ELSEVIER

Information and Software Technology 44 (2002) 923–941

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Generating three-tier applications from relational databases: a formal and practical approach

Macario Polo*, Juan Ángel Gómez, Mario Piattini, Francisco Ruiz

Escuela Superior de Informática, Paseo de la Universidad, 4, 13071 Ciudad Real, Spain

Accepted 8 August 2002

Abstract

This article describes a method for building applications with a three-tier structure (presentation, business, persistence) from an existing relational database. The method works as a transformation function that takes the relational schema as its input, producing three sets of classes (which depend on the actual system being reengineered) to represent the final application, as well as some additional auxiliary classes (which are 'constant' and always generated, such as an 'About' dialog, for example). All the classes generated are adequately placed along the three-tiers.

The method is based on (1) the formalization of all the sets involved in the process, and (2) the mathematical formulation of the required functions to get the final application. For this second step, we have taken into account several well-known, widely used design and transformation patterns that produce high quality designs and highly maintainable software.

The method is implemented in a tool that we have successfully used in several projects of medium size. Obviously, it is quite difficult for the obtained software to fulfill all the requirements desired by the customer, but the uniformity and understandability of its design makes very easy its modification.

Keywords: Database reengineering; Code generation; Pattern formalization

1. Introduction

Reverse-engineering is one of the most important activities of the software maintenance process. Its goal is to obtain the representation of a system in a higher level of abstraction than the original [3]. Nowadays, reverse-engineering is often being used to recover conceptual and architectural models of old systems that are later used as the basis to 'forward engineering' the system to new environments and paradigms, as the Internet, client–server, object-oriented systems, distributed computation, component-based software, etc. According to Arnold [2], this whole process of reverse-and-forward is also known as reengineering. In both techniques, the software engineer is shifting the software product representation from one abstraction level to another.

In the rigid sense of the Arnold's concept of reverse-engineering, after every step, a faithful representation of the product in the preceding abstraction level must be produced, without introducing changes in its structure or behavior. During the forward stage, however, restructuring techniques may be used at some abstraction levels to change the product characteristics by adding new functionalities, improving its performance, changing its representation language, etc. i.e. classes with 15 methods instead of 30, Java instead of C++, UML instead of OMT, etc. (Fig. 1).

Although it is the most frequent, source code is not always the starting point of reengineering: in fact, several authors have studied and proposed techniques for reverse-engineering other types of products, such as relational databases. Their goal is to obtain a conceptual diagram representing the original problem domain, usually as an entity-relationship (ER) or extended ER diagram.

In outline, ER diagrams represent the structural relationships among the entities of the universe of discourse being modeled. In the last years, some database designers have started using other modeling languages to represent conceptual schemas, as UML class diagrams, since this language covers a broader spectrum, is more expressive (as a matter of fact, there are several proposals for mapping ER

* Corresponding author. Tel.: +34-926-2953500; fax: +34-926-295354.
E-mail address: macario.polo@uclm.es (M. Polo).

0950-5849/02/$ - see front matter
PII: S0950-5849(02)00130-1

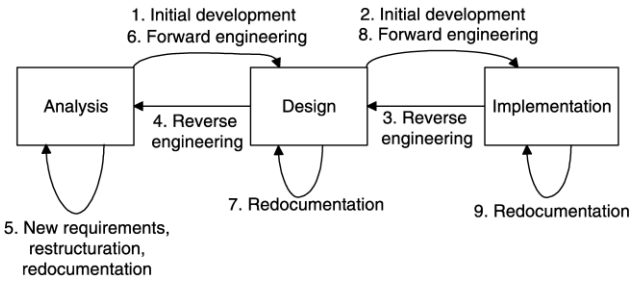

Fig. 1. A simplified reengineering model.

to OO schemas, i.e. Refs. [9] or [14]) and designers need only to learn one modeling language.

The use of a class diagram representing the conceptual schema of a relational database instead of an ER allows the incorporation of object-oriented constructions (as behavior) at this early stage, that can be used as the basis for the forward engineering stage. Moreover, as the nature of the object-oriented paradigm avoids finding a clear boundary between analysis and design, and methodologies for object-oriented development actually take this fact into account, it would be advisable to use class diagrams as the starting point for the forward stage of reengineering projects instead of the ER approach. In this manner, the software engineer can take advantage of some of the many analysis, design or architectural patterns proposed in the literature for restructuring the reverse-engineered software product.

This article presents a complete reengineering method to obtain a high quality three-tier application from the relational schema of a database, and a tool implementing such method: a class diagram representing the possible conceptual schema is built from the relational database. This class diagram is used in the forward engineering stage as the basis for generating a three-tier application through the use of a set of well-known design and transformation patterns (such as the *expert* or the *pure fabrication* design patterns, and the *one class*, *one table* and the *one inheritance tree*, *one table* transformation patterns). A minimal, but far from negligible, set of functionalities is supplied by the generated application.

The method is based on mathematical descriptions of all sets and functions involved in the reengineering process. In this sense, Manfred Broy claims that Software Engineering, as any other engineering discipline, needs mathematical descriptions for its modeling aspects, description techniques and development methods [6]. For this author, such mathematical theory must not be too complex, since then it would not be a very helpful contribution for software engineers: the theory should give semantics to usual description techniques (as class diagrams, state charts, etc.) and should explain methods from a mathematical rational, more in the form of a 'formal description technique' (FDT) than as a 'formal method'. This should lead to a deeper understanding of software processes and, what in our opinion is quite important, to a basis for more powerful support tool. Other authors that follow this same principle have also used a similar approach to recover and maintain a system's architecture, proposing the use of the 'relation partition algebra', that describes systems in terms of sets, binary relationships and operations [25], using familiar algebraic terminology.

According to the Broy's idea of leading to a powerful support tool, our work does not only remain in the proposal of the FDT, but it also contains a tool supporting the entire method.

The transformation functions for both the reverse and the forward engineering stages provide mathematical descriptions of the afore-mentioned architectural, design and transformation patterns, that produces highly maintainable software. This is an important issue because the generated code will probably suffer modifications in order to fulfill all the needed functional requirements, that will not be completely covered by the generation process, that is, the application generated has foundation enough to be modified using what some authors have called 'proactive maintenance' [26].

Besides our concrete method and tool, one of the main ideas and conclusions of our work derives from the previous point: as patterns are good solutions for common, recurrent problems, mathematical descriptions of them are welcome for providing them with tool support, making the work of software engineers easier and cheaper. Some authors (cf. [17]) have provided formal descriptions of design patterns, but in our opinion they fall into the problem advanced by Broy, that is excessive mathematical notation (although the fact is that these authors need them in VDM++, which is a formal object-oriented language).

The article is organized as follows: Section 2 is a brief study of some related works; in Section 3, the whole transformation process (reverse and forward engineering, this one for code generation) used in our method is depicted and explained in natural language; Section 4 presents the formalization of the used sets and functions, illustrating with some examples. Then, the tool we have built and used is presented in Section 5, including some architectural details. Section 6 draws our conclusions and future lines of work.

2. Background

Applying a reengineering process to a database can be due to the desire of migrating it from an old data model (Codasyl, for example) to a new one, to the change of database management system, to the detection of integrity errors, etc. Three sequential stages are followed during the development of the database: conceptual design (representing the information schema independently of the system where it will be saved), logical design (the conceptual schema is translated into an optimized logical schema depending on the

final data model, as relational or object-relational) and physical design (the logical schema is translated into a set of static and dynamic structures now depending on the final target system, management system, etc.).

Several authors have made different proposals for reverse-engineering relational databases. Hainaut et al. [15] propose a general method to apply reverse-engineering to any database system or files collection, consisting of four big processes, inverse to the forward engineering stage: (1) data structure extraction; (2) data structure conceptualization; (3) untranslation, that detects compliant constructs (with respect to the database management system) and replaces them by independent constructs; and (4) conceptual normalization, that recovers the high level structures produced during the forward stage.

Other approaches in this sense are those of Shoval and Shreiber [27] and Chiang et al. [12]. The first work presents a method that obtains a binary relationship diagram from a relational database. The main drawbacks of this work, such as it is presented, are the lack of automation of the process of data input (the information on the relational schema must be manually introduced), and the short utilization of the binary relationship model for conceptual data modeling. Moreover, this would lead to devote a big effort for adapting the proposed algorithms to other conceptual models more widely used. In the second work, an algorithm to get an extended ER schema is proposed. It uses some schema information (relation names, attribute names and primary keys), the actual data saved in the database and questions to the user to obtain the schema, including inclusion dependencies. These ones are inferred by the algorithm, but finally determined by the user. This interaction is not required in our algorithm, since we use more information from the data dictionary, such as relationships among tables. Another difference with our method is that we also use the column data types, since the class diagram we obtain requires this information for the forward engineering stage.

Pedro de Jesus and Sousa [8] also presented an interesting study analyzing the characteristics of several proposals (required initial state of the database, output products…) and presented a method allowing the mixed use of all of them. Eight methods of different authors are analyzed: except one of them [23], that produces an OMT class diagram, the others produce ER or extended-ER conceptual schemas. These methods produce their best results when the legacy data fulfill a set of desirable preconditions, as to be at least in the third normal form, lack of inconsistencies, etc. The method by Pedro de Jesus and Sousa takes advantage of all the previous ones because it obtains a set of database clusters, every one of them grouping elements according to their suitability for being reverse-engineered with the use of one of the methods they analyze. At the end, their 'macromethod' also produces an ER schema, that is used as starting point for the construction of the new database.

In the context of federated database systems, Castellanos [11] obtains object-oriented specifications of the relational databases belonging to the system, using the data on the database and the corresponding metadatabase. The obtained model is represented in BLOOM, an object-oriented specification language that captures different types of generalization and aggregation relationships. The model is used for increasing the knowledge about the individual databases of the system. Differences with our method are the target product (in both cases a class model is obtained, but in our case is an intermediate product, being executable code and maybe a restructured database our final product), and the language used for representing the conceptual schema (UML instead of BLOOM). Our approach also differ in the reverse-engineering stage, since we do not use the actual data in the database, only those in the metadatabase.

