be side-effect free: a change on one instance cannot affect any other instance. This kind of automatic solution, however, works only for certain simplified views [45,46].

Another approach using views is to write each mapping implementation in the database as stored procedures, and/or triggers. This approach gives greater flexibility, are easier to adopt with active records, but hides the mapping inside the database, affecting the database portability.

*Metadata Mapper* ORMSs usually allow the mapping of one class to multiple tables. *JPA, EF, SQLAlchemy,* and *RAR* have mechanisms to map a class to more than one table, offering some mechanism to resolve the persistence of the instances. ORMSs that require manual mapping, such as *MyBatis*, provide the most flexibility by requiring the specification of the CRUD operations.

*JPA, EF,* and *SQLAlchemy* allow the mapping of one class to many tables by joining their PKs. The framework retrieves the data by performing inner joins, and persist data by automatically issuing CRUD operations for each table in the mapping. *SQLAlchemy* also allows the specification of arbitrary join conditions by requiring the implementation of CRUD operations. Although *JPA* did not specify how to map a class to arbitrarily joined tables, frameworks implementing *JPA* are free to offer their own extensions overcoming this limitation. *Hibernate* extends *JPA* by allowing the definition of the SQL statements that perform the joins and database changes [47].

*RAR* allows many tables by using nested attributes [48]. Differently from the other data mappers, the secondary tables must first be defined as classes, following the *Active Record* pattern, with the necessary associative mappings. Therefore, *RAR* does not hide secondary tables from the domain.

#### 4.2.2. Models, metadata mapping, and code generation

ORMSs have two non-exclusive techniques to implement *metadata mappers*:

- Generating code from the *metadata*: a tool generates the code, usually the mapper classes, from the *metadata*. This code is hidden, or write protected, if possible.
- Using reflection: An API, usually a general mapper class, reads the *metadata*, stores them in memory, and uses reflection to access, instantiate, and populate domain objects.

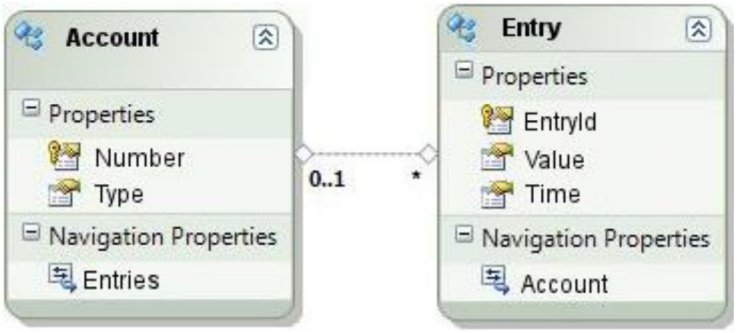


**Fig. 6.** Entity Data Model for *EF*.

C++ does not have reflection, thus *ODB* uses code generation. The *ODB* compiler reads the domain headers that contains "*#pragma db*" directives, parses the code to identify properties and relationships, and generates the code for the classes. With *ODB* the developer cannot implement operations as methods in the generated domain classes, but C++ allows the implementation of operations of one class in the header, or in separated *cpp* files.

*Java* does have reflection, and *JPA* makes use of reflection, but in a very similar way. The domain classes have mapping *metadata* specified as class annotations or in an XML file, and the persistence layer analyzes that *metadata* when it starts up, thus building its mappings in memory instead of generating code. The majority of studied ORMSs uses reflection: *JPA, Python, RAR, Bookshelf.js, MyBatis, and Doctrine*. Sometimes, frameworks that use reflection will also generate classes, such as proxy classes, but at run-time, keeping them invisible to the developer.

The *EF* uses code generation, but instead of analyzing headers, it uses a configuration file, generated from a graphical modeling tool called Entity Data Model (EDM) [32,49]. The mapping starts with a model (with an E-R logical inspired notation) that can be created from scratch or by reverse engineering a database. This conceptual model is extended with visual elements to identify ORM patterns supported by the tool, such as navigation properties for relationships. Finally, this model is used to generate source code and configuration that implements, by a template mechanism, the domain objects and its mappings.

Fig. 6 presents an example of EDM. The *Entry* entity has a many to one relationship to *Account*, specified by the navigation property *Account*. The *Account* has a reverse navigation property named *Entries* that contains a collection of *entries*.

The disadvantage of using an E-R like model (such as EDM) as the source for generating the domain model is the absence of visual behavior specification for the entity classes. Even when dealing with code and mappings, currently it is only possible to define methods that call stored procedures outside the model. The classes generated by EDM are partial, a *C/VB#* keyword that allows classes to be defined at separated files. The developer can add the operations of the class in a separate, non generated file, but these are not maintained by EDM. It is also possible to use *EF* without a model, but a model surely makes the mappings much more readable.

*Cayenne* is another example of *model based* tool that generates the source code from a visual model. Instead of one model, *Cayenne* requires the specification of two models. The first is a data model and the last is a class model bound to this data model. Nevertheless, the domain classes of *Cayenne* are specializations of the generated classes, as shown in Fig. 2, and are not read only. The developer is free to add operations to the domain classes.

Fig. 7 presents an example of *Cayenne* model for a similar *Account x Entry* model. The "C" icon represents the classes (*ObjEntities*) and the boxed icon represents tables *(DbEntities)*. The modeler allows the specification of how each *ObjEntity* is mapped to one or more *DbEntity*. The model itself uses an "explorer tree" visual representation, far away from the E-R visual language.

#### 4.2.3. Discussion

Without *Metadata mapping*, the ORM expects the specification of query statements for each CRUD operation on each domain object. These statements comprise the mapping between the domain and the database, being sensible to changes in both ends, and often dependent to the chosen database platform. It allows more flexibility to map objects to multiple tables, but also requires writing the queries even when the mapping is trivial. If the mapping is written in SQL, it may be coupled to the database provider dialect.

The *Metadata mapping* approach requires the specification of equivalence between domain classes and database tables, what sometimes may be difficult to achieve in legacy systems. The more resourceful is the *Metadata Mapper*, more freedom the domain model will have from the database schema and/or *vice versa*. Transferring the responsibility of assembling queries to the ORMS may impact performance, and this may influence the design, requiring a careful parametrization of the mapping. The designer must take measures to document the correspondence between classes and tables, specially when they are not mapped one-to-one (raises Q4).

A visual modeling approach allows for a better understanding of the mappings between classes, specially its relationships. EDM option for E-R models naturally limits behavior specification, better expressed with class models. *Operations,* on the application side, are not represented in the models, although they can be added directly at the source files (raises Q1).

The separation of class and table visual concepts on *Cayenne,* potentially duplicates the number of elements that the developer have to deal with; elements that represent the same thing, in a conceptual viewpoint. Its modeling tool is much more a visual editor, than a notation itself, leaving most of mapping information hidden, behind wizards and menus, instead of graphic displaying its details. Moreover, it has the same limitation of the EDM regarding the lack of *operations* specification. A *model based* framework, with round-trip engineering, supporting changes in the model and/or the generated code, is not being offered by any of the tools available for the researched ORMSs.

### 4.3. Identity

The PK is the minimal subset of columns that uniquely identifies a row in a database table. The PK may be composed by meaningful, or meaningless, information to the developer viewpoint, and it is usually immutable. PKs can be referenced in other tables by Foreign Keys (FKs).The *identity* problem refers to the mapping of the PK concept to the domain object.

