JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2007; **19**:383–417
Published online 1 October 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.353

Research

An MDA-based approach for database re-engineering

Macario Polo*,†, Ignacio García-Rodríguez and Mario Piattini

Escuela Superior de Informática, Universidad de Castilla-La Mancha, Paseo de la Universidad, 4, 13071 Ciudad Real, Spain

SUMMARY

This article presents the technical and functional descriptions of a tool specifically designed for database re-engineering. As is well known, re-engineering is the process of (1) applying reverse engineering to a software product to obtain higher-level specifications and (2) using these specifications as the starting point for the development of a new version of the system. Thus, the complete process can be seen as a sequence of transformation functions that operate on the different sets involved in the whole process. The starting point of the re-engineering process is the physical schema of the database which is translated into a vendor-independent metamodel (the logical schema) and then translated into a class diagram representing a possible conceptual schema of the database. This diagram is then taken as the starting point for the code generation process, which produces an executable application for four possible different platforms.

Received 25 January 2007; Revised 27 April 2007; Accepted 11 July 2007

KEY WORDS: code generation; re-engineering; reverse engineering; model-driven reengineering

1. INTRODUCTION

In 1990, Chikofsky and Cross [1] defined some key concepts in software maintenance, such as *re-engineering*, *restructuring*, *reverse engineering* and *forward engineering*. For these authors, *re-engineering* is the 'examination and alteration of a subject system to reconstitute it in a new form

*Correspondence to: Macario Polo, Escuela Superior de Informática, Universidad de Castilla-La Mancha, Paseo de la Universidad, 4, 13071 Ciudad Real, Spain.
†E-mail: macario.polo@uclm.es

Contract/grant sponsor: Junta de Comunidades de Castilla-La Mancha; contract/grant number: PBI-05-058
Contract/grant sponsor: Ministerio de Industria, Turismo y Comercio; contract/grant number: FIT-340000-2005-161
Contract/grant sponsor: Ministerio de Educación y Ciencia; contract/grant number: TIN2006-15175-C05-05

and the subsequent implementation of the new form [...]. It generally includes some form of *reverse engineering* (to achieve a more abstract description) followed by some form of forward engineering or restructuring'. In the same reference, *restructuring* is defined as 'the transformation from one representation level to another at the same relative abstraction level'. Finally, these authors state that the term *forward engineering* is the 'traditional process of moving from high-level abstractions [...] to the physical implementation of a system'. The adjective 'forward' is needed in this context to 'distinguish this process from reverse engineering'.

In many cases the source and target platforms of the re-engineered system coincide, but in other situations re-engineering is used to produce versions of the original system adapted to new environments and paradigms, such as the web, object orientation, distributed computation, component-based software, etc. In the same way, re-engineering is sometimes used for migrating a subset of the source system to a new paradigm, such as the exposition of a couple of the system functionalities as web services.

In the reverse-engineering step, the source code is the main starting point, although it can start from many other elements, such as traces from program executions [2,3], data files [1] or artifacts produced in intermediate stages of software development [4]. When reverse engineering starts from the source code, the typical extracted target domain is one or more class models. Relational databases are also a common source of reverse engineering [5–13]. Most of the works in this line are concerned with the extraction of the conceptual schema of the database; thus, the target product of the reverse engineering is often an entity-relationship (ER) or extended entity-relationship (EER) diagram.

An ER or EER diagram is useful when the final software product will be a new version of the database (for the same or a different database manager). However, as databases are usually managed by external programs, obtaining the conceptual schema in another format (such as a UML class diagram) would help the construction of both the new version of the database and the program or set of programs in charge of managing it.