Andersson [1] extracts the possible ER schema used to build the database analyzing the data manipulation language, looking for among the SQL statements embedded in the source code of the programs that use the database. So, for example, primary and foreign keys are identified through the study of the join conditions existing in the SQL statements.

The models obtained by the algorithms proposed by these authors can be sometimes richer than ours, but only from a merely conceptual point of view. Our method, however, is more powerful than these ones in the sense of that it uses information on the data types, produces a standardized conceptual model that is supported and operable by common, commercial tools, and that also generates easily maintainable executable code.

As users generally manage the information kept in a database through a set of external programs, the structure of the documents where the database is represented (i.e. data models) has a strong influence on the corresponding documents representing the programs (i.e. functional models). The object-oriented paradigm increases the cohesion of both models unifying data and behavior under a common point of view. So, the ER schema obtained from the reverse-engineering stage could be translated into a class diagram (excepting when a class diagram is obtained, as in Refs. [11] or [23]) in order to take advantage of object-oriented constructions, tools, etc. for the next forward stage.

Object-oriented conceptual schemas can be improved with formal notations and languages, as object constraint language (OCL) [30], T_{ROLL} [16] or OASIS [21] generating completely executable applications. In some cases, their formal notations can be hidden beyond a standard graphical notation, as UML. Our approach will be much more modest than these, since what we want is to produce executable applications with just some functionalities, but with high maintainability in order to enable their modification without too much effort.

Some of the tools for generating code that can be used are Rational Rose (from Rational Software Corporation

[24]) and Together (from Together Soft [29]), which automatically generate different types of code from class diagrams. Rational Rose 2000, for example, comes with an add-in that provides a round-trip engineering process from SQL92 to UML class diagrams, and vice versa. However, it is difficult to give the class model all the details that the add-in requires to generate the schema (associations must be navigable in one direction only, the length must be specified for some data types, etc.). Inheritance relationships, although supported in the class model, are not directly represented in the database as it is usual (that is, with primary keys propagation): primary keys are generated adding new columns to each table that do not exist in the class model (instead of having some mechanism to highlight those fields in the class corresponding to the primary key, such as a stereotype), and a table is always built for every class (although this is the most usual, other options are possible, as we say in Section 3.2). An additional drawback is that classes must be specifically stereotyped for Oracle, a fact that impedes the use of the same model for generating code for other targets, as C++ or Java programs. In spite of these problems, this add-in is a powerful aid for developers, that can use some of the tools included in Oracle 8 for building their applications (Developer or JDeveloper, for example).

Several integrated development environments have wizards that build programs directly from the physical database: Microsoft Visual Studio 6.0 includes one for accessing a database and to generate a single application allowing data listing and the most common operations (insert, delete, etc.) with records. It takes advantage of several ActiveX components that produce non-portable code. Persistence responsibilities are monolithically mixed with user screens (although depending on the options that the user selects, some code can be placed in internal classes, directly coupled to the database and without any correspondence between the code and the database). In this case, a class is generated for every selected table, but without relationships among them. In any case, the class model obtained is not at all similar with the possible class or ER model initially used as conceptual diagram for the further construction of the database. On the other hand, navigation across related records in different tables is not allowed.

The tool we present in Section 5 builds a class diagram representing the conceptual schema of the database being reverse-engineered. The tool considers this class diagram as the *business* tier of the application to be generated, and it is therefore the basis for the automatic generation of all the code, what includes, besides the business tier, also presentation and persistence. Moreover, the class diagram is saved in Rational Rose format, what allows its restructuration, correction, etc.

3. Design and transformation patterns

It is a good idea to use some pattern while focusing on the development of a new application with a strong database support. Software patterns are good solutions for frequent problems, and the implementation of tools for managing relational databases is one of the most common problems that small and medium software companies must deal with. For this issue, the use of a three-tier architecture is a good pattern to give the application a robust architecture.

3.1. Three-tier architecture

Fig. 2 shows a specimen of the class structure of an application, designed using three-tier architecture. The middle tier is called business tier, and is mainly composed by all objects that are a meaningful part of the problem's universe of discourse. Usually, it contains a wide set of *persistent* classes, that are those which must survive systems

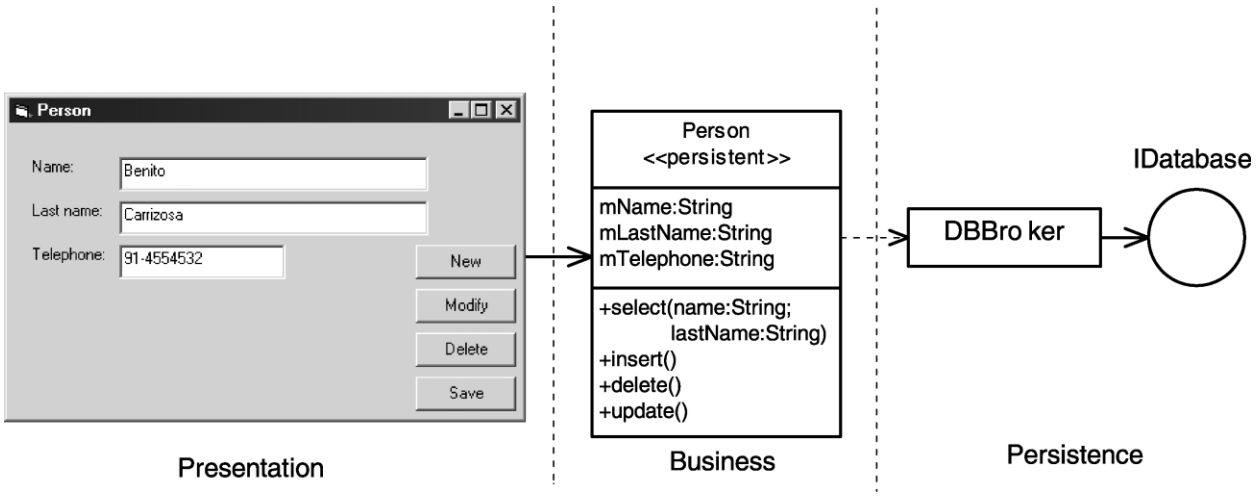

Fig. 2. A specimen of a three-tier application.

stops, keeping their instances in some kind of database. Objects in this tier are hidden for the user, who only can manipulate them via a set of screens, located on the *presentation* tier. This is composed by all available user screens, and has direct access to business objects through their public members. Persistent objects, located in the business tier, access the database via a set of classes located in the *persistence* tier. In the example of Fig. 2, a broker [7] has been put into this tier to centralize all communications among business objects and the database.

In order to separate the complexity of the business logic from their presentation, business objects are not directly related to those in the presentation tier. This allows the development of software that is more maintainable and reusable, to devote specialized people to the development of each tier, etc. [18]. Changes in the state of business objects are communicated to the interested objects in the presentation tier via intermediate objects, that often correspond to instances of the *observer* pattern [13].

3.2. Transformation patterns

The set of persistent classes in the business tier is often used as the conceptual diagram for the next design stage of the database. Although in the last years, a couple of new types of databases have emerged (active, object-relational and object-oriented), most enterprises are still using relational technology to save their data [19]. With this fact in mind, several transformation patterns can be used to translate a conceptual diagram to a relational schema.

1. Through the one class, one table pattern, one relational table is built for each class in the business tier (Fig. 3b). Some relationships among tables are translated into new tables (to represent, for example, many to many relationships), whereas others are represented via foreign key constraints.
2. Through the one inheritance tree, one table, one relational table is used to provide persistence to a complete inheritance tree (Fig. 3c). With this transformation, there is a correspondence from many business classes to just one table, which implies that the table will probably have a large number of NULL value columns.
3. Through the *one inheritance path, one table*, one relational table is built for each path in an inheritance tree (Fig. 3d).

In practice, it is difficult that only one transformation pattern be suitable for the whole business model, and the previous patterns are applied to different fragments of the diagram. Our method and tool is capable of transforming class diagrams into relational databases using these tree patterns individually; we are now studying the automation of the combined use of them.

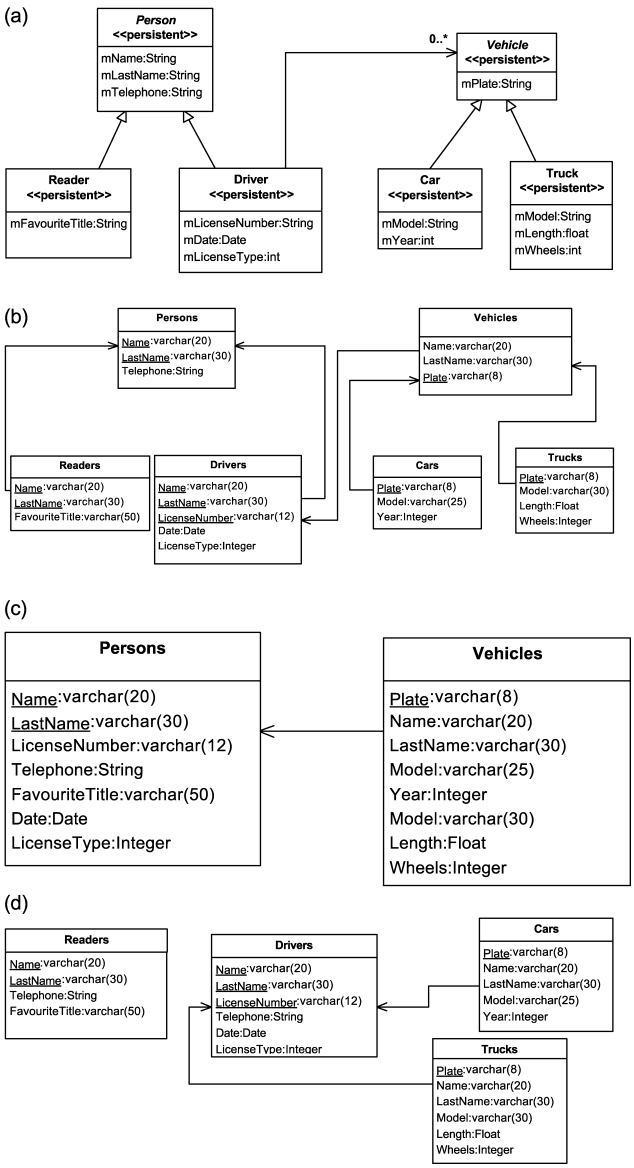

Fig. 3. A class model (a) and some possible transformations to relational schemas (b–d). (a) Original class model as conceptual data schema. (b) Transformation according to the one class, one table pattern. (c) Transformation according to the one inheritance tree, one table pattern. (d) Transformation according to one inheritance path, one table pattern.