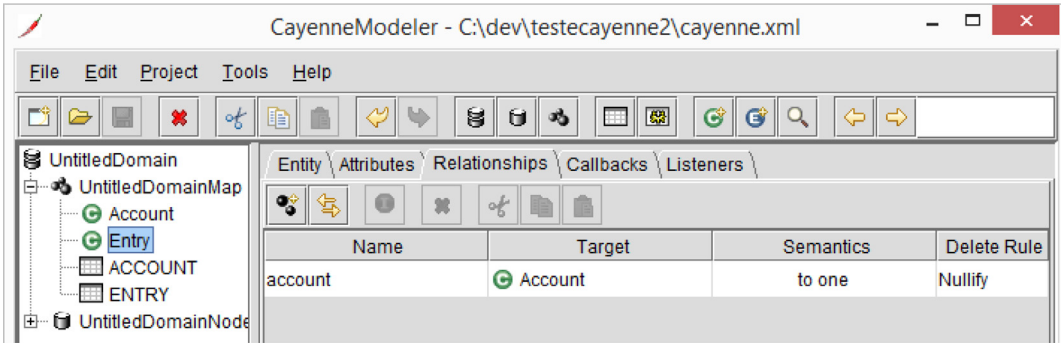


**Fig. 7.** *Cayenne* models domain and database elements separated.

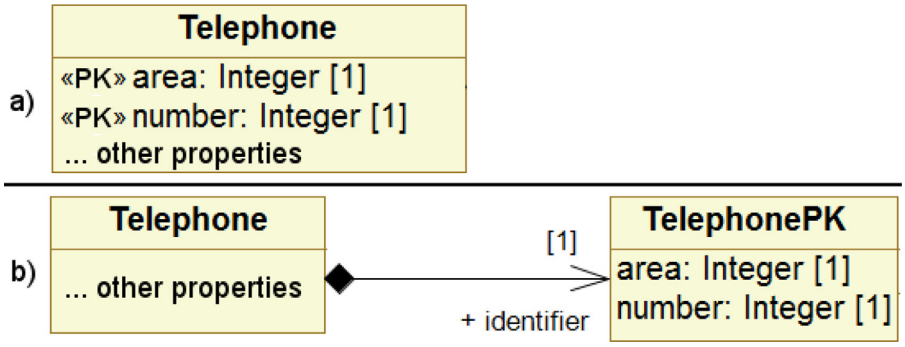


**Fig. 8.** Primary key fields (a) and primary key class (b).

Objects usually do not need a declarative *identity,* because they are internally identified by their memory position. From an OO viewpoint, if an object is serialized to an external system and later reconstructed to the same state, it is another instance and therefore a distinct object. Non standard equality, and guarantee of uniqueness, are behaviors that must be implemented on the class, such as the *equals* operation on the *Java* platform [50].

The *Identity Field* pattern [11] lists the following characteristics that we consider relevant in this survey:

- *Identity Complexity*: can be of *single* or *compound complexity: single* identities are formed by a single column; *compound* identities by a set of columns. *Compound* keys are often used to implement weak entities.
- *Identity Uniqueness scope*: can be of *table* or *database uniqueness*. It concerns if the database is designed to have PKs that are unique in the context of the *table,* or the entire *database. Table uniqueness* is the most common situation on legacy databases.
- *Identity Assignment*: can be of *auto-generation, counter or user assigned.* A database can offer different resources to assign the PK: With *auto-generation* the database assigns a key for each inserted row; With *counter assignment*, the developer obtains a new key from a named unique *counter* (also called *sequence generator)* that may be shared by several tables; and with *user assignment,* the application is responsible of assigning the key, asking the user for a meaningful key or generating the key by a client side solution.

It is desirable that ORMSs support as many as possible combinations of the above characteristics. Some ORMSs only support single key mappings, such as the *RAR* [35]. Some characteristics are supported by all frameworks, such as *single complexity* and *table uniqueness.* All ORMSs in this survey allows *user assigned* ids.

*Composite keys* may be represented by a separated *PK class,* referenced by a single composite variable in the persistent class, or a list of *fields* identified within the class as the PK. In Fig. 8, diagram (*a*) shows the composite PK for *Telephone* as two properties, *area* and *number*, directly containing the columns values; and diagram (*b*) depicts the PK as an *identifier* relationship, between *Telephone* and *TelephonePK* classes. Some frameworks, such as the ones that follow the *JPA* specification, allow both representations, although the definition of *PK classes* is required, even if the identified fields approach is used (raises Q12) [29].

Key assignment is a tricky problem to the ORM. A domain object is first created in memory and then it *can* be persisted in the database. If the key is *user assigned*, the ORM must ensure that a key is provided before that object is persisted. On the other hand, if the key is generated by the database, it must not be assigned by the user. If it is an *auto-generation* field, the ORM must implement some way of capturing the generated value, to store in the object for later updates. When a *counter* is employed, the *counter* specification must be informed in the mapping, or obtained from some default naming convention.

The *user assigned identity* may be automatically generated in the ORM level. The most common techniques are based upon Globally Unique Identifier (GUID) generation, key tables and table scans. A framework may allow the user to create customized generations, combining different techniques.

*Complexity* and *assignment* are affected by the chosen *uniqueness* scope. *Composite* and *auto-generated* identities are usually related to *table* scope, while GUID and *single* identities are the usual combination for *database* scope. The ORM may treat all scoping as *table*, offer some facility to distinguish *database* or *table* scoping, or just require *database* scoping. By treating all scoping as *table*, the developer can still implement some sort of *database* scoping, if the ORM supports *user assignment* to generate GUIDs for instance. Conversely, requiring *database* scope turns out to be an impediment for a database schema with *table* scoping and access by legacy systems.

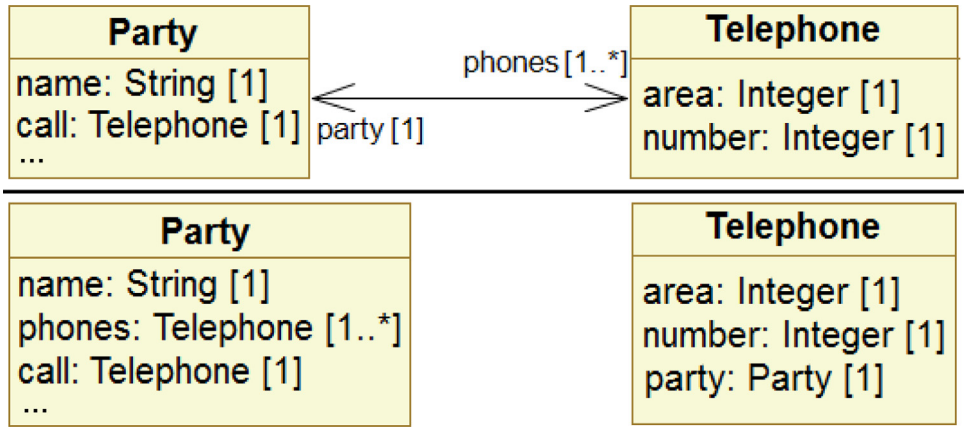


**Fig. 9.** Association between classes (upper model) and its implementation (lower model).

*4.3.1. Discussion*

PK mapping is often forgotten in the OO project stage, spanning unpredicted PK fields and classes in the implementation stage. Another common problem is the absence of the PK attributes in the domain classes, since they are required only to model the database (raises Q5). The *assignment*, and *uniqueness* of scope, may impact in the inheritance support of the ORMS (raises Q6).

There is a practice among some OOPL developers of using only single column, generated, integer based PKs. However, from the RDB viewpoint, this practice is an anti-pattern, known as *One Size Fits All* [23]. Therefore, ORMS support for *compound* keys is not exclusive for legacy databases.

If a *Metadata Mapper* is employed, it will require information about how the PK will be represented. If not, the ORM will require the presence of "find-by-primary-key" factory methods for most domain classes.

### 4.4. Foreign key

One of the most important features of RDBs is the referential integrity, a mechanism that guarantees coherent relationships between rows in different tables [51,52]. One row references the other by having a FK, a subset of its columns that references the PK columns (or sometimes an alternate key) of the other table. The database must either ensure that the FK is pointing to some row that exists in the referenced table, or that it has NULL values in all its columns. This criterion examines the problem of how classes represent FKs in the domain model, and what mapping information is required.

The *Foreign Key Mapping* pattern describes the common solutions for representing database relationships in classes (and *vice versa*) for the *one-to* and *to-one* cases [11,12]. In OO systems, classes may reference other classes by variables, what denotes a (usually transient) dependency relationship. An association (or structural) relationship between two classes A and B typically happens when there is an instance variable on A with type being B, or a collection of B elements [22]. This relationship is often persistent, and may be unidirectional or bidirectional, the later typically requiring another instance variable on B referencing A.

Fig. 9 shows two diagrams relating *Party,* a pattern to represent a person or organization, and *Telephone* [4]. The upper diagram is a UML representation of a bidirectional relationship between *Party* and *Telephone*. The lower diagram, however, shows the implementation of this relationship, using the instance variables *phones* and *party*. It is not clear, without looking at the upper diagram, if *phones* and *party* forms a bidirectional relationship or two independent unidirectional relationships. If a second variable references the same destination class (*call*), there is no structural information that indicates, on the classes, which property is in the opposite side of the relationship. There is nothing like declarative FK constraints available in OOPLs.