When a new management information system is being built using the object-oriented paradigm, the class model corresponding to the domain/business tier of the application is often used as the conceptual schema for constructing the database (in most cases, a relational database [14]), thus bringing the world of objects to the world of tables. In general, there are at least three ways to translate a set of classes into a set of tables [15]: (1) obtaining a table from each class in the diagram, translating associations, aggregations and inheritance relationships into foreign key constraints (this is known as the 'one class, one table' transformation pattern); (2) building, a table for each inheritance path in the class model ('one inheritance path, one table'); (3) translating, a whole inheritance tree into a single table ('one inheritance tree, one table'). Each possibility has some advantages and drawbacks; thus, it is common to apply different combinations of these methods to the same class diagram.

In a three-layer system, the domain class diagram is also the basis for the creation of the rest of the layers in a multilayer application, such as the presentation or the persistence layer. The presentation layer contains the windows, forms and screens that the user uses to interact with the application: the presentation layer receives messages from the user and sends them to the adequate class in the business layer, which may require executing some operation with the database via the classes in the persistence layer. Obviously, other layers or sublayers may be required depending on the type of application.

DOI: 10.1002/smr

This article presents a method and a tool (called *Relational Web*) that integrates a complete process of re-engineering, including the reverse-engineering, restructuring and forward-engineering stages. The starting point is a relational database, whose physical schema is reverse engineered into a class diagram representing its conceptual schema (following the 'one class, one table' pattern). In restructuring, the class diagram is manipulated by the user and then passed as input to an automated code generation process, which builds a multilayer system using different programming languages and platforms. Moreover, the tool also includes the possibility of migrating the database from one database manager to another. The architectural design of the tool makes it possible to add new code generators for other programming languages easily.

The paper is organized as follows: Section 2 discusses some related works. Section 3 presents the re-engineering method and the tool implemented, describing each step in a different subsection. Section 4 analyses the correspondence of the proposed re-engineering process with the model-driven architecture (MDA). Section 5 describes some case studies where the tool has been applied. Finally, conclusions and future lines of work are presented in Section 6.

2. RELATED WORK

Reverse engineering of databases has been widely studied as an essential part of re-engineering, probably because databases save all the relevant information from companies and are one of the business elements most susceptible to becoming out-of-date and degraded due to maintenance tasks which, in turn, have a strong influence on the programs in charge of managing them.

Until a few years ago, most research related to database re-engineering dealt with the reverse-engineering stage. In 1996, Hainaut *et al.* [8] described the main steps of database reverse engineering as the opposite process of its construction (Figure 1): if the forward engineering of a database requires Conceptual Design, Logical Design, Physical Design and View Design, its reverse engineering needs Data Structure Extraction and Data Structure Conceptualization. Data Structure Extraction produces a complete description of the data structures according to the model of the DMS, whereas Data Structure Conceptualization 'tries to make the semantics of the logical schema explicit by recovering the intention of the optimized DMS data structures'. According to Hainaut *et al.*, Data Structure Extraction 'appears as the inverse of the Physical Design forward process', whereas Data Structure Conceptualization is 'to a large extent the reverse of the Logical Design forward process'.

2.1. Traditional approaches for database reverse engineering

Taking the Hainaut *et al.* work as a suitable description for the database reverse-engineering process, several authors have made concrete proposals to recover conceptual schemas from databases. Some relevant works are the following:

1. Andersson [6] proposes a method to recover ERC+ specifications (an extension of the ER model) from relational databases using rudimentary information that is expected to be found in the legacy database (names of tables and fields, indexes and view definitions). These data are looked for in the data manipulation statements extracted from the application code, written in a 4GL environment.

DOI: 10.1002/smr

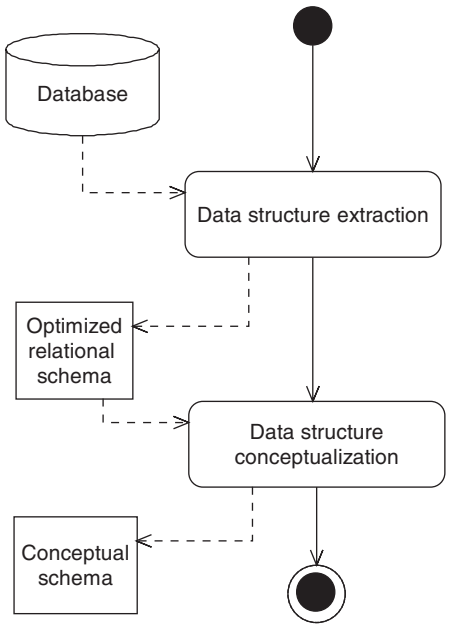


Figure 1. Database reverse engineering (adapted from Hainaut *et al.* [8]).