3.3. CRUD pattern

A set of methods must be given to every persistent class to manage its own persistence. The minimal set of methods is known as *CRUD* operations [22,31]: *Create*, to save the information of an instance in the corresponding database table or tables, and that may correspond to an *insert* SQL statement; *Read*, to build instances from the information saved in the database, that may correspond to a *Select*; *Update*, to refresh the information of the object in the database; and *Delete*, to remove the instance information from the database.

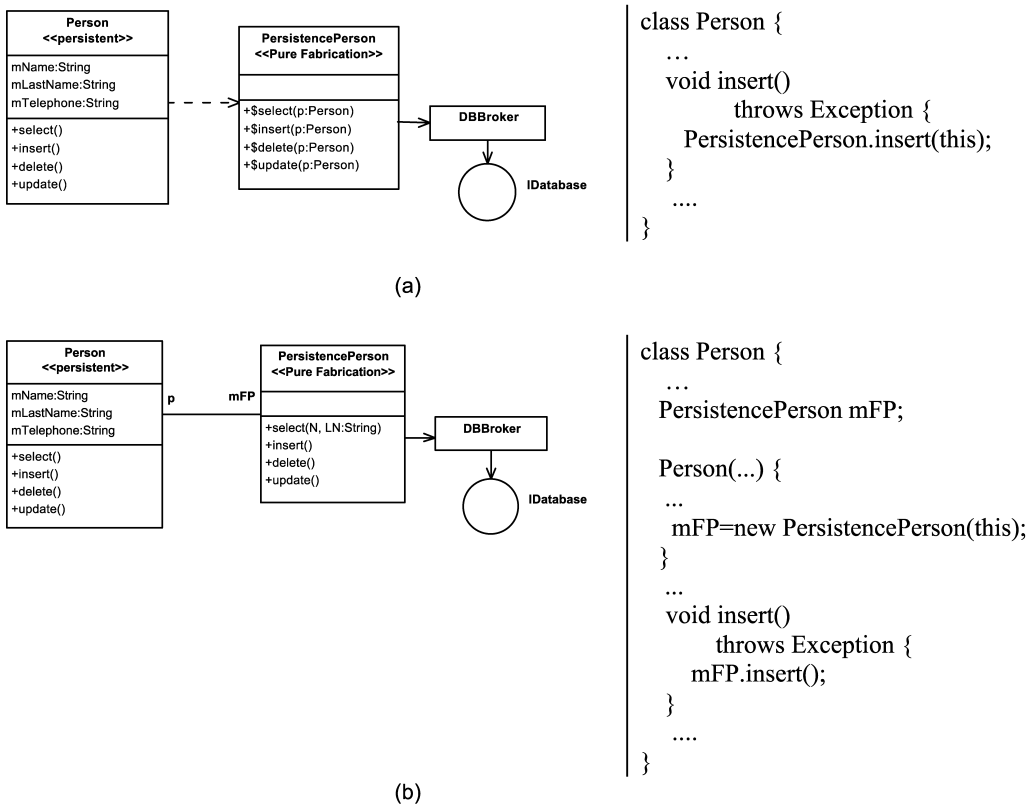

Fig. 4. Two different implementations of the pure fabrication pattern, used in this example to delegate persistence operations to associated classes. (a) Delegation to static methods. (b) Delegation to an associated instance.

The CRUD methods may be located in their own classes or may be delegated to associated classes—pure fabrications, as Larman calls them [18]—in order to keep the high cohesion and low coupling of the business classes. Fig. 4 shows two possible ways to delegate the persistence operations to associated classes, although there are many more (i.e. [22]). Furthermore, with pure fabrications, business classes are not directly coupled to either the database or to the broker, which removes any possible dependence on the right tier.

In any case, either using pure fabrications or the expert pattern [18] to implement persistence operations in the self class, the persistence instance itself is responsible of calling persistence operations, maybe in response to an external stimulus (a button pressed by the user, a periodic event triggered by a clock), or due to the insertion of a new instance of one of its subclasses (as it happens when a Reader is inserted in the database of Fig. 3b).

Any of these behaviors may be described using an interaction diagram in which the final implementation language is not taken into account. Supposing the class diagram of Fig. 4b, a possible sequence diagram for inserting an instance in a normal scenario of person could be that shown in Fig. 5.

As can be seen, the sequence diagram for inserting any other persistent instance would be exactly the same, requiring only the substitution of the names of the involved classes and parameters. Regarding the code, the implementation of the *insert* method is quite similar in all classes, since there are variations only in the names of the table, columns, and in the name of the class fields. The transformation pattern used for building the relational schema from the class diagram is the most influencing factor for this issue:

- In Fig. 3b, the insertion of a *Reader* requires the prior creation of a transaction for inserting the corresponding instance of *Person*, then the insertion of the *Reader* and the closing of the transaction, perhaps taking some decisions in the middle if something fails.
- In Fig. 3c, the insertion of the same *Reader* requires to put null values in those columns that proceed from the *Driver* class.
- In Fig. 3d, no transaction needs to be opened for the single insertion of just a *Reader*, since its corresponding table contains just the information required by these instances. However, this pattern could introduce integrity problems (the telephone of the same physical *Person* could be different in tables *Readers* and *Drivers*).

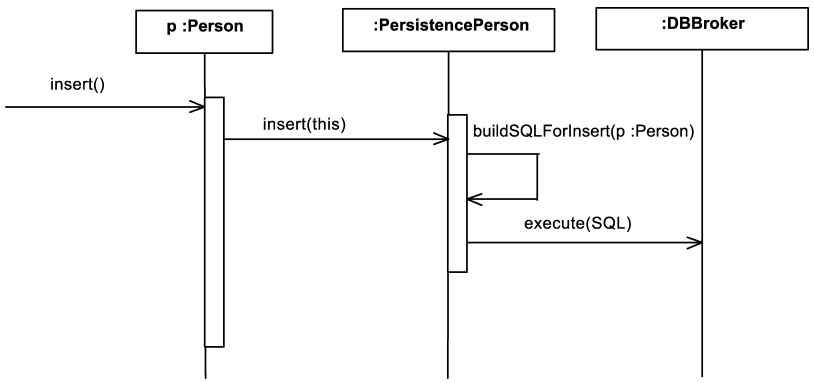

Fig. 5. Sequence diagram for inserting a person.

3.4. Minimal behavior

Thanks to the three-tier architecture, the problem of persistence responsibility assignment is transparent to presentation tier designers: in fact, from the persistence point of view, only the interface offered by persistent classes is required to perform their job. A minimal set of functionalities in the presentation tier should allow the user the execution of all persistence methods and the navigation among related instances (these are rows in the relational database).

The general behavior of the presentation tier can be also described using interaction diagrams. A minimal set of functionalities could be the following:

- The application must give the user the possibility of opening any table in the database and to see a list with all its records.
- The user can select any row in any list of records and see the specific data of the selected row in a 'card' window.
- Card windows must give the user the possibility of creating, updating and deleting instances.

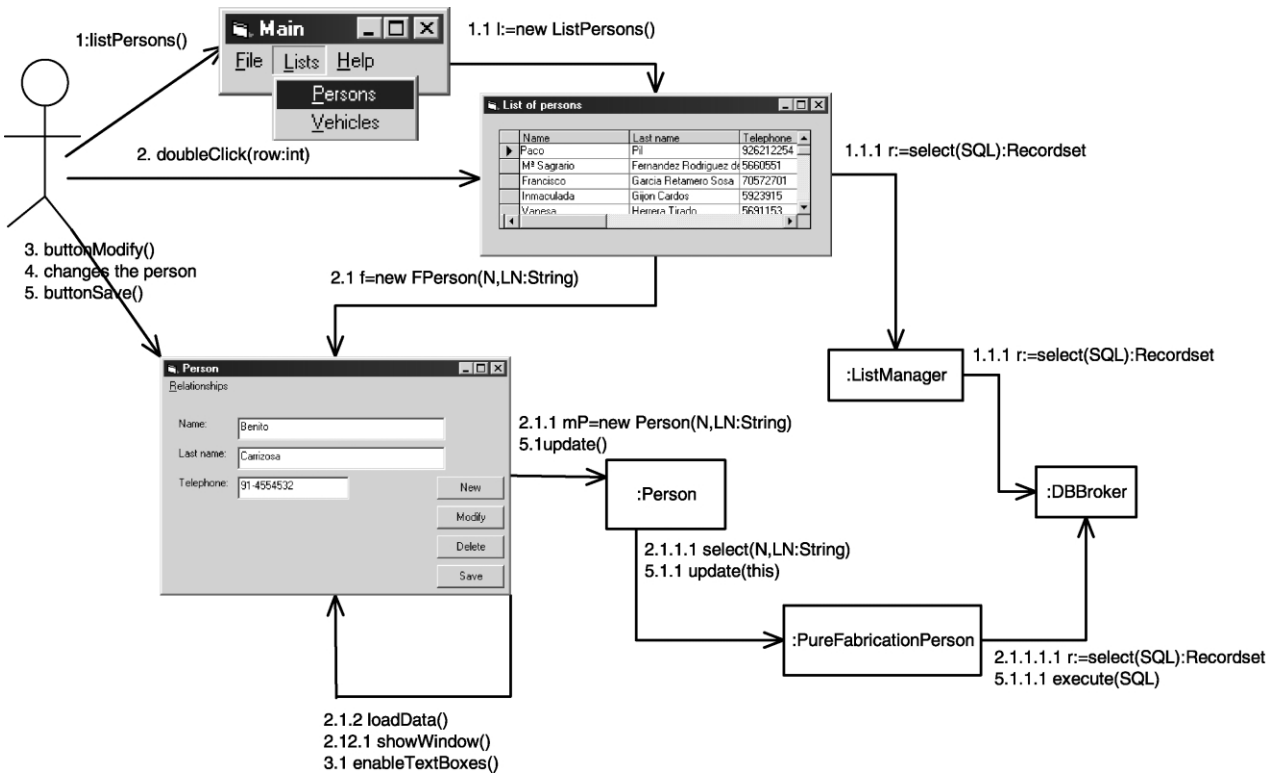

Fig. 6. Collaboration diagram for describing some functionalities.