On RDBs the FK is placed in one table, depending on the cardinality of the relationship. The row with the FK can reference at most one row in the other table by this FK. Rows in the table referenced by the FK, often named master table, can be referenced by zero or more rows within the same constraint rule. Nevertheless, database relationships can be considered bidirectional, because the SQL query language allows to access both the master row knowing the detail FK, or the detail rows by knowing its master PK.

The *Foreign Key Mapping* pattern simplest version is to have a field that references, in the class/table that owns the FK, an instance of the master class/table. This mapping is known as *many-to-one*. Sometimes the inverse mapping, named *one-to-many,* is a more significant design, such as the party referencing its phones, in such case the field is a collection, owned by the master class. *One-to-one* relationships differ very little from *many-to-one,* regarding the implementation in unidirectional situations.

Nevertheless, if both classes references each other, the *foreign key mapping* is named bidirectional, and it must deal with the problem of keeping both sides of the relationship aligned. If one *Telephone* is added to the *phones* of *Party*, the *Telephone* itself must have its *party* reference updated, and if the *Telephone* is assigned to a different *Party*, both parties must be updated: the old *Party* collection of *phones* must have this *Telephone* removed and added to the *phones* collection of the new *Party* (raises Q17).

The simplest case of mapping unidirectional *many-to-one* is supported by all studied ORMSs. The instance variable *identity* is mapped to a FK, automatically, if the name matches the database structures. Additional mapping information is needed when the variable name did not match with the column name, or when the identifier is *compound*.

The bidirectional case must support some mechanism to specify what is the opposite instance variable, as shown in the Fig. 9 *party-phones* relationship. To represent the *to-many* side, the instance variable requires some collection instance, with variable size support, such as classes of the Java Collection framework. Mechanisms such as Templates, or Generics, allow the framework to type collections (raises Q14).

Some programming languages do not support the typing of collections, leading to some mechanism to specify the target of the relationship. One solution, employed by the *RAR*, is to match relationships by the variable name. If this match is not possible, the relationship can still be defined, by overriding the standard *Active Record* implementation. Most ORMSs have some customization

approach using reflection, instantiation (*SQL Alchemy*), or *external configuration* (*JPA, ODB, EF*).

The *one-to-many* unidirectional relationship is one exception scenario. By using the *FK* pattern, the key is stored in the many side, which should supposedly not know about the relationship. This can be handled, at some ORMSs, by implementing *one-to-many* with the *Association Table Pattern*, avoiding a FK owned by an unnavigable side.

For the *one-to-many* direction of the relationship, some frameworks work with read only collections and add/remove methods for the elements, while others allow the direct manipulation of collections, later reflecting these changes as updates in the referring object. *Cayenne,* for instance, requires a method to add elements to the collection, while the *JPA* allows direct manipulation of collections. However, for bidirectional relationships, changes on one side may not be automatically reflected in the opposite side, hence add/remove methods are a good practice to encapsulate such details, when this is the case.

*Compound* PKs often include a FK, but an ORMS that supports *compound* keys may not support a *navigation from the weak entity*. In order to support this navigation, the ORMS must accept properties with a persistent type, besides the scalar types such as *integer* and *string*.

Polymorphic associations are a special *one-to-many* relationship, with the *one* side assuming one of various unrelated classes, according to the value of a column in the *many* side [35]. This is only possible with dynamic typed objects, and cannot be expressed with a FK. Polymorphic associations with unrelated classes are considered an *anti-pattern*, and should be avoided when possible [23].

#### 4.4.1. Fetch strategy

Objects are connected to each other through associations, forming a graph that is often huge. When one object is retrieved from the database, the *fetch strategy* indicates whether each associated object will be loaded along from the database, controlling the depth of the retrieved object graph [40]. For the example in Fig. 9, the *fetch strategy* specified for the *phones* of a *Party*, would specify whether these phones should be (*eager*) loaded from the database along with that instance. Conversely, when a *Phone* instance is retrieved from the database, the loading of *Party*, into the *party* instance variable, is also subject to the *fetch strategy* specified for this instance variable.

The *Lazy Load* pattern offers a conservative solution to the fetch strategy, by deferring the loading to the first moment the information is requested by the system [11]. For collections, the most common solution is to provide a transparent wrapper around the collection that checks if the collection was initialized, loading it only when needed. ORMSs that allow associations to be configured between various fetch strategies are more flexible for performance tuning.

In the *to-zero/one* scenario, the solution may be a virtual *proxy*, an instance of the class that actually is a pointer to the real object, retrieved on demand from the database [18]. However, using virtual proxies may end up affecting polymorphism. For instance, imagine the following scenario: the *Party* class has a specialization called *Person*. If the system retrieves a *Telephone* with lazy loading strategy, a *Party* virtual *proxy* will be instantiated despite the possibility of the *party* being a *Person.* If the system access the *party* variable, the related *Person* will be instantiated and loaded, but the already instantiated *Party proxy* will be wrapping that *Person* instance. The only way to know beforehand the correct type of a proxy is when the type can be discovered by looking at the FK, usually because it is a composite of type-discriminator and ID. Some ORMSs even allow the selective load of columns of the objects.

With *MyBatis*, the lazy loading is controlled by the *nesting type* used in the configuration. In the previous example, the *Party* class might be mapped to an outer join with *Telephone* and a nested *resultset*; or, alternatively, mapped by one query to *Party* and a nested "on demand" query for *Telephone. Bookshelf* implements only lazy fetch, and therefore it is not possible to choose the strategy. All other ORMSs in this survey, present the possibility of *fetch strategy configuration*, but only *JPA* and *SQLAlchemy* allow the configuration of *proxies* for relationships.

#### 4.4.2. Discussion

Relationships are best represented in visual models rather than code or SQL, in which the reader must interpret statements to discover the nature of one relationship (raises Q18). For ORMSs, some mapping is often needed to specify elements such as relationship direction, type, or collections that are connected by FKs.

Performance is a concern in relationship mapping. The *fetch strategy* plays an important role in ORM (raises Q15). Lazy fetch in *to-zero/one* relationships has different consequences on each ORMS, leading to the use of proxies and impacting on the way that the domain objects behave (raises Q16). When implementing associations, the default fetch strategy of each association property will determine how many queries, how many rows, and the overall complexity of these queries when retrieving one or more objects, or accessing its lazy properties.

### 4.5. Association table

RDBs do not deal transparently with *many-to-many* relationships. These relationships must be implemented by a third *association table*, containing mandatory *many-to-one* relationships to both related tables. This criterion examines the problem of modeling, mapping, and managing these relationships with ORMS.

The *Association Table Mapping* pattern describes the common solutions to represent OO *many-to-many* relationships in databases [11,12]. The basic idea behind the *Association Table Mapping* is to use a link table to store the association. This table has only the FK IDs, for the two tables that are linked together, and it has one row for each pair of associated objects. The link table has no corresponding in-memory object. As a result, it has no ID and its PK, if it exists, is the *compound* of the two PKs of the tables that are associated.

Fig. 10 shows an example of *many-to-many* table mapping, between classes *Employee* and *Post* of the *Post* pattern [4]. The *Employee* class has the collection field *posts* referencing *Post* objects. *Employee* and *Post* are stored in the tables with the same names, and the *employees-posts* collection is stored in the *Employee_Post* association table. If a *Post* is added into the collection, a row must be inserted in the *Employee_Post* table, with corresponding *empId* and *postId* values; if it is removed from the collection, this row is deleted in the same way.

Tables such as *Employee_Post,* present in the database model, may not be of interest from an OO design standpoint. In OO language models, *many-to-many* relationships can be transparently mapped into object references, as long as the underlying relationship table does not contain information by itself. For example, an *employees-posts* association may be transparently mapped, as long as it does not have other information, such as *experience* in years, or *type of degree*. When the relationship has attributes, it usually ends up being a *Class,* represented in UML by an *Association Class* [15].