2. Shoval and Shreiber [16] obtain a binary relationship diagram from a relational database. The data input step is not automated (the information on the relational schema must be introduced manually) and the binary relationship model is not used for conceptual data modelling.
3 Chiang *et al.* [7] describe an algorithm to obtain an EER schema from a relational database. 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 are inferred by the algorithm but finally determined by the user.
4. In the context of Federated Database Systems, Castellanos [17] obtains object-oriented specifications of the relational databases forming the system, using the data on the database and the corresponding data dictionary. 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 to increase knowledge about the individual databases of the system.
5. Tari *et al.* [18] also apply a set of algorithms to several federated databases to obtain a single set of C++ source files representing the classes that make up the object-oriented conceptual schema.
6. Alhajj and Polat [5] present an approach to transform a relational database into an object-oriented database, based on the data dictionary and on expert knowledge. The authors also deal with the migration of data to the new environment.
7. On the basis of only the data dictionary, Soon *et al.* [19] propose a detailed set of steps to obtain an object-oriented database conceptual schema from a relational one.

DOI: 10.1002/smr

8. Premerlani and Blaha [20] present some conclusions about their work on several case studies. The authors use three sources of information (the schema, observed patterns of data and the semantic understanding of the application) to obtain an OMT representation of the database. They summarize the process in five steps, beginning with the consideration of a class for each table and finishing with some transformations to improve time and space performance.
9. Pérez *et al.* [21] produce an OASIS object-oriented conceptual schema from the data dictionary of a relational database. The work deals with the migration of the data from an old to a new relational database, although they obtain an intermediate object-oriented schema.
10. Yeh and Li [22] obtain the ER diagram from a legacy dBase III system. They import the old data files into tables of an SQL Server database and base the reverse engineer on the manual analysis of the data and the table structure.

Thus, strategies for database reverse engineering can be classified in several ways. Table I gives the aforementioned references, classified according to (1) the procedure for the data input and (2) the source and target systems.

Most of the works analysed here share the principal goal of obtaining a new model representing the original database (sometimes in object-oriented notation, other times in ER diagrams, and even new relational databases), and at that point they finish the process.

An interesting approach is to take the conceptual diagram obtained as a UML model, and to use it as the starting point for a new automated forward-engineering stage, making it possible to obtain not only a conceptual diagram but also a semi-automated construction of a set of applications to

Table I. Classification of some works on database reverse engineering.

Reference	Source system	Data input	Target system
Alhajj and Polat [5], 2001	Relational database	Data dictionary, expert knowledge	Object-oriented database
Andersson [6], 1994	Relational database	Analysis of DML embedded in program code	ERC+
Castellanos [17], 1993	Federated relational databases	Data dictionary, data	BLOOM specification
Chiang *et al.* [7], 1994	Relational database	Data dictionary, data, expert knowledge	Extended ER diagram
Pérez *et al.* [21], 2002	Relational database	Data dictionary	Relational database (intermediate OASIS schema)
Premerlani and Blaha [20], 1994	Relational database	Data dictionary, data, expert knowledge	OMT class diagram
Shoval and Shreiber [16], 1993	Relational database	Manual	Binary relationship diagram
Soon *et al.* [19], 2004	Relational database	Non-specified	Object-oriented conceptual schema (textual)
Tari *et al.* [18], 1997	Federated relational databases	Data dictionary	C++ source files
Yeh and Li [22], 2005	dBase III files	Importation to SQL Server	ER diagram