Fig. 6 uses a collaboration diagram to show some of these functionalities. Observe, for example, that when a card screen is loaded to show a person, the user needs to press the *Modify* button to enable textboxes and change the data. At this moment, the program knows that if the *Save* button is pressed, then the instruction to be executed on the associated instance of *Person* must be *insert* (), whereas it would be *update* () if the pressed button would have been *New*. The behavior, of course, could be quite different. This one is just a proposal.

With the previous discussions and some experience, any software engineer can transform a class diagram into a whole well-built, well-architectured and easy-to-maintain application providing a minimal (but not scornable) set of functionalities. Additionally, if an automatic tool would be provided for this forward engineering activity, much recurrent work would be saved. Clearly, the transformation process is so automatic, that it is not too difficult to give it an implementation. As we mentioned in Section 1, it is much better to support it with a mathematical basis.

4. Formalization of sets and functions

In this section, we provide formal definitions of all the sets involved in the transformation process: relational databases and three-tier applications. The approach for defining a software system used in Ref. [4] is suitable for our goal: a system S is a pair composed by modules (that will be tables or classes in our case) and relationships (that will be foreign key constraints, associations, etc. in our case):

$$S = (M, R), \qquad R \subseteq MxM.$$

4.1. Formal description of a relational database

From our reverse-engineering point of view, a relational schema is a graph composed by tables, which are its nodes, and foreign key constraints, which are the arcs. Other characteristics of the database, as triggers or check constraints are not currently taken into account neither in the method nor in the tool, although we are analyzing the inclusion of these ones and of other elements, such as data types defined in the database. Therefore, with this consideration:

$DB = (Tables, FKS)$, where *Tables* is the set of tables, and *FKS* is the set of foreign key constraints. A foreign key constraint is defined between two tables (the parent and the child) to restrict the possible range of values of one or more columns in the child table to the set of existing values in the corresponding set of columns of the parent table. As all foreign keys impose constraints among existing tables, then $FKS \subseteq Tables \times Tables$.

- $t \in Tables = (Name, Columns, Indexes)$, where *Name* is the name of the table and *Columns* is a set of columns. Each element of *Columns* is defined as: $c \in Columns = (Name, Type, allowsNotNULL, isPK)$, that corresponds to the column's name and type, whether it allows null values and is part or not of the primary key. *Indexes* is the set of alternative keys (unique indexes) in this table, and is composed by subsets of columns.
- $f \in FKS = (tMain.Name, tSec.Name, cMain, cSec)$, where $tMain \in Tables$ is the child table (that with the primary key or unique index involved in the constraint); $tSec \in Tables$ is the child table (which has the foreign keys); $cMain \subseteq tMain.Columns$ is the set of columns in the child table which constitute the primary key or alternative key; and $cSec \subseteq tSec.Columns$ are the set of columns which compose the foreign key in *tSec*, and whose possible values are restricted to those in *cMain*.

4.1.1. Basic operations

We will use the notation *Element.Subelement* to simplify calls to operations on *Element* that must return the element whose name is *Subelement*. For example, if $t \in Tables$, then *t.Name* returns the name of *t*. We will also use this notation to return a set: for example, given $t \in Tables$, the expression *t.Columns* returns the set of columns in the table *t*.

Moreover, we define a number of functions working on relational schemas:

$getPks(t \in Tables) = \bigcup_{\forall c \in t.Columns} (c / c.isPK = true)$, which returns all the primary key columns in *t*. This function may be expressed as an algorithm:

```
getPks(t ∈ Tables) : Columns {
  R = ∅
  ∀c ∈ t.Columns {
    if (c.isPK) R = R ∪ {c}
  }
  getPks = R
}
```

The following function returns whether the table has or not primary key: $hasPks(t \in Tables) = (getPks(t) > 0)$, which returns *true* or *false* depending on the existence of primary key columns in *t*.

The following returns all primary key columns in the *t* table, but as non-constrained columns, that is, it returns the set of primary key columns in *t*, but allowing null values and not being primary keys. This function is useful for adding to a table a set of foreign keys pointing to the primary keys of another table:

$$getPksAsFks(t \in Tables) = \bigcup_{\forall c \in t.Columns} ((c.Name, c.Type, true, false) / c.isPk = true)$$

Its algorithmic version is:

```
getPksAsFks(t ∈ Tables) : Columns {
   R = ∅
   ∀c ∈ t.Columns {
      if (c.isPk)R = R ∪ (c.Name, c.Type, true, false)
   }
   getPksAsFks = R
}
```

4.2. Formal description of a three-tier application

According to Section 3, we want to generate a three-tier application to manage a relational database with a minimal set of functional requirements, mainly related to the execution of persistence operations and navigability across records. In our discussion, every persistent class has a corresponding card screen in the presentation tier (Fig. 2) in charge of allowing its visualization and manipulation by the user, as well as an associated class in the persistence tier with the unique purpose of managing its own persistence (Fig. 4). Therefore, a tier-application like those described consists of three class models, which will be the first concept to be formalized in this subsection. Then, connections among them will be presented.

4.2.1. Formal description of a class model

Through a textual description, a class model is seen as a set of classes and interfaces mutually related. Both classes and interfaces consist of a set of fields and methods. Interfaces can be considered as pure abstract classes, since all their methods must be abstract; classes may be or not be abstract, since their methods may have a body. In interfaces, an initial value is given to fields, whereas this is not mandatory in classes.

Classes are related to classes and interfaces via associations, aggregations and dependence relationships; inheritance relationships are allowed from one class to another, whereas classes may be related to interfaces via implementation relationships. As an interface may be considered as a pure abstract class, implementations may be also considered as inheritance relationships. Interfaces may be also related to other interfaces via implementations, which is similar to an abstract class that inherits from another abstract class.

Although several programming languages, as Smalltalk or Java, have constructors with a direct correspondence with the general object-oriented concepts previously mentioned, it is also true that such concepts may be understood from a conceptual point of view, and then to map them to other types of programming languages: interfaces do not exist in C++, but they can be translated as pure abstract classes, converting implementations into inheritance. Other concepts, as multiple inheritance, may be adequately implemented in programming languages that do not support it.

With these ideas in mind, it is possible to provide a formal description of a class model in an algebraic way. In fact, in the line of Ref. [4], a class model can be viewed as a system composed of classes, interfaces and relationships among them:

$M = (C, R)$, where C is the set of classes and interfaces and R is the set of existing relationships among elements in $C : R \subseteq C \times C$.

The set of relationships is: $R = (A, D)$, being A the set of associations of aggregations, and D the set of dependences. The formal modeling of inheritance and implementation relationships is considered in the formal description of C's elements, that is, in the description of classes and interfaces.

The two following two epigraphs provide formal descriptions of C and R.

4.2.1.1. Formal description of C, the set of classes and interfaces. Assuming that interfaces are abstract classes, then the elements in C can be defined as:

$c \in C = (Name, Fields, Methods, Parents, isInterface, Tier)$, where *Fields* is the set of the class fields; *Methods* is the set of methods; *Parents* is the set of parent classes and implemented interfaces by c; *isInterface* is a boolean value that indicates whether c is or not an interface; and *Tier* is the name of the tier where the class is located.

We can analyze these components in more depth, giving details on the definition of each field ($f \in Fields$) and each method ($m \in Methods$).

- $f \in Fields = (Name, Type, Visibility, correspondsToPk)$, where *Name*, *Type* and *Visibility*, respectively, are the name, type and visibility of the field. As we are focused on the translation of class models into other class models (that is, a restructuration) and into relational schemes, we add the element *correspondsToPk* to the definition of f to indicate whether the field corresponds to a column that will be part of the primary key. In a class diagram written, for example, in UML, this information can be saved with the field as a field's stereotype, by a convention in its name or by any other method.
- $m \in Methods = (Name, Type, Visibility, Arguments, isStatic)$, where *Type* is the class of the result returned by the method; *Arguments* is a set composed by pairs (*Name*, *Type*); $isStatic \in \{true, false\}$.
- $Parents = \{p \in C / p$ is a parent of or an interface implemented by this class$\}$
- $isInterface \in \{true, false\}$

With these descriptions, the set C remains as follows:

$$C = \begin{matrix} \{(Fields_1, Methods_1, Parents_1, isInterface_1), \\ (Fields_2, Methods_2, Parents_2, isInterface_2), \\ \cdots \\ (Fields_n, Methods_n, Parents_n, isInterface_n)\} \end{matrix}$$

4.2.1.2. Formal description of R, the set of relationships. The set R includes associations, aggregations and dependence relationships. Actually, in the work described in this paper we give the same treatment to associations and aggregations, and this is the reason of putting them together into the same set, A. A possible distinction between them that we do not take into account would create cascade updates and deletes with aggregations, and 'set null' propagations with associations. On the other hand, dependences are temporary, casual relationships among classes and are located in a different subset of R, D. Then, $R = (A, D)$.