The *Employee_Post* table is not represented in the class model, but ORMSs usually have to store *metadata* about this table, in order to implement the relationship. Some tools may assume the table name from the class names, but usually the developer supplies the name of the *association table*. Nevertheless, association tables

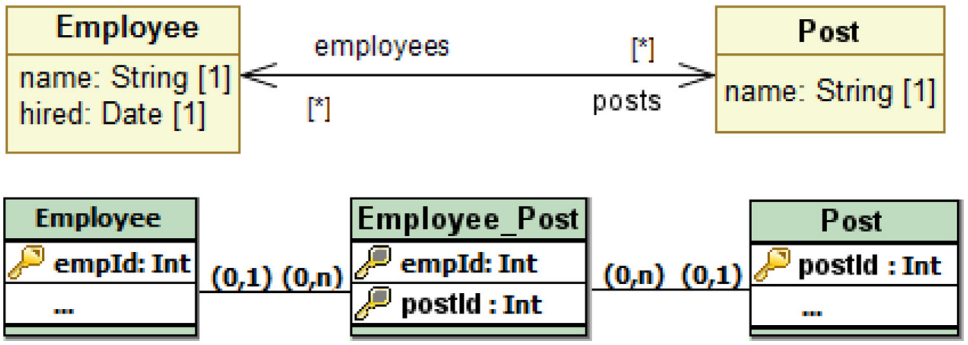


**Fig. 10.** *Association table* example.

must be well documented, otherwise changes on the classes may be difficult to implement (raises Q19).

For bidirectional relationships, the ORM must deal with two instance variables (*employees* and *posts*) that represent dependent collections. If a *Post* is added or removed from the *posts* collection of an *employee*, this *employee* should be added/removed from the *employees* collection of this *Post* instance.

Most persistence frameworks are able to deal with transparent association tables, but some of them may have restrictions (raises Q20). For instance, the *EF* assumes that the *association table* has a PK consisting of the two FKs and is unable to recognize any other column in this table. An *association table* with a unique identifier is therefore unsupported. The *Cayenne* Framework allows some flexibility of the transparent *association table* for read-only relationships.

Transparent *association tables* are not directly supported by *MyBatis*. Nevertheless, the mapping language allows the definition of collections over nested results, or nested queries that can emulate the *association table*, without the existence of an association class to explicitly map the *association table*.

*JPA* supports transparent mapping of *many-to-many*, and allows to represent the relationship as a *Map (or Dictionary)* containing the associative attribute(s) (raises Q21). For example, the *posts* of an *employee* can be represented as a *Map* with *post* as the key, and *experience* in years attribute as the mapped value. This value will be mapped to an attribute in the *Employee_Post* table. Although other ORMS, such SQLAlchemy, can also use *Maps*, we did not find any example of *Maps* with association attributes.

#### 4.5.1. Discussion

*Transparent association tables* are a great abstraction for modeling systems, and may simplify the implementation of domain classes, by removing association classes required only to emulate *many-to-many* relationships. However, support for *association tables* is often limited in ORMSs, while it is common to find imaginative variations in legacy databases that are unsupported by *transparent association tables*, regarding, for instance, the questionable practice of using surrogate keys in *association tables*.

### 4.6. Embedded values support

Not all classes that represent persistent information in the OO design model make sense as database tables. Some classes may represent persistent data only when related to another persistent instance. This criterion examines the problem of modeling and mapping classes whose persistence depends on its relationship with other persistent objects.

In cases where two classes relate in a *one-to-zero/one* basis, the dependent classes are the best candidates to be stored within the owner table, by the application of the *Embedded Value* or *Single-Table Aggregation* patterns [11,12]. Imagine the situation where the system deals with contacts for its employees and customers, within the *Accountability* domain (Fig. 11). A *contact* may be an *Email* or a *Phone* with SMS support. Both *Employee* and *Customer* can have one *Phone* and one *Email* contact, but the customer may not inform the contact information. To reuse and encapsulate the contact behavior, phone and email are separated classes, but from the database viewpoint these are simple attributes of the parent tables.

*Phone* and *email* can be seen as regular value objects (like *String* or *Date*) and are only persisted when related to an owner instance, such as *employee* or *customer*. Usually the relationship is unidirectional from the owner to the *embedded* class, which denotes an "*attribute of*" relationship [15]. This relationship may be a composition, although it is common to see the owner reassignment allowed by the persistence framework. Usually the ORMS provides some way to differentiate attributes mapped as embedded objects and mapped to database columns (raises Q11).

*Cayenne, Doctrine,* and *JPA* have support for *embeddable* classes. In *EF* the *Embedded Value* is named Complex Type; in *SQL Alchemy* it is named Composite Column; in *ODB* it is called Composite Object; and in *RAR* it is named Aggregated Value Object. *MyBatis* does not need to distinguish embeddable classes, because for each class the developer have to provide the SQL for persistence.

Some frameworks such as *JPA* and *EF* allow the definition of standard *mappings for embeddable* classes, such as preferred database types (raises Q8). Others, such as *RAR* and *SQL Alchemy*, only support *mappings at the container* class, requiring a specific mapping of each reference to an embeddable class. In all ORMS, the designer must be careful about the contract requirements for embeddable classes, such as necessary mappings and/or operations that are usually implemented, such as constructors, accessors, or equality checkers.

Another issue is if the embeddable class can contain references to persistent classes or other embeddable classes, and if these are persisted along the container class (raises Q10). This characteristic is difficult to assess, while some ORMSs explicit forbid non scalar properties, such as *Doctrine* and *EF*, or explicitly allow it, such as *JPA*, others offer ways for the developer to emulate this behavior, with some creative programming, such as *RAR* and *SQLAlchemy*. With *RAR*, for example, one can override the constructor of the composite.

Another pattern that falls under this *embeddable* criterion is the *Dependent mapping*. In fact, as a pattern, *Dependent mapping* is a generalization of *embedment* that deals with classes that are persistent due to the relationship to other persistent classes, regardless of the nature or cardinality of the association [11].

Fig. 12 revisits the *embeddable* example replacing the *one-to-zero/one* with *one-to-many* relationships between *Employee*/*Customer* and *Phone*/*Email* contacts. In order to store more