DOI: 10.1002/smr

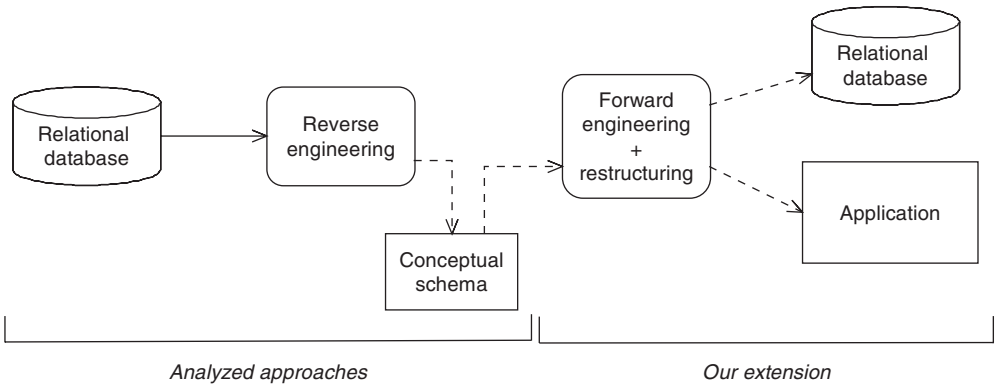


Figure 2. Roadmap of the re-engineering process described in this article.

manage the database, as well as, most likely, a new version of the database. This article explores how the traditional approaches analysed above can be extended by adding a further stage of forward engineering (Figure 2).

2.2. Object–relational mapping

The difference between the object and the relational worlds has led to the development of many techniques and tools to map classes with tables. Larman [23] offers an excellent overview of persistence frameworks to preserve these mappings. This author lists some of the key aspects covered by a persistence framework:

1. Preserving the mapping between tables and classes.
2. Object identity.
3. Control of the materialization and unmaterialization of objects.
4. Use of object caches.
5. Control of transactions.
6. Lazy materialization.

On a practical level, there are several common implementations of object–relational mapping techniques.

In order to present the characteristics of the object–relational mappings, the example in Figure 3 is used. This shows the possible structure of the database used by an editorial to manage the submissions of articles to its journals. The database saves information about authors, reviewers, articles, journals and reports.

Torque and *Object–relational bridge* are two open-source projects of the Apache Software Foundation (http://www.apache.org). From an XML representation of the database schema, they generate a set of classes with persistence capabilities. In general, these projects follow the 'one class, one table pattern' [24], although the generated structure of classes is prepared to preserve the easy maintenance of classes.

DOI: 10.1002/smr

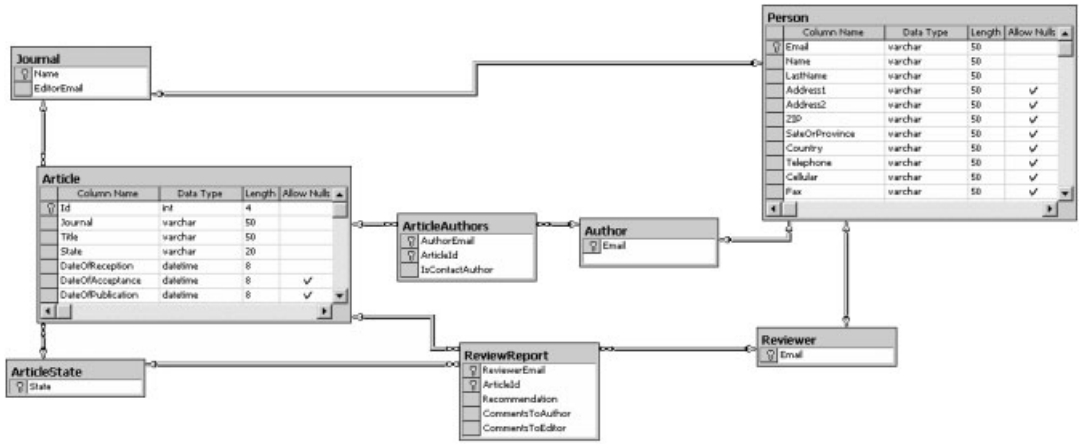


Figure 3. A sample database.