In this work, we only consider the existence and modeling of binary relationships, setting for future advances of this research the considering of higher-order relationships, as we point out in Section 6.

Let us describe the elements in these subsets of R:

- $a \in A = (cL, cR, isFinalL, isFinalR, cardL, cardR, Name, Fields)$, where $cL \in C$, $cR \in C$ represent the 'left' and the 'right' classes involved in the relationship; *isFinalL* and *isFinalR* are boolean values meaning whether *cL* or *cR* are final classes in the relationship, respectively, ('final' must be understood in the context of the association navigability); *cardL* and *cardR* represent the cardinalities of *cL* and *cR* in the relationship; *Name* is the possible name that the relationship has; and *Fields* is the possible set of fields of the association or aggregation. The structure of *Fields* has been described in Section 4.2.1.1.
- $d \in D = (cL, cR)$, where $cL \in C$, $cR \in C$. *cL* and *cR* represent the two classes involved in the dependence relationship, in the sense that *cL* depends on *cR*.

4.2.1.3. λ, a distinguished symbol. We will use λ to represent an empty element with the same sense as in automata and language theory (cf. [5]). For example, if an association *a* has no name, this circumstance is represented with the expression: $a.Name = \lambda$.

4.2.2. Formal description of the three-tier application

As a three-tier application is composed by three class models, it can be represented as:

$T = (U, B, P)$, where U represents the presentation tier, containing the classes in charge of interacting with the user; B is the class model representing the business tier, and P is the persistence tier, with all classes in charge of giving persistence to business objects.

The transformation function we must describe returns T from DB, being T the final application and DB the relational database: $f(DB) = T = (U, B, P)$. Let us carefully describe the behavior of f.

4.3. Transforming functions

This section describes the process of obtaining the whole final application from the relational schema. Class models representing every tier are obtained separately without making connections among them. Then, a second step establishes the needed relationships among classes in different tiers. A third, last step adds the needed behavior to all classes.

4.3.1. Obtaining the business class model

In this section, we describe the algorithm used to obtain a business class model representing the possible conceptual schema that was used during the development stage for building the relational database. The algorithm works as whether the one class, one table pattern had been used to build the relational database. Obviously, different patterns could have been used to build the database, and therefore, the obtained class model might not be a faithful representation of the original conceptual model: for example, if the database is that represented in Fig. 3c, our algorithm and tool would build a class diagram with only two classes related with a one to many association, instead of recovering the actual conceptual model used, shown in Fig. 3a. In spite of this, the obtained class model adequately describes the database structure and, thanks to the use of well-known patterns, the final application is easily modifiable. Moreover, our experience in the application of the tool shows that the most of the structure of most databases proceed from the one class, one table pattern; therefore, the loss of significance is usually reduced to a small piece of the diagram. Thereinafter, we are currently improving the method for detecting other possible patterns used for building the database, such as the one inheritance tree, one table pattern, what requires the analysis of actual data instances, as other researchers have done (Section 2).

When the one class, one table pattern is used for forward engineering, our algorithm builds a class for each table in the relational database, a field is added to the class for each column in the table, and those fields proceeding from the columns that compose the primary key are adequately highlighted.

The following algorithm returns the structure of the business class model (M) of the relational database passed as parameter.

```
1   getBusinessModel(B) : M {
2       M=(∅, ∅)   // (Classes, associations)
3       ∀t∈B.Tables {
4               Fields=∅
5               Arguments=∅
6               ∀col∈t.Columns {
7                       f=(col.Name, correspondingType(col), "#", col.isPk)
8                       Fields=Fields ∪{f}
9                       if (col.isPk)
10                        Arguments=Arguments ∪{ (col.Name, correspondingType (col)}
11              }
12              emptyConstructor=("empty", λ, "+",∅, False)
13              materializerConstructor=("materializer", λ, "+", Arguments, False)
14              insert=("insert", λ, "+",∅, False)
15              delete=("delete", λ, "+",∅, False)
16              update =("update", λ, "+",∅, False)
17              Methods={ emptyConstructor, materializerConstructor, insert, update, delete }
18              c=(t.Name, Fields, Methods, ∅, False, "Business")
19              M.Classes=M.Classes∪{c}
20      }
21      ∀fk∈B.FKS {
22              // The following four lines get, for clarity, the names of both tables involved
23              // in the foreign key constraint, as well as their respective sets of columns.
24              mainClassName=fk.tMain.Name
25              secClassName=fk.tSec.Name
26              mainCols=fk.cMain
27              secCols=fk.cSec
28
29              isAnInheritance= True
30              ∀col∈secCols {
31                      if ¬ col.isPK ∨ |secCols|≠|getPks(fk.tSec)|
32                              isAnInheritance=False
33              }
34
35              if isAnInheritance {
36                      c=findClass(secClassName, M.Classes)
37                      ∀f∈c.Fields {
38                              if f.Name∈getNames(mainCols)
39                                      c.Fields=c.Fields-{f}
40                      }
41                      c.Parents=c.Parents ∪ findClass(mainClassName, M.Classes)
42              } else {
43                      if secCols∈fk.tSec.Indexes
44                              cardR=1
45                      else
46                              cardR=N
47                      if mainCols∈fk.tMain.Indexes
48                              cardL=1
49                      else
50                              cardL=N
51                      c=findClass(secClassName, M.Classes)
52                      ∀f∈c.Fields {
53                              if f.Name∈getNames(mainColumns)
54                                      c.Fields=c.Fields-{f}
55                      }
56                      M.Associations=M.Associations ∪
57                          ∪(mainClassName,secClassName,True,False, cardL, cardR)
58              }
59      }
60      getBusinessModel=removeEmptyClasses(M)
61  }
```

As it is seen, lines 3–20 create and add a new class to the business model for each table in the relational schema, also adding a new field for each column in the table and a minimal set of methods:

- An empty constructor, to build instances with default values in all its fields.
- A 'materializer constructor', that takes as many parameters as there are columns in the primary key. This constructor is used to materialize instances from the information stored in the database.
- The *insert*, *update* and *delete* methods are used to manage the persistence of the instance.

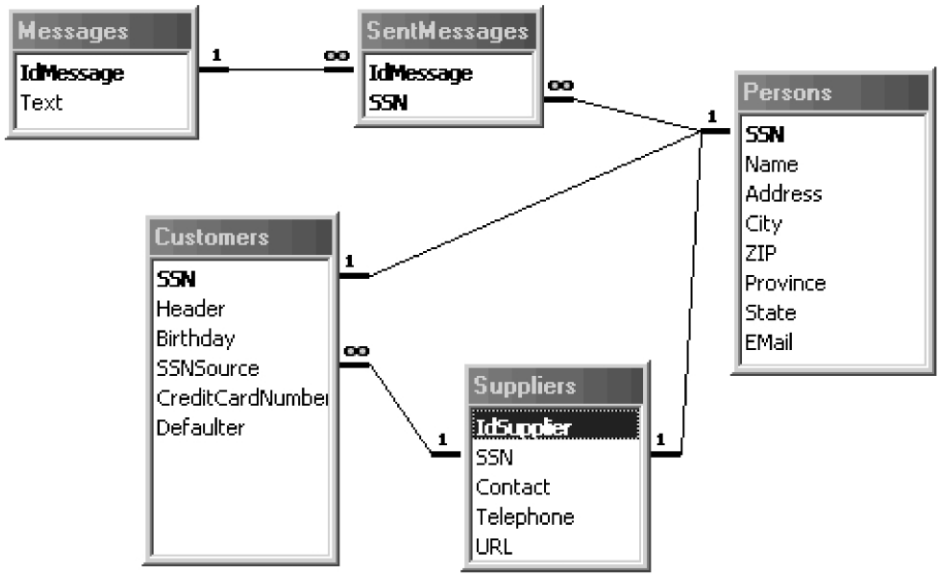

Fig. 7. A relational schema used as example.

After building all classes, foreign key constraints are processed in order to produce some kind of relationship between classes: in line 29, the algorithm supposes for a while that the current constraint is representing an inheritance relationship. This is confirmed or denied by the loop in lines 30–33, that works in following way:

If any column of this foreign key is not a primary key in the child table of the constraint, then the algorithm supposes that the constraint is representing an association relationship. Fig. 7 is a brief piece of one of the databases where we have applied our method and tool. There is a foreign key relationship between *Persons* (parent table) and *Suppliers* (child table) through the *SSN* column. Initially, this constraint is supposed to be representing an inheritance relationship; however, as SSN in *Suppliers* is not part of its primary key, the modeled relationship is passed as an association.

Even though all the columns in the child table of the foreign key are part of its primary key, if the algorithm sees that the number of columns in this foreign key constraint is different than the number of columns composing the primary key, then the constraint is also supposed to be representing an association relationship. In Fig. 7, this is the case of the foreign key constraint between *Messages* and *SentMessages*: *IdMessage* is the only column composing the set of columns in the child table of this constraint, and it is also part of the primary key of *SentMessages*. However, as the number of columns composing the primary key (which is two) is greater than the number of secondary columns (it is one), the algorithm considers that the constraint is also modeling an association between the class *Message* and *SentMessage*.

When the columns composing the foreign key constraint in the child table are all of them a primary key, then the algorithm considers that the constraint is modeling an inheritance relationship (this approach is also used in Ref. [20]). In Fig. 7, the SSN column in the foreign key constraint between *Persons* (main table) and *Customers* (secondary table) is a primary key, and it is also the whole primary key of the secondary table. Therefore, the algorithm supposes that this relationship is modeling an inheritance relationship between the class corresponding to the child table and the parent one.

The detected inheritance relationships are processed in lines 35–41: this piece of code adds the superclass to the set of parents of the child class, and removes from the child those fields proceeding from its parent. When the constraint has been reverse-engineered into an association, the algorithm checks whether the association cardinality must be 1, searching in the set of indexes of the table the cardinality of the foreign key (i.e. if it is an unique index, then the cardinality of the association is set to 1 in the class proceeding from the child table). Fields proceeding from the columns in the child table are also removed from its respective class, and an association is added to the model. Sequence columns do not receive a special treatment by the algorithm neither by the tool. In the generated application, the programmer must change the code to give the adequate treatment for the fields proceeding from these columns.