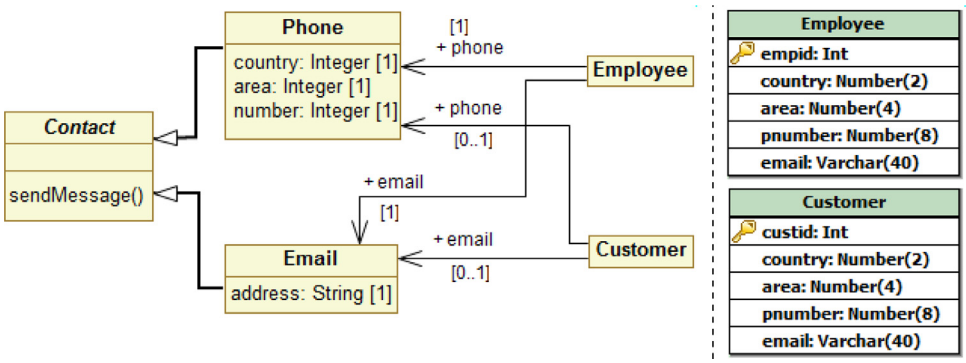


**Fig. 11.** UML model (left) and tables (right) of the *Embedded values* example.

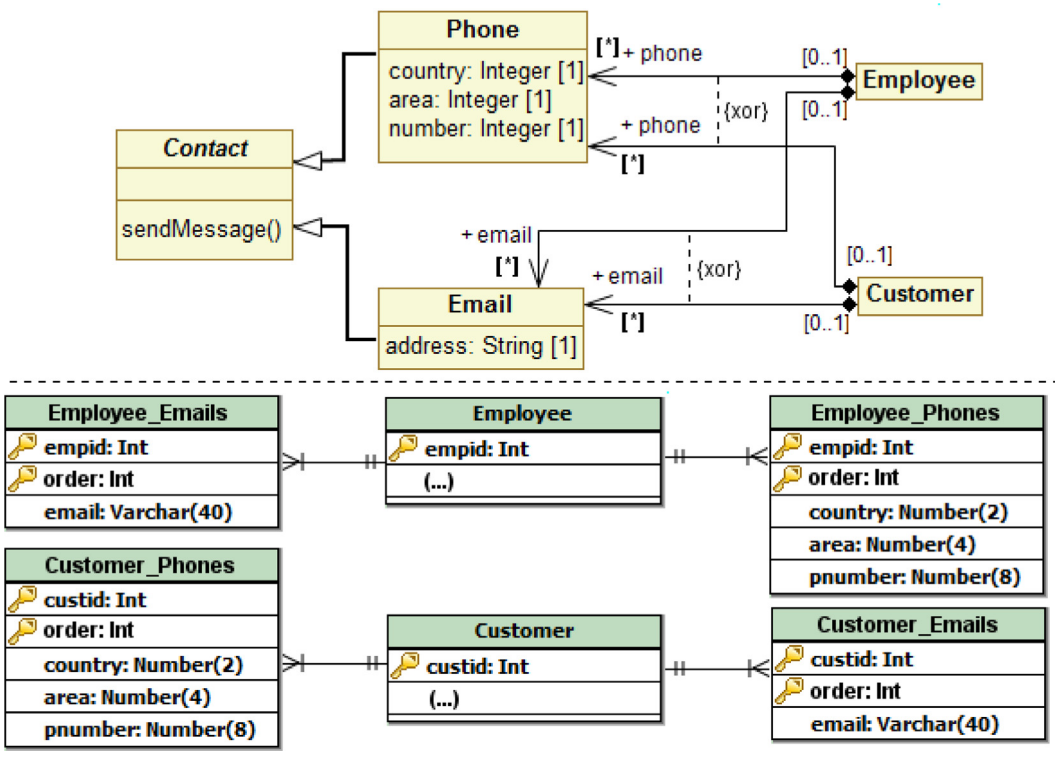


**Fig. 12.** Dependent mapping pattern example: class model (upper) and database model (lower).

than one associated element, a new table is required, as what happens with the *FK* pattern. The key difference here is that the table that stores the *phones* of *Customer* is not the same table that stores the *phones* of *Employees*. The table that stores a dependent mapping should do it exclusively for one owner, because a dependent must have exactly one owner [11]. Thus the same phone object will never relate to *Employee* and *Customer* at the same time.

In this example, the dependent tables are designed with composite keys with one column referencing the owner table, and the other registering the *order* of the element in the relationship. The composition relationship (dark diamonds in Fig. 12) simplifies the persistence changes, in a collection of dependents that can all be safely deleted and reinserted when required.

The *Dependent Mapping pattern* described by Fowler states that the owner class is responsible to the persistence of the owned objects. Considering ORM tools taking over this work, or with *metadata* mappings performing general persistence, this pattern can be used to describe *embedded collections,* such as collections of *objects* and basic types.

The support of more generic dependent mapping is not yet widely supported by ORMSs. *JPA* supports collections of embeddable objects (named *element collections*), mapping such collections into a separate table that refers back to one owner entity table (raises Q21). The element collection mapping allows the mapping of *one-to-many* unidirectional dependent mappings.

#### 4.6.1. Discussion

*Embedded values* are a valuable asset to bridge persistent and transient elements of the domain model. Reuse and encapsulation are great advantages of OO, and the forces that lead OO design to break a class in two or more pieces are often contrary to the database normalization forces that put that information together [3].

The possibility of defining default mappings for the *embedded* classes can facilitate the design by reducing repetitive mapping. Information, such as field types and lengths, will probably be the same for all *embedded* mappings that target the same class. However, *embedded collections* are not very popular among ORMSs.

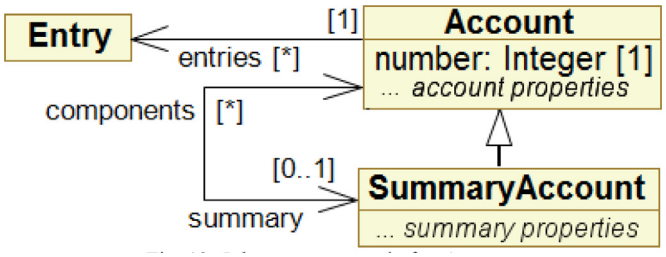


**Fig. 13.** *Inheritance* example for *Account*.

Some ORMS allows the use of *embedded* classes to declare as composite PK. This may require the class to implement certain contract, such as the *equals* and *hashcode* methods on JPA (raises Q9). Otherwise, when designing classes meant to be embedded, it is often necessary to observe if the class needs to be explicitly mapped as embeddable, or follow a specific contract (raises Q7).

### 4.7. Inheritance mapping

Inheritance is not supported in pure RDB models. It must somehow be emulated with optional fields, discriminators and/or table joins. This criterion examines the problem of mapping inheritance, the distinct strategies and their limitations and consequences to the domain model.

There are three preferred ways, identified as patterns, for mapping inheritance [11,12]:

- "*Single-table*" or "one inheritance tree one table", meaning that one table contains all possible attributes of the class tree.
- "*Class-table*", "Vertical inheritance", or "one class one table". Each class is mapped to one table containing only the attributes for that class.
- "*Concrete-table*", "Horizontal inheritance", or "one inheritance path one table". Each concrete class is mapped to one table, but each table contains the sum of all attributes of the class and its hierarchy of super classes.

The choice of patterns depends on implementation and platform specific issues. The balance of performance forces, such as *update and write access versus polymorphic read,* may be more determinant than maintenance and query complexity, in the decision about what pattern should be followed. Resources of the RDB must be taken in account, before deciding the best strategy, such as NULL compression, which saves space for the *Single-table* pattern [12] (raises Q22).

The *Account* example is expanded in Fig. 13 to illustrate a specialization of accounts, named *SummaryAccount,* which represents accounts that are a union of other accounts. *SummaryAccount* inherits properties from *Account*, such as *name*, and adds its own properties. The *components* relationship may connect a *SummaryAccount* to any *Account*, including another *SummaryAccount*. The *Entry* class has a polymorphic association with the *Account* class: it may refer to one *Account,* or to one *SummaryAccount* object.

A *Single-table* pattern solution for the account model is shown in Fig. 14. The *Accounts* table contains the sum of all attributes of the hierarchy and the sum of all associations. The *type* attribute (marked with ∗) was added to discriminate between an *Account* and a *SummaryAccount* row, although it could be replaced by using the *synthetic object identifier* pattern [12]. Mandatory associations are mapped as optional associations, but our example association was already optional. Nevertheless, the standard database integrity mechanisms cannot prevent an *Account* from being referenced by another *Account* which is not a *SummaryAccount.* It would be necessary to check the discriminator value, in order to determine that such constraint was violated (raises Q24).

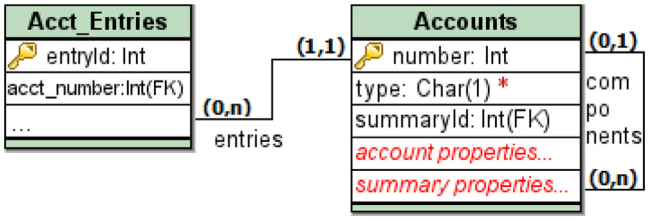


**Fig. 14.** *Single-table* approach for *Account* example.

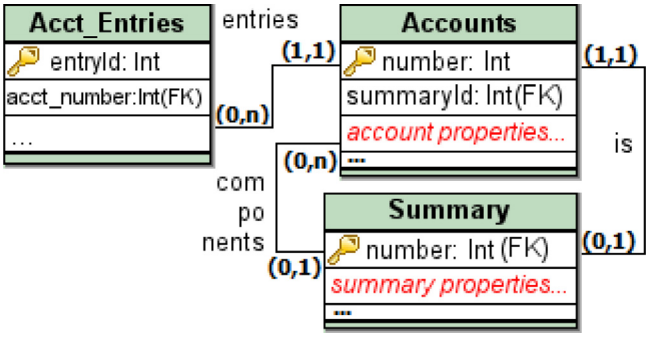


**Fig. 15.** *Class-table* approach for *Account* example.

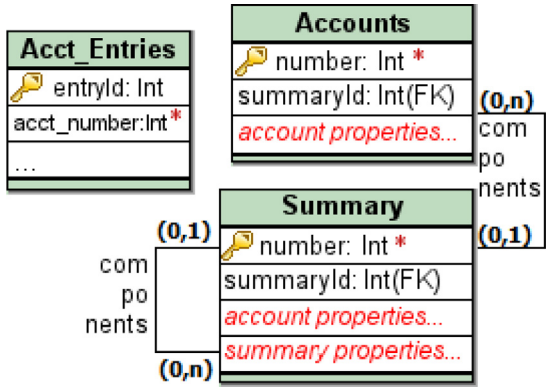


**Fig. 16.** *Concrete-table* approach for *Account* example.

A *Class-table* pattern mapping is shown in Fig. 15. *Accounts* and *Summary* represent *Account* and *SummaryAccount* classes, with a similar attribute/association distribution. To retrieve a *SummaryAccount*, the system must join tables *Summary* and *Accounts,* by its common PK. The association is now placed in the *Summary,* allowing the database to enforce the *components* association. However, the cost of joining *Accounts* and *Summary,* to obtain a single *SummaryAccount* instance, is usually higher than using a single table.