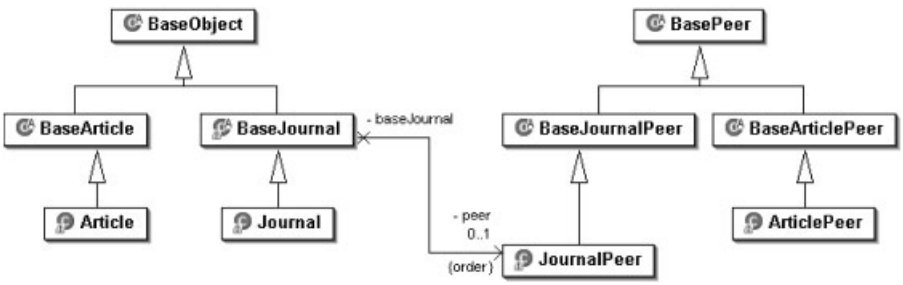
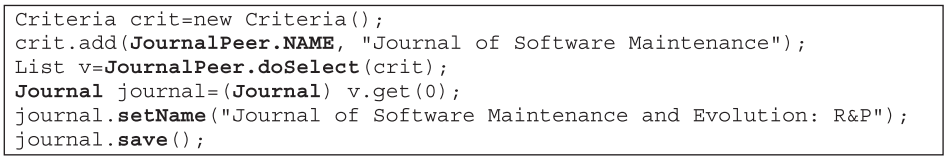


Figure 4. Some reverse-engineered classes using Torque.

Figure 4 shows some of the classes obtained by applying Torque to the database in Figure 3 (the structure for Object–Relational Bridge is very similar). For each table *X*, the following classes are built: *BaseXPeer*, *XPeer*, *BaseX* and *X*. *BaseX* (*BaseJournal*, for example) is an abstract class that keeps the mapping between the class and the table (it defines a field for each column in the database, methods to set and to get values from the fields and some auxiliary methods); thus, the programmer must use its specialization, *X*(*Journal*, for example), to manipulate records from the table, adding business methods, etc. Peer classes (*BaseJournalPeer* and *JournalPeer*) are implementations of the 'Pure Fabrication' pattern of Larman [23]. *BaseJournalPeer* contains static implementations of the persistence methods for the corresponding table; thus, non-peer instances represent records and they delegate the execution of their corresponding persistence instructions to the associated peer classes.

Sentences in Figure 5 would materialize an instance corresponding to the '*Journal of Software Maintenance*' from the database and would change the record value. Note that the construction of

DOI: 10.1002/smr



```
Criteria crit=new Criteria();
crit.add(JournalPeer.NAME, "Journal of Software Maintenance");
List v=JournalPeer.doSelect(crit);
Journal journal=(Journal) v.get(0);
journal.setName("Journal of Software Maintenance and Evolution: R&P");
journal.save();
```

Figure 5. Code to materialize an instance from a record.

the instance is made through the *JournalPeer* class, but records are recovered as *Journal* instances. In the same way, the call to the *save* method shown in the last line finishes in a call to the *doUpdate* method of *JournalPeer*.

Hibernate (http://www.hibernate.org/) is also an open-source project. It supports different mappings between tables and classes and lazy materialization (objects are not created until they are required). Hibernate makes it possible to write queries in SQL and HQL (a specific query language). It is integrated with J2EE and can be used as persistence manager in EJB 3 environments.