Sometimes, it is possible that some classes lose all their fields after modeling all their relationships with other classes. In this case, the class must be removed from the model and its' relationships must be substituted by others. These operations are made by the following algorithm, which is called in line 60 of the previous one:

```
removeEmptyClasses(M) : M {
    ∀c∈M.Classes {
        if c.Fields=∅ {
            ∀a∈M.Associations {
                if  a.cR=c ∨ a.cL=c {
                    ∀a'∈M.Associations {
                        if  (a'.cR=c ∨ a'.cL=c) ^ (a=a') {
                            if (a.cR=c ^ a'.cR=c)
                                newA=(a.cL,a'.cL,True,True,a.cardR,a'.cardR,λ, ∅)
                            if (a.cR=c ^ a'.cL=c)
                                newA=(a.cL,a'.cR,True,True,a.cardR,a'.cardL,λ, ∅)
                            if (a.cL=c ^ a'.cR=c)
                                newA=(a.cR,a'.cL,True,True,a.cardL,a'.cardR,λ, ∅)
                            if (a.cL=c ^ a'.cL=c)
                                newA=(a.cR,a'.cR,True,True,a.cardL,a'.cardL,λ, ∅)
                            M.Associations= M.Associations∪{na}-{a}-{a'}
                        }
                    }
                }
            }
        }
    }
    ∀c∈M.Classes {
        if c.Fields=∅
            M.Classes=M.Classes-{c}
    removeEmptyClasses=M
}
```

In the example of Fig. 7, the class corresponding to the *SentMessages* table losses its two fields because they are modeling two associations with *Messages* and *Persons*. *removeEmptyClasses* detects this situation and searches all associations where the *SentMessagges* class is involved; the algorithm adds a new association between *SentMessages* and *Persons* and, then, removes the empty class and the original associations from the final class model.

The resulting business model (after having manually written in singular the class' names) for the database in Fig. 7 is shown in Fig. 8.

```
getPersistenceTier(M) : M {
    broker=getBroker()
    broker.Tier="Persistence"
    Persistence= { broker}
    ∀c∈M.Classes {
            selectMethod=("select", λ, "+", ("o", c), True)

            insertMethod=("insert", λ, "+", ("o", c), True)
            deleteMethod=("delete", λ, "+", ("o", c), True)
            updateMethod Method=("update", λ, "+", ("o", c), True)
            pf=("PF"+c.Name, ∅,
                    {selectMethod, insertMethod, deleteMethod, updateMethod},
                    ∅, "Persistence")
            Persistence = Persistence ∪{ pf }
            M.Associations=M.Associations ∪{ (pf, broker, False, True, 1, 1, λ, ∅) }
            M.Dependencies=M.Dependencies ∪{(c, pf)}
    }
    M.Classes=M.Classes∪Persistence
    getPersistenceTier=M
}
```

4.3.2. Obtaining the persistence tier

In the persistence tier, we put a pure fabrication class for each business class, that implements all persistence operations. In our architecture, the database is accessed via a broker that centralizes all calls to the database. The broker always has the same structure for all the applications, and can be considered as a constant.

From the several possibilities for implementing pure fabrications for persistence, we have selected the first alternative of Fig. 4: a class containing static methods directly callable by business instances. An algorithm to generate this tier is:

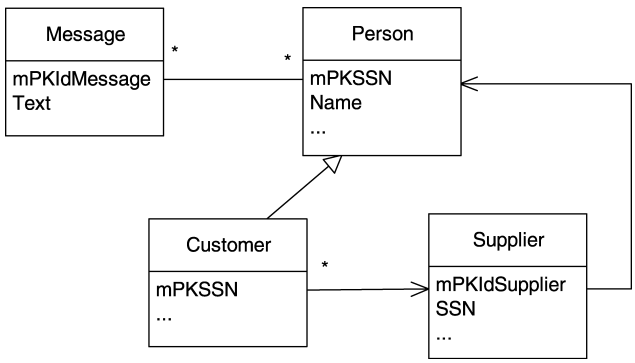

Fig. 8. Resulting business model.

This algorithm creates the associated class for each class in the business model, adding the four CRUD operations and the associations with the broker, that is known by all pure fabrications. The broker is built by the *getBroker* operation, which returns a 'constant class'.

According to our architecture, instances of business classes have only direct knowledge of other instances in the same tier, and of their corresponding pure fabrications for persistence, placed in the right side tier. These are the objects than can receive messages from business objects.

4.3.3. Obtaining the presentation tier class model

The business model obtained in the previous step is the required input for generating the presentation tier because, as we stated in Section 3.1 and followings, a card screen is built for each persistent class. The user must also be able to open a list of all related records to the instance currently shown.

In this tier, we also include a set of standard classes (dialogs to show messages and asking for confirmations, for example) and an initial screen with a menu to go to a list of all the records in all the tables (Fig. 6). Lists are also shown in a special screen that although it is standard as well, it is analyzed below individually. This screen knows the persistence tier through a *ListManager* class, that is placed in the business tier in order to maintain the absolute decoupling of the presentation tier from the persistence one.

The following algorithm creates the presentation tier of the application:

```
getPresentationTier(M) : M {
    M.Classes = M.Classes ∪ getStandardClasses()  // Standard dialogs, etc.
    MainScreen=("Main", ∅, ∅, ∅, False, "Presentation")
    ListClass=getListClass()   // See below the getListClass algorithm
    M.Classes=M.Classes ∪ {ListClass, MainScreen}
    ∀c∈M.Classes {
            // A method for listing a table is added to the main class.
            listOf_Method=("listOf" + c.Name, λ, "+", ∅, False)
            MainScreen.Methods=MainScreen.Methods ∪ { listOf_Method }
            // Every "card screen" has a number of methods to manage instances (section 3.4)
            // The last arguments returns the adequate sequence diagram representing this method
            newMethod=("New", λ, "+", ∅, False, getSDForNew(c))
            modifyMethod=("Modify", λ, "+", ∅, False, getSDForModify(c))
            saveMethod=("Save", λ, "+", ∅, False, getSDForSave(c))
            deleteMethod=("Delete", λ, "+", ∅, False, getSDForDelete(c))
            cardClass=("card"+c.Name, ∅,
                    {newMethod, modifyMethod, saveMethod, deleteMethod}, ∅, "Presentation")
            // Navigation methods are also added. getNavigationMethods is shown below.
            cardClass.Methods=cardClass.Methods ∪ getNavigationMethods(c, M)
            // A widget is created for each field in the class.
            ∀f∈c.Fields {
                    // widgetFor returns a suitable widget for this field (i.e.:TextField for Strings
                    // and numbers, a CheckBox for booleans, etc.
                    widget=(f.Name, widgetFor(f.Type))
                    cardClass.Fields=cardClass.Fields ∪ widget
            }
            M.Classes=M.Classes ∪ { cardClass }
            // The association between the card screen and the business class is established.
            M.Associations=M.Associations ∪ { (cardClass, c, False, True, 1, 1, λ, ∅) }
    }
        M.Classes=M.Classes ∪ Persistence
        getPresentationTier =M
    }
```

The algorithm to build the lists screen is the following:

```
getListClass() : Class {
    GetRowsMethod=("list", λ, "+", {("SQL", String)}, False)
    ShowRowMethod= ( "showRow", λ, "+", { ("selectedRow", Integer)}, False )
    CurrentTableField=("CurrentTable", "String")
    ListClass=("List",                  // Name of the class
      {CurrentTable},    // Current table (to know in what table to search when double-clicking)
      {GetRowsMethod, ShowRowMethod},
      ∅, False, "Presentation"}
    getListClass=ListClass
}
```

The next one adds to the card screen corresponding to the business class passed as parameter so many methods for navigating across related records as classes it knows:

```
getNavigationMethods(c, M) : Methods {
    Methods=∅
    ∀a ∈M.Associations {
            // If the left class of this association is c and is navigable from c to the associated class,
            // then the navigation method is added to the result.
            if a.cL=c ^ a.isFinalR {
                    showRelatedMethod=("show"+a.cR.Name,
                            λ, "#", ∅, False)
                    Methods=Methods ∪{ showRelatedMethod }
            }
            // The same with the right class.
            if a.cR=c ^ a.isFinalL {
                    showRelatedMethod=("show"+a.cL.Name,
                            λ, "#", ∅, False)
                    Methods=Methods ∪{ showRelatedMethod }
            }
    }
    getNavigationMethods=Methods
}
```

4.4. Example

When we apply the algorithms presented in the previous sections to the database of Fig. 7, they produce the class diagram of Fig. 8, which is saved in Rational Rose for doing possible modifications. This class diagram is also used as the basis for generating the three-tier structure shown in Fig. 9. This is not the final structure of the code representing the application, since the tool can be customized in order to incorporate more elements and characteristics, as we explain in Section 5.

The process for connecting the tiers is quite trivial, and this is the reason of not providing a formal description for it: every card screen has a field of the type of the business object it is showing, that is the receptor of the messages produced by the user; in the same manner, every business object is connected to its corresponding pure fabrication class, which is in charge of managing its persistence.

5. The tool

We have developed a tool that implements the reverse-engineering method explained in this article. It takes an ODBC data source or a Microsoft Access database as input parameter, and builds a three-tier structure for a possible application to manage the database. All sets involved in the transformation process (classes, relational database, models, etc.) have been modeled as classes, and operations have been translated into methods.

The tool has been developed in the Microsoft J++ language, since it is similar to Java and allows the easy integration of ActiveX components within the application. We use these components to easily work with Rational Rose models (which are accessible as ActiveX components), which is the intermediate point of work. The access to the metadatabase is got using some interfaces provided in the *java.sql* package (Fig. 10), as *DatabaseMetaData*, that implements for example the methods *getTables* and *getColumns*); in the case of Microsoft Access, we use the Microsoft Data Access Objects library, since some of the methods of *java.sql* are not implemented by this database manager. The next release of the tool will be implemented in the new J# language of the Microsoft.NET platform.