A *Concrete-table* pattern mapping example is shown in Fig. 16. Each concrete class of the hierarchy is mapped to one table, but data retrieval does not need to join tables. The consequence is that specialization tables will also contain columns for the inherited attributes.

Even without the FK constraint, in the specialized concrete tables, a unique *number* (marked with ∗) for the entire hierarchy is the best practice, because otherwise it would be impossible to represent a polymorphic association to a super class. In the example, *Acct_Entries* refers to an *acct_number* that can be a row at *Accounts* or *Summary* tables. This kind of relationship cannot be enforced by a FK, thus falling in the aforementioned polymorphic association anti-pattern (raises Q23).

Both *Account* and *SummaryAccount* can be components of a *SummaryAccount*, requiring two column references, to represent the *components* association of Fig. 13. The *summaryId* column, of

**Table 4**
How each ORMS names the inheritance patterns.

| ORMS/pattern | Single-table | Class-table | Concrete-table |
|---|---|---|---|
| RAR | Single table inheritance | *Unsupported* | *Unsupported* |
| Cayenne | Single table | Vertical | Horizontal *(Unsupported)* |
| EF | Table-per-Hierarchy | Table-per-Type | Table-per-Concrete Type |
| JPA 2 standard | Single table per class hierarchy | Joined subclass | Table per concrete class |
| MyBatis | *Does not apply* | *Does not apply* | *Does not apply* |
| SQLAlchemy | Single table inheritance | Joined table inheritance | Concrete table inheritance |
| Bookshelf.js | *Unsupported* | *Unsupported* | *Unsupported* |
| ODB | Table-per-hierarchy | Table-per-difference | Table-per-class |
| Doctrine | Single table inheritance | Class Table Inheritance | *Unsupported* |

*Accounts,* can reference the *Summary* PK, but a *Summary* PK can also be referenced by a *summaryId* column, of another *Summary*. In the *Concrete-table* strategy, it is not uncommon to have one association become two, or more, column references between tables.

Besides problems related to relationship mapping, the *Concrete-table* has a severe performance problem for polymorphic queries [12]. If one needs to query the entire hierarchy, for a specific condition, an expensive UNION operation is issued to all tables.

Surely, this example does not explore all possible combinations of problems adapting each inheritance pattern to the other relationship, PK, FK, and so forth patterns. If, in this scenario, the *Concrete-table* seems to be a bad choice, it may be turned into a good choice, depending on the relationships, the number of estimated account records, what classes are persistent in the class tree, and/or presence of a legacy database schema. Some flexibility, to change between each approach, is a valuable asset for this solution.

Most ORMSs implement some of the three inheritance patterns, but with distinct names, as shown by Table 4. The *Concrete-table* support is optional in *JPA,* and is not available in the *EF* visual editor. The *Cayenne* team plans to support *Concrete-table* in a future version. The *RAR* implements only the *Single-table* strategy, by introducing a discriminating column named *inheritance_column* [35].

The *MyBatis* framework requires the specification of the SQL expressions for each mapping, and offers the *discriminator* mapping that uses one table column to discriminate the correct class of the hierarchy. The discriminator can be utilized to implement a *Single-table* strategy, but it is possible to emulate the *Class-table* by manually providing adequate SQL expressions. However, the discriminator column is required for all "type select" situations.

The discriminator column may be required depending of ORMS and pattern in use. With *JPA*, a discriminator is required for *Single-table*, but optional for *Class-table* inheritance. In *Ruby*, the discriminator may not exist for *Single-table*, but it means that the instance type will not be automatically detected. It is sometimes possible to identify the record type in the *Single-table* strategy by the presence of null values, but it is not really supported by the researched tools, due to its poor performance.

*SQLAlchemy* allows fine control on fetching *Class-table* objects, but requires a *discriminator* column in the root table. When retrieving *Account*, for instance, only the *Accounts* table may be queried, and the remaining attributes are loaded on demand, avoiding the expensive join. With the *discriminator*, the persistence framework can determine which type must be instantiated, without performing an outer join to each possible sub-type.

The special case of *Concrete-table* mapping where the superclass is abstract, or not persistent, but its persistent specializations persist the inherited properties, is named *mapped superclass. Mapped superclasses* are mapped with one table for each concrete subclass, and no concrete super classes. *Doctrine* supports *mapped superclasses*, but does not implement *Concrete-table; JPA* and *ODB* (*ODB* names it as *reuse inheritance*) distinguish *mapped superclasses* from *Concrete-table* mappings; the others do not make distinction.

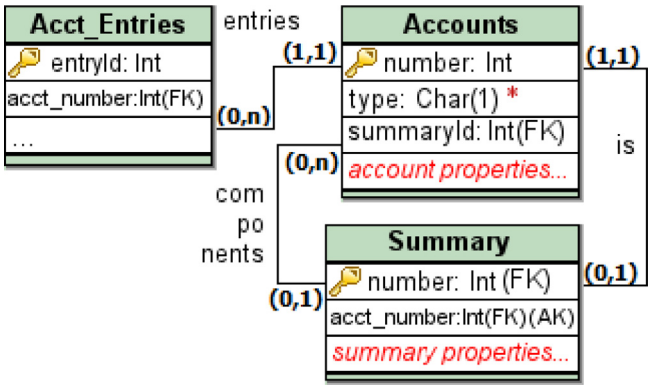


**Fig. 17.** Independent keys in *Class-table* inheritance example.

This distinction is relevant if you need a *mapped superclass* that is not abstract at the OOPL level, but that is not persisted.

Another issue rises in the joining of inheritance tables. *EF* requires that the PK, of each table, must be the same, and a FK to the master table (Fig. 15). The *JPA* specification states that the PK should be specified only in the root entity, but at same time, allows the FK to the super-class to be redefined by subclasses, using the *PrimaryKeyJoinColumn* annotation. The effect of this redefinition, in *JPA,* may have unexpected consequences depending on the implementing framework, because it is not possible to declare in the mappings the PK of the subclass. *SQL-Alchemy* explicitly allows user defined FK relationship between each *Class-table* (raises Q25).

Fig. 17 shows an example of *Class-table* mapping with independent PK and a discriminator (marked with ∗) column. Flexibility in the FK *inheritance mapping* is important to map legacy database relationships as inheritance relationships, when the FK relationship is not in the PK, but an alternate key.

None of the consulted ORMS explicitly allows mixing the three strategies within the same hierarchy (raises Q27). Fig. 18 introduces the *SpecialAccount* into the account hierarchy, using the *single-table* strategy, although *SummaryAccount* uses the *class-table* strategy. In the left diagram, the classes are siblings, whereas in the right *SpecialAccount* inherits from *SummaryAccount*. Both models can be mapped to the schema in the center diagram, with type as the discriminator column.

JPA does not require the implementing framework to support multiple strategies, and the strategy is defined at the parent class, what makes it difficult to define a distinct strategy for a sibling. Hibernate orients its users to use secondary tables to emulate *class-tables* within *single-table* mappings, such as presented in Fig. 18. Doctrine and EF specific states that one strategy is defined per hierarchy. SQLAlchemy does not mention this option, but it limits the number of discriminators at one per hierarchy. ODB allows mixing reuse and polymorphic inheritance, what in practice means using a *mapped superclass,* but not mixing the distinct polymorphic strategies.

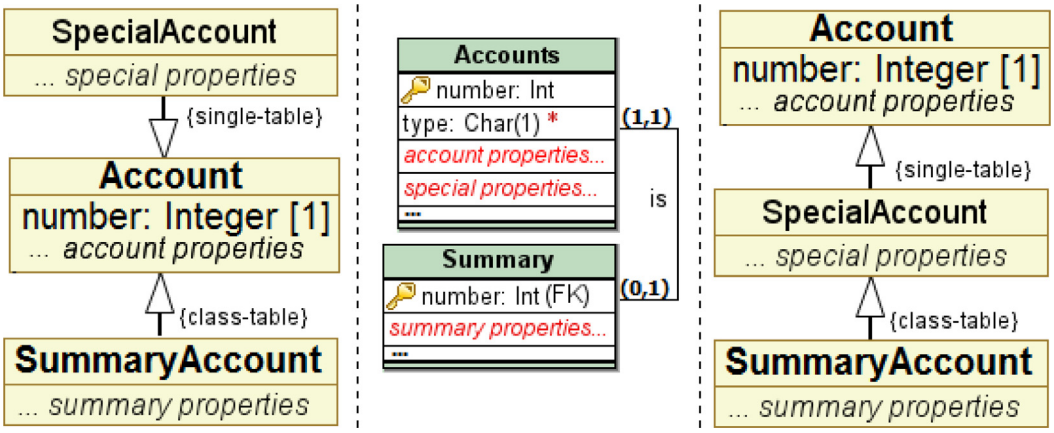


**Fig. 18.** Mixing strategies in the same inheritance on siblings (left) or ancestors (right), and the mapped schema (center).