With *RCRUD* Polo *et al.* [25] propose a reflective persistence framework, which supports different mapping techniques (one class, one table; one inheritance tree, one table; one inheritance path, one table). The idea in this work is to take advantage of the reflective characteristics of several programming languages (Java or those in Microsoft .NET) to generate at runtime the set of persistence methods of classes. The authors propose using an abstract class, *RCRUD* ('Reflective CRUD'), which includes a set of concrete operations. These access the database structure in runtime and generate the corresponding persistence instruction. Thus, when an instance must be inserted in the database, it calls its corresponding *insert* method, which is completely defined in RCRUD. RCRUD is the superclass of the whole hierarchy of persistent classes. Subclasses of RCRUD do not include the implementation of persistence instructions, but only of business methods, which must be added manually by the programmer.

In general, all the reviewed approaches tend to leave persistence methods in a distinguished set of classes, whereas the programmer must add business methods in a separated set. This type of design increases the cohesion of classes and obtains a good level of coupling, which are two desirable characteristics of object-oriented systems [23].

Several commercial and open-source CASE tools that reverse engineer relational databases into classes according to these approaches exist. In this respect, research in the field of the reverse engineering of relational databases has reached maturity.

2.3. Model transformation

According to OMG [26], MDA is 'an approach to system development which increases the power of models in that work [system development] because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification'.

The ideal goal of MDA is the construction of software using only graphical representations of the system. For this, MDA suggests the use of three viewpoints to represent software systems (computation-independent, platform-independent and platform-specific viewpoints), establishing that the development of software systems can be seen as a series of successive transformations

DOI: 10.1002/smr

from one viewpoint to another (i.e., from one viewpoint model to another). Different types of metamodels can be mapped to each viewpoint. It is possible, then, to have the relational (for representing relational databases) or the object-oriented metamodel and to define a set of transformation rules to transform one metamodel instance into another. For example, an instance of the object-oriented metamodel can be transformed into an instance of the relational one, or vice versa. Thus, a metamodel is simply a model of models [27].

In model-driven re-engineering, the MDA approach is applied to the re-engineering of legacy systems. Here, one or more models (actually, instances of metamodels) are obtained from the legacy system and are translated via a set of transformations into new models, each time closer to the target system.

Following the model-driven approaches, MOMENT [28] is a framework for 'MOdel manageMENT'. In MOMENT, metamodels are specified as sets of elements and transformations are specified by means of algebra in the Maude rewriting term system [29]. A MOMENT metamodel is an abstract representation for some kind of metadata. MOMENT is an implementation of the technique in Pérez *et al.* [21], which was analysed in Section 2.1.

There are other languages closely related to models, metamodels and transformations, such as ATL. ATL (ATLAS Transformation Language) is a metamodel-based transformation DSL (Domain-specific Language) intended to be compliant with the OMG/QVT recommendation and designed to express model transformation as required with any MDA approach [30,31].

In general, current research tends to distinguish between two different strategies in model transformation:

(a) Transformations based on rewriting rules (such as MOMENT), which require user intervention to guide the translation. These strategies make it possible to describe the final system before or during the transformation, although this task requires coding the complete set of rewriting rules, which can be tedious and prone to error.
(b) Transformations based on marks and templates, which completely automate the translation process. The main drawback of these strategies is that modifications cannot be done until the final system has been obtained.

The technique presented in this article, and its corresponding tool, is an intermediate proposal between both approaches: the main goal is the generation of an object-oriented application from a relational database. For this, the database structure is extracted and represented using a relational metamodel, which is later transformed into a class diagram. This can be directly used to generate the final application. However, before generating the final application, the software engineer can modify the default behaviour of classes with state machines, which implies the modification of the finally obtained system.

3. DESCRIPTION OF THE RE-ENGINEERING PROCESS

This section describes the complete re-engineering process implemented in the tool, including algorithms and metamodels. The first subsection provides a general overview of the main step of the process and justifies why it is considered a re-engineering process. Each step is then explained in detail in further subsections.