Through the 'Reverse' button, the tool builds a class diagram representing the possible conceptual diagram used to build the database (or a piece of it), as was explained in Section 4.3.1. Then, the user builds an executable three-tier application with a minimal set of functionalities by pressing the 'Forward' button. The user can also make a number of

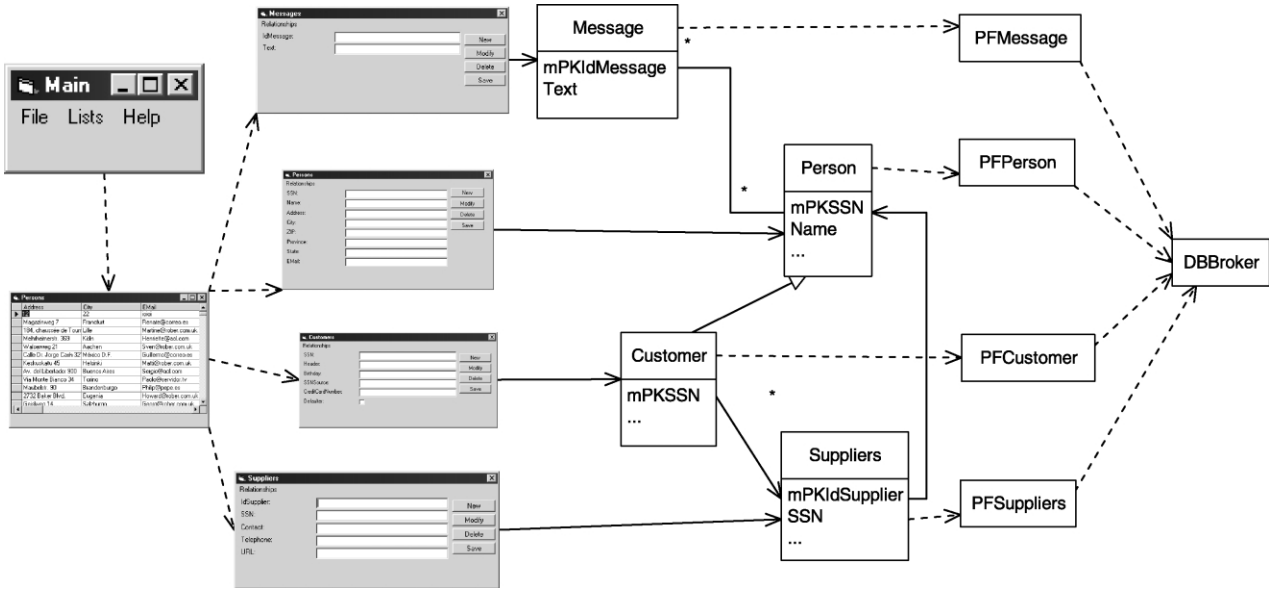

Fig. 9. Structure of the three-tier application obtained from the database of Fig. 7.

choices to select some characteristics of the application to be generated. As is shown in the right side of Fig. 11, the tool can also generate a new relational schema from the business class model using any of the three transformation patterns mentioned in Section 3.2.

The architecture of the subsystem in charge of the forward engineering process is depicted in Fig. 12: on its left side, there is a set of classes representing the structure of the generated three-tier applications. The self three-tier application class has references to three interfaces, that can be implemented by as many code generators as programming languages, platforms, development environments, etc. for which we want to provide code. The delegation of generation responsibilities to external classes makes the tool very expandable.

All the classes used by the tool implement the *Serializable* Java interface, that allows the structure used to represent the three-tier application or only one of the tiers to be kept in a secondary storage media. Moreover, the tool can also save the whole application or just a tier into a Rational Rose model, that is very useful for performing changes of the reverse-engineered model.

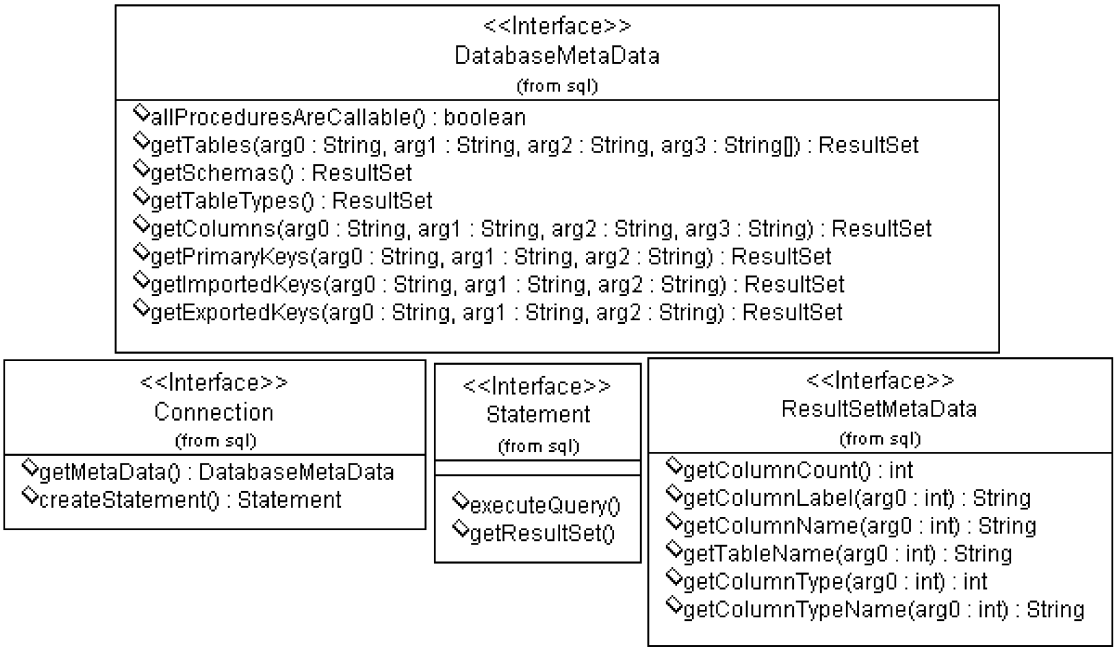

Fig. 10. Some operations in the interfaces provided by the *java.sql* package.

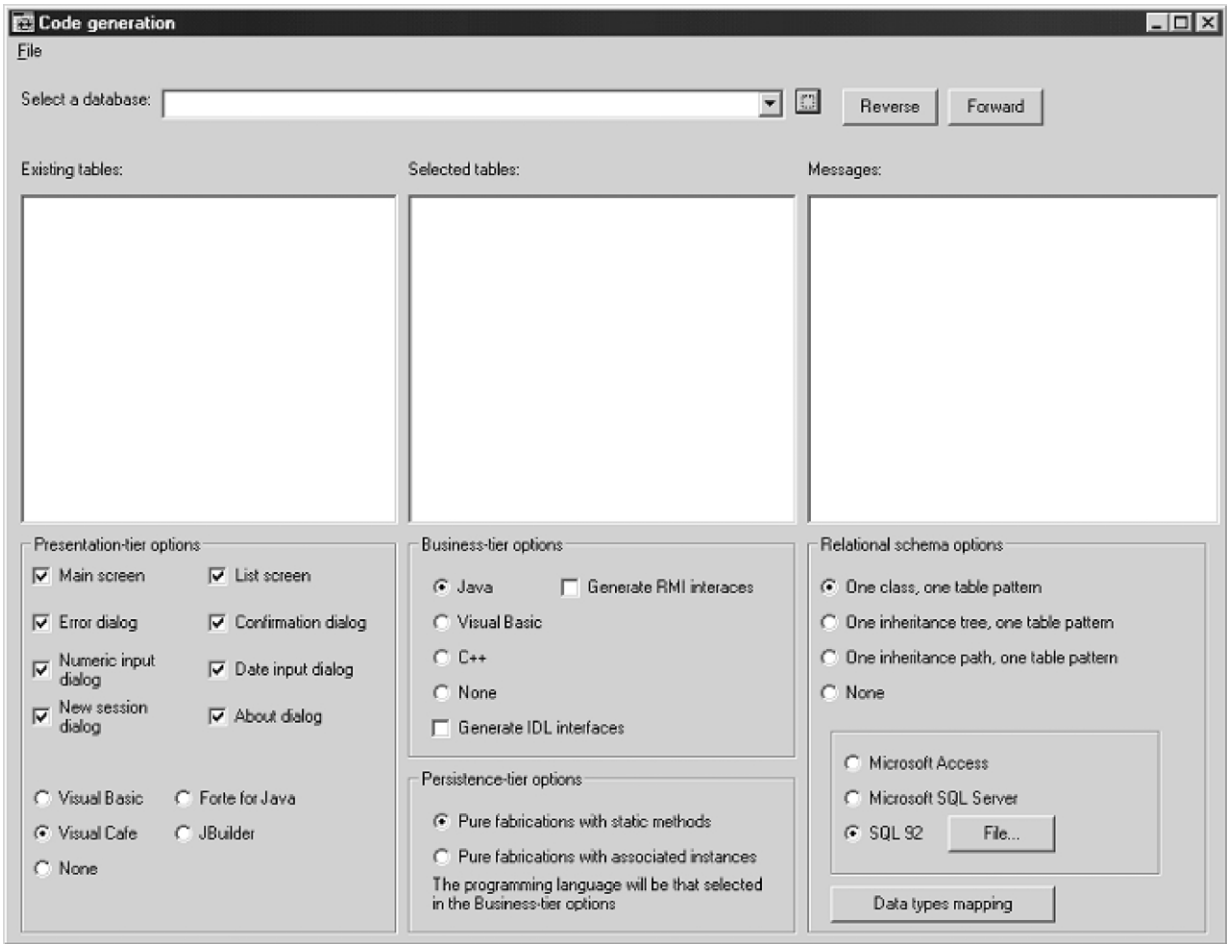

Fig. 11. Main screen of the tool.