**Table 5**
ORMS support for each proposed criterion.

| Criteria\Frameworks | | RAR | Cayen. | EF | JPA 2 | MyBatis | SA | ODB | Doc. | BS |
|---|---|---|---|---|---|---|---|---|---|---|
| Coupling | Loose | No | No | Dep. | Yes | Yes | Yes | No | Yes | No |
| | External conf. | No | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes |
| Mapping | Metadata Mapper | Yes | Yes | Yes | Yes | No | Yes | Yes | Yes | Yes |
| | Extract metadata | Yes | No | No | No | No | No | No | No | Yes |
| | Uses reflection | Yes | Yes | No | Yes | Yes | Yes | No | Yes | Yes |
| | Visual models | No | Yes | Yes | No | No | No | No | No | No |
| Identity | Auto-generation | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| | Counter gen. | Yes | Yes | No | Yes | Yes | Yes | Em. | Yes | No |
| | Compound keys | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| | PK as Field(s) | Yes | Yes | Yes | Yes* | Yes | Yes | No | Yes | Yes |
| | PK Class | No | No | No | Yes | Yes | No | Yes | No | No |
| FKs | Bidirectional | Yes | Yes | Yes | Yes | Em. | Yes | Yes | Yes | Yes |
| | Collection update | Yes | Yes | Yes | Yes | Em. | Yes | Yes | Yes | Yes |
| | Navig. weak entity | No | No | Yes | Yes | Yes | Yes | No | Yes | No |
| | Fetch strategy | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No |
| | Proxy option | No | No | No | Yes | No | Yes | Yes | Yes | No |
| Transparent Assoc. Table | | Yes | Yes | Yes | Yes | Em. | Yes | Yes | Yes | Yes |
| Embed. Value | Are Supported? | Yes | Yes | Yes | Yes | Em. | Yes | Yes | Yes | No |
| | Map at aggregated | No | Yes | Yes | Yes | No | No | Yes | Yes | No |
| | Non scalar props. | No | No | No | Yes | No | No | No | No | No |
| | Embed. collections | No | No | No | Yes | No | No | No | No | No |
| Inheri-tance | Single-table | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No |
| | Class-table | No | Yes | Yes | Yes | Em. | Yes | Yes | Yes | No |
| | Concrete-table | No | No | Yes | Dep. | Em. | Yes | Yes | No* | No |
| | Join class-table | No | Flex | PK | Flex* | Em. | Flex | Yes | No | No |
| | Mix strategies | No | Yes | No | No | Yes | No | No | No | No |

* A characteristic has special situation discussed below.

*MyBatis*, however, supports mixing *Single-Table* and the *Class-Table* emulation. Cayenne explicitly allows mixed strategies using *flattened attributes*, despite not supporting the *Concrete-Table* strategy. Both solutions achieve the mapping by requiring a discriminator column at the root class, not stating the strategy employed at the specialization, and allowing the use of joined secondary tables to implement the *Class-Table* strategy. Despite not being supported by the studied ORMSs, the mapping of all three strategies at the same hierarchy is possible, as proposed by the $M^2ORM^2$+*HIE* model [53].

#### 4.7.1. Discussion

Inheritance is one key feature of ORM, and the most difficult issue to deal with. The mapping strategy to be chosen is a decision that affects behavior, performance, and design limitations of the domain model. Some of these limitations are dependent of the ORM tool, and others are inherent to the strategy itself (raises Q26).

Mapping inheritance in legacy *schemas* may be difficult, because certain design conditions must be met to support inheritance, and these conditions change from tool to tool. Moreover, not all constraints can be enforced by the database for every strategy, and if there are other non ORM based applications, using the same schema, the RDB will not prevent the storage of invalid data.

### 4.8. Results

Table 5 presents a summary, with each analyzed problem criterion, and its pattern characteristics, related to each ORMS analyzed over the previous sections. The information summarized by Tables 3 and 5 relate resources and decisions common to the studied persistence frameworks, helping with designing and documenting applications, as well as migrating applications across distinct ORMSs.

The first criterion is the domain *coupling*, in which some frameworks offer *loose coupling* and others are tied by inheritance *cou-*

*pling*. The *EF* is usually *not loose coupled*, although with some customization it is possible to design a loose coupled domain, hence its marked with *depends (dep.)*. The *external configuration* characteristic includes files and annotations in code, both requiring maintenance when the code changes, which is often not automatically verified.

The *mapping* criterion is determinant for the mapping abstraction level. With *metadata mappers* the developer builds the mappings directly with SQL statements, therefore some of the criteria can be satisfied by emulating (em.) their mapping requirements. The other characteristics are if ORMSs *extract metadata* from the RDBs; if they *use reflection* to build *metadata*; and if they use *visual models* as an input of *metadata* from the developers.

On the *identity* criterion, *auto-generation* fields are supported by all ORMSs, but *counter generation* (gen.) is not always included. Among the ORMSs that support *compound keys*, they can be mapped with *PK as fields* of the class, or with a *PK class* referenced by the class. *JPA* allows the mapping of composite *PK as fields*, but requires a *PK class* nevertheless.

On the *FK* criterion, all ORMSs support bidirectional mappings; collection update refers to adding objects from the *one-to-many* side of these bidirectional mappings. Navigation from the weak entity (*Navig. weak entity*) tells when a FK placed in the PK will be a direct reference to a persistent object. *Fetch startegy* is true when it is possible to choose between using or not lazy load, and *proxy option* when *many-to-one* and scalar variables can also be lazy loaded.

For embeddable values *(embed. Value),* it is important to know if the mapping can be also done in the embeddable class (*map at aggregated*); and if the embeddable class can have *non scalar properties,* such as references to other persistent classes, and they are mapped using FKs. *Embedded (embed.) collections* are equivalent to the *one-to-many* relation to embeddable values.

## 5. Conclusion and future work

The popularization of object-relational mapping solutions may give the false idea that these tools can automatically solve all persistence problems, when in fact they just offer resources to the developers dealing with the impedance mismatch problem. There is not a silver bullet for this problem, but well prepared software designers and developers.

The pattern literature covers a wide range of scenarios for object-relational mapping, but it was not clear how these patterns are applied by distinct object-relational mapping solutions. Nevertheless, analysis models, agnostic to platform details such as the impedance mismatch problem, have to be adapted to be used under these solutions.

This paper identifies, discusses, and organizes the knowledge concerning existing object-relational mapping solutions, with the purpose of helping designers towards making better and informed decisions concerning the Impedance Mismatch Problem. Nine object-relational mapping solutions were studied, selected from the top ten list of most popular programming languages, using the pattern literature as the base to identify a common language, and challenges shared among all solutions.

Despite the differences between object oriented languages and relational databases, the analyzed object-relational mapping solutions have several similarities with each other, mostly following the existing patterns in the literature. The differences among the framework are more about flexibility and completeness than totally new concepts for the Impedance Mismatch Problem. By knowing such patterns in advance, and their cross platform characteristics, the designer can anticipate mapping decisions, keeping them documented for future use and anticipating refactoring due to mapping limitations, or platform changes.

As a contribution of this paper, we proposed a set of pattern based criteria to asses object-relational mapping solutions, and a set of questions that modelers should ask when implementing their analysis models. We also identified many similar solutions by their patterns that can help developers to migrate from one solution to another.

This paper focused at the design of structural models that encompasses the changes required to map classes and tables, and cope with the impedance mismatch problem. Future work can discuss the design of the behavioral models, and the dialects used to query objects, relating object-relational mapping solutions and other patterns in the literature.

## Acknowledgments

This work was partially financed by CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) and CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico).

## References

[1] G. Copeland, D. Maier, Making smalltalk a database system, SIGMOD Rec. 14 (1984) 316–325 doi: 10.1145/971697.602300.
[2] D. Maier, Representing database programs as objects, in: Advances in Database Programming Languages, ACM, 1990, pp. 377–386.
[3] S. Ambler, Agile Database Techniques: Effective Strategies for the Agile Software Developer, first ed., Wiley, USA, 2003.
[4] M. Fowler, Analysis Patterns: Reusable Object Models, Addison-Wesley Professional, Boston, MA, USA, 1996.
[5] G. Hohpe, C. Easy, SOA Patterns–New Insights or Recycled Knowledge? (accessed December 26, 2015).
[6] P. Atzeni, C.S. Jensen, G. Orsi, S. Ram, L. Tanca, R. Torlone, The relational model is dead, SQL is dead, and I don't feel so good myself, SIGMOD Rec 42 (2013) 64–68 doi: 10.1145/2503792.2503808.
[7] J.W. Yoder, Q.D. Wilson, M. Douglas, R.E. Johnson, Connecting Business Objects to Relational Databases, EUA, Monticello IL, 1998.
[8] H.W. Buff, Why Codd's rule no. 6 must be reformulated, SIGMOD Rec. 17 (1988) 79–80.
[9] S.W. Ambler, Building Object Applications that Work: Your Step-By-Step Handbook for Developing Robust Systems with Object Technology, first ed., Cambridge University Press, 1998.
[10] K. Brown, B.G. Whitenack, Crossing Chasms: a pattern language for object-RDBMS integration: the static patterns, in: Pattern Languages of Program Design 2, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996, pp. 227–238.
[11] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA, 2002.
[12] W. Keller, Mapping objects to tables - a pattern language, in: Proceedings of the 1997 European Pattern Languages of Programming Conference, Irrsee, Germany, 1997.
[13] M. Fowler, OrmHate, http://martinfowler.com/bliki/OrmHate.html (n.d.). (accessed 26.12.2015).
[14] T. Neward, Interoperability Happens - The Vietnam of Computer Science http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx, (accessed 26.12.2015).
[15] OMG, UML 2.5, (2015). http://www.omg.org/spec/UML/2.5/ (accessed 2.12.2015).
[16] M.P. Atkinson, O.P. Buneman, Types and persistence in database programming languages, ACM Comput. Surv. 19 (1987) 105–170 doi: 10.1145/62070.45066.
[17] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik, The Object-Oriented Database System Manifesto (1989).
[18] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, first ed., Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1995.
[19] W. Keller, Object/relational access layers - a roadmap, missing links and more patterns, in: Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing, universitäts verlag konstanz, 1998, pp. 1–25.
[20] D. Smith, A Brief History of TopLink, (2003). http://www.oracle.com/technetwork/topics/history-of-toplink-101111.html (accessed 9.12.2015).
[21] IDC, Enterprise Database Management Systems Market Forecast and Analysis, 2000-2004, IDC, USA, 2000 www.idc.com.
[22] M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, F. Velez, The Object Data Standard: ODMG 3.0, Morgan Kaufmann Publishers Inc, 2000.
[23] B. Karwin, SQL Antipatterns: Avoiding the Pitfalls of Database Programming, first ed., Pragmatic Bookshelf, Raleigh, N.C., 2010.
[24] T.-H. Chen, W. Shang, Z.M. Jiang, A.E. Hassan, M. Nasser, P. Flora, Detecting performance anti-patterns for applications developed using object-relational mapping, in: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, 2014, pp. 1001–1012.

[25] P.A. Bernstein, S. Melnik, Model management 2.0: manipulating richer mappings, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 2007, pp. 1–12.
[26] R. Lämmel, E. Meijer, Mappings make data processing go 'round—an inter–paradigmatic mapping tutorial, Gener. Transform. Tech. Softw. Eng. (2006) 169–218 vol. 4143 of LNCS.
[27] C. Ireland, D. Bowers, M. Newton, K. Waugh, A classification of object-relational impedance mismatch, in: Advances in Databases, First International Conference on, IEEE Computer Society, Los Alamitos, CA, USA, 2009, pp. 36–43.
[28] W. Keller, J. Coldewey, Relational Database Access Layers - A Pattern Language, in: Proceedings PLoP '96, Allerton Park, 1996.
[29] L. DeMichiel, JSR-000338 Java Persistence 2.1 - Final Release, (2013). https://jcp.org/aboutJava/communityprocess/final/jsr338/index.html (accessed 27.12.2015).
[30] C. Begin, B. Goodin, L. Meadors, iBatis in Action, first ed., Manning Publications, New York, 2007.
[31] Code Synthesis, ODB - C++ Object-Relational Mapping (ORM), (2015). http://www.codesynthesis.com/products/odb/ (accessed 29.12.2015).
[32] Microsoft, Entity Framework, The Official Microsoft ASP.NET Site. (2015). http://www.asp.net/entity-framework (accessed 26.12.2015).
[33] Apache Foundation, Apache Cayenne, (2015). http://cayenne.apache.org (accessed 27.12.2015).
[34] M. Bayer, SQLAlchemy - The Database Toolkit for Python, (2015). http://www.sqlalchemy.org/ (accessed 17.12.2015).
[35] D.H. Hansson, activerecord | RubyGems.org | your community gem host, (2015). https://rubygems.org/gems/activerecord/ (accessed 22.12.2015).
[36] Doctrine team, Welcome to Doctrine 2 ORM's documentation!—Doctrine 2 ORM 2 documentation, (2015). http://www.doctrine-project.org/ (accessed 25.12.2015).
[37] T. Griesser, Bookshelf.js, (2015). http://bookshelfjs.org/ (accessed 2.12.2015).
[38] Apple inc, Core Data Programming Guide: What Is Core Data?, (2015). https://developer.apple.com/library/tvos/documentation/Cocoa/Conceptual/CoreData/index.html (accessed 23.12.2015).
[39] R.S. Pressman, B.R. Maxim, Software Engineering: A Practitioner's Approach, eighth ed., McGraw-Hill Publishing, 2015.
[40] C. Bauer, G. King, Hibernate in Action, second ed., Manning Publications, New York, 2004.
[41] J. Martin, Managing the Data-Base Environment, Prentice-Hall, Englewood Cliffs, NJ, USA, 1983.
[42] M. Fowler, Martin Fowler Bliki: POJO, (2000). http://www.martinfowler.com/bliki/POJO.html (accessed 7.12.2015).
[43] Microsoft, EntityObject Class (System.Data.Objects.DataClasses), (2015). http://msdn.microsoft.com/en-us/library/system.data.objects.dataclasses.entityobject.aspx (accessed 7.12.2015).
[44] J. Derstadt, D. Vega, POCO Proxies Part 1 - ADO.NET team blog - Site Home - MSDN Blogs, (2009). http://blogs.msdn.com/b/adonet/archive/2009/12/22/poco-proxies-part-1.aspx (accessed 5.12.2015).
[45] U. Dayal, P.A. Bernstein, On the correct translation of update operations on relational views, ACM Trans. Database Syst 7 (1982) 381–416 doi: 10.1145/319732.319740.
[46] A.M. Keller, Algorithms for translating view updates to database updates for views involving selections, projections, and joins, in: Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, ACM, New York, NY, USA, 1985, pp. 154–163.
[47] Red Hat Middleware, Hibernate Tools - JBoss Community, (2014). http://hibernate.org/tools/ (accessed 7.12.2015).
[48] rubyonrails.org, Active Record Nested Attributes, (2015). http://api.rubyonrails.org/classes/ActiveRecord/NestedAttributes/ClassMethods.html (accessed 17.12.2015).
[49] A. Adya, J.A. Blakeley, S. Melnik, S. Muralidhar, Anatomy of the ADO.NET entity framework, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, ACM, Beijing, China, 2007: pp. 877888. doi: 10.1145/1247480.1247580.
[50] J. Gosling, B. Joy, G.L. Steele, G. Bracha, A. Buckley, The Java Language Specification, first ed., Java SE 8 Edition Addison-Wesley Professional, 2014.
[51] H. Garcia-Molina, J.D. Ullman, J. Widom, Database Systems: The Complete Book, second ed., Prentice Hall, Upper Saddle River, New Jersey, USA, 2008.
[52] R. Elmasri, S.B. Navathe, Fundamentals of Database Systems, seventh ed., Pearson, 2015.
[53] L. Cabibbo, A. Carosi, Managing inheritance hierarchies in object/relational mapping tools, in: O. Pastor, J.F.E. Cunha (Eds.), Advanced Information Systems Engineering, Springer, Berlin Heidelberg, 2005, pp. 135–150.