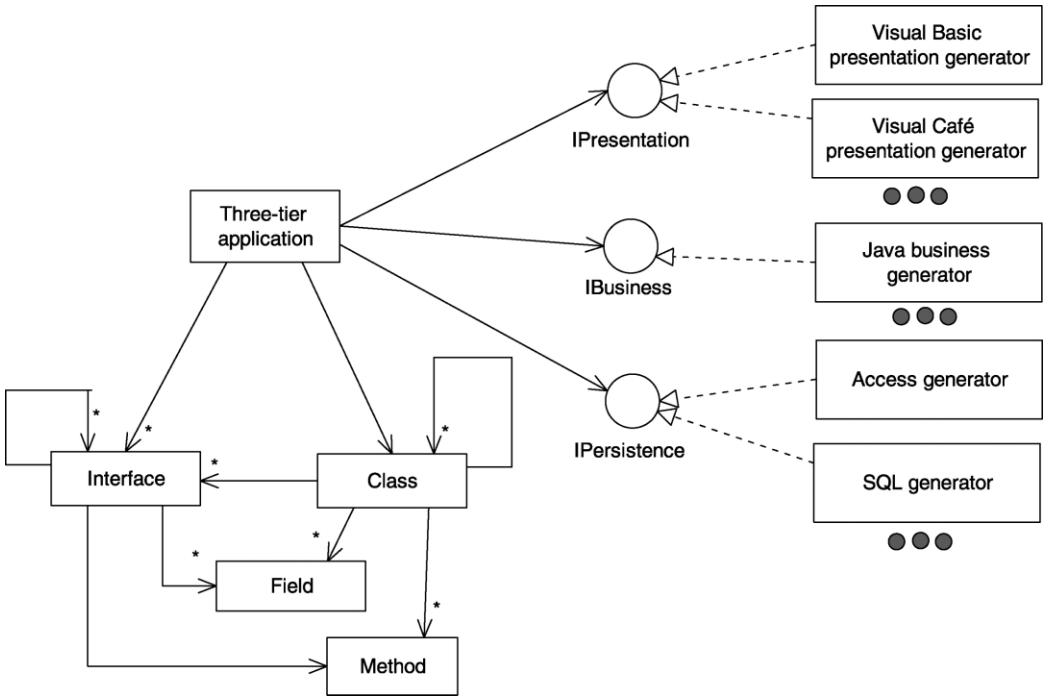

Fig. 12. A fragment of the tool architecture.

The tool has been recently applied to a number of migration projects: transition from national currencies to the Euro has been the best excuse of many companies for updating their information systems to new operating systems and environments. Some of the companies still had their programs running on MS-DOS mode and managing a dBase III database. In these cases, the reengineering process started with the migration of the files and indexes to a Microsoft Access or SQL Server database. Then, the software engineer established the foreign key relationships among tables, maybe changed currency-columns values to euros and, then, the new version of the program was generated in one or more selected programming languages: we have always preferred any Java environment and/or servlets for the presentation tier, as they support object-oriented constructions completely and have total compatibility (without any effort) with the business tier also generated in Java. This has allowed the customers to access their data from any remote place via Internet, which is very appreciated.

6. Conclusions and future work

This article has presented a method for generating fully executable applications from a relational schema representing the information structure of a system. The method uses formal descriptions of the different sets used in the transformation process and defines algorithms to implement the translation. This makes the process independent of the target programming language or platform. A translation tool, according to this method, has also been presented.

The reverse-engineering algorithm produces the business class modeling of the relational schema working as if the one class, one table pattern was used during the development stage. We want to extend the method and the tool to consider the possibility of reverse-engineering the relational database according to other patterns and to combinations of patterns. Other improvements that are being currently studied include the processing of *n*-ary relationships and the detection of other possible patterns that were used during the database construction. For both issues, the analysis of actual data instances will be a help, as has been evidenced by Soutou [28] (for example, the detection of a set of records with a certain column set to *null* could mean that the one inheritance tree, one table pattern has been used). Moreover, several additional modules are being programmed to generate the tiers for other environments and programming languages.

For concluding, it is also interesting to study the possibility of reverse-engineer an object or object-relational database, as well as to produce one of these from a legacy relational database. However, our industrial experience for this moment has not required to deal with models, although we will probably put on it our attention in a next future. Fortunately, the independence of both the reverse and the forward engineering stages allows us to devote different resources and efforts to these issues.

References

[1] M. Andersson, in: Loucopoulos (Ed.), Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering, Proceedings of the 13th International Conference on Entity-Relationship Approach, Springer-Verlag, Berlin, LNCS, vol. 881, 1994, pp. 403–419.
[2] R. Arnold, Software Reengineering, IEEE Press, New York, 1992.
[3] T.J. Biggerstaff, B.G. Mitbander, D.E. Webster, Program understanding and the concept assignment problem, Communications of the ACM 37 (5) (1994) 72–83.
[4] L.C. Briand, S. Morasca, V.R. Basili, Property-based software engineering measurement, IEEE Transactions on Software Engineering 22 (1) (1996) 68–86.
[5] J.G. Brookshear, Theory of Computation: Formal Languages, Automata and Complexity, The Benjamin/Cummins Publishing Company, Redwood City, CA, 1989.
[6] M. Broy, Toward a mathematical foundation of software engineering methods, IEEE Transactions on Software Engineering 27 (1) (2001) 42–57.
[7] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, A System of Patterns: Pattern-Oriented Software Architecture, Addison-Wesley, Reading, MA, 1996.
[8] L. Pedro de Jesus, P. Sousa, in: Nesi, Verhoef (Eds.), Selection of Reverse Engineering Methods for Relational Databases, Proceedings of the Third European Conference on Software Maintenance, IEEE Computer Society, Los Alamitos, CA, 1998, pp. 194–197.
[9] J. Biskup, R. Menzel, T. Polle, Transforming an entity-relationship schema into object-oriented database schemas, in: Eder, Kalinichenko (Eds.), Advances in Databases and Information System, Moscow, Workshops in Computing, Springer, Berlin, 1996, pp. 109–136.
[11] M. Castellanos, A Methodology for Semantically Enriching Interoperable Databases, Proceedings of the 11th British National Conference on Databases, 1993, pp. 58–75.
[12] R. Chiang, T. Barron, V.C. Storey, Reverse engineering of relational databases: extraction of an EER model from a relational database, Journal of Data and Knowledge Engineering 12 (2) (1994) 107–142.
[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addisson-Wesley, Reading, MA, 1995.
[14] M. Gogolla, R. Herzig, S. Conrad, G. Denker, N. Vlachantonis, in: V. Elmasri, Kouramajian, B. Thalheim (Eds.), Integrating the ER Approach in an OO Environment, Proceedings of 12th International Conference on Entity-Relationship Approach, Arlington, Texas, USA, 1993, pp. 376–389.
[15] J.L. Hainaut, J. Henrard, J.M. Hick, D. Roland, V. Englebert, Database Design Recovery, Proceedings of the Eighth Conference on Advance Information Systems Engineering, CAiSE'96, Springer, Berlin, 1996, pp. 463–480.
[16] R. Jungclaus, G. Saake, T. Hartmann, C. Sernadas, Object-oriented specification of information systems: the TROLL language, Informatik-Bericht 91 (04) (1991).
[17] K. Lano, J. Bicarregui, S. Goldsack, Formalising Design Patterns, First BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Computer Science, Springer, Berlin, 1996.
[18] C. Larman, Applying UML and Patterns, Prentice-Hall, Upper Saddle River, NJ, 1998.
[19] N. Leavit, Whatever happened to object-oriented databases?, IEEE Computer 33 (8) (2001) 16–19.
[20] M. Malki, A. Flory, M.K. Rahmouni, Extraction of object-oriented

schemas from existing relational databases: a form-driven approach, Informatica 13 (1) (2002) 47–72.
[21] Ó. Pastor, E. Insfrán, V. Pelechano, J. Romero, J. Merseguer, OO-METHOD: an OO Software Production Environment Combining Conventional and Formal Methods, Proceedings of the Ninth Conference on Advanced Information Systems Engineering (CaiSE, 1997), 1997, pp. 145–158.
[22] M. Polo, M. Piattini, F. Ruiz, RCRUD: Reflective Create, Read, Update and Delete, Proceedings of the Sixth European Conference on Pattern Languages of Programs, 2001, Also available at (May 5, 2002); http://www.inf-cr.uclm.es/www/mpolo.
[23] W. Premerlani, M. Blaha, An approach for reverse engineering of relational databases, Communications of the ACM 37 (5) (1994) 42–49.
[24] Rational Software Corporation, Available at (May 5, 2002); http://www.rational.com.
[25] J. Reinder, M.G. Loe, A. Glas, R.L. Krikhaar, T. Winter, Maintaining a legacy: towards support at the architectural level, Journal of Software Maintenance 12 (3) (2000) 143–170.
[26] S.R. Schach, A. Tomer, A maintenance-oriented approach to software construction, Journal of Software Maintenance 12 (1) (2000) 25–45.
[27] P. Shoval, N. Shreiber, Database reverse engineering: from the relational to the binary relationship model, Journal of Data and Knowledge Engineering 10 (1993) 293–315.
[28] C. Soutou, Relational database reverse engineering: algorithms to extract cardinality constraints, Journal of Data and Knowledge Engineering 28 (1998) 161–207.
[29] TogetherSoft Corporation, Available at (May 5, 2002); http://www.togethersoft.com.
[30] J. Warmer, A. Kleppe, The Object Constraint Language, Addison-Wesley, Reading, MA, 1998.
[31] J.W. Yoder, Patterns for Making Business Objects Persistent in a Relational Database, Tutorial at the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA, 2001), Tampa Bay, Florida, USA, 2002, Examples can be found at (May 5, 2002); http://www.joeyoder.com.