DOI: 10.1002/smr

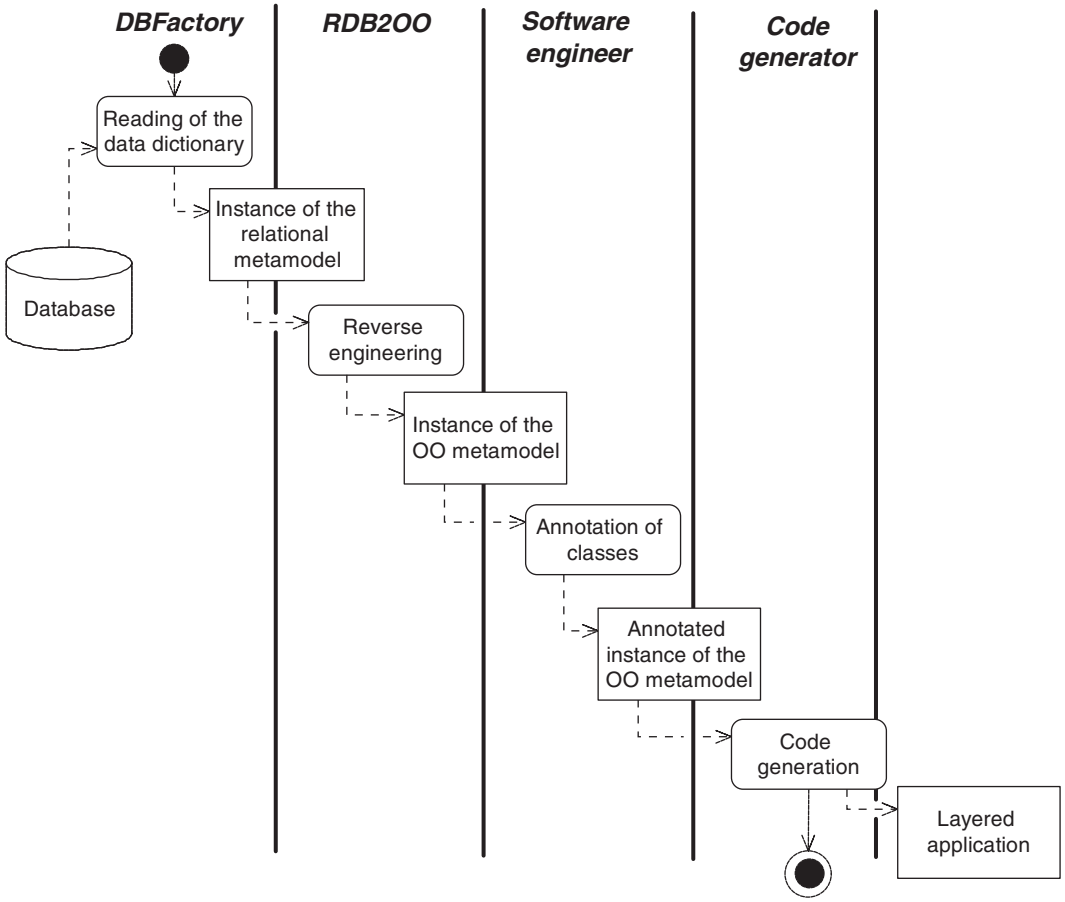


Figure 6. General view of the re-engineering process.

3.1. General description of the process

Figure 6 describes, using a UML activity diagram, a general view of the re-engineering process which is also implemented in the tool. As seen here, four principal entities intervene in the process: a *DBFactory* is in charge of reading the database data dictionary and producing an instance of the relational metamodel; *RDB2OO* applies reverse engineering to obtain a class diagram (i.e., an instance of the object-oriented metamodel) representing the conceptual schema corresponding to the relational model; then, the S*oftware Engineer* may modify, still at a conceptual level, the default behaviour of the classes obtained with the addition of state machines; finally, a *Code Generator* (with different specializations) generates a multi-layer application for the desired target.

At the end of the intermediate stages, the *Software Engineer* should check and validate the different models obtained. As the architecture of the final application has a complete dependence

DOI: 10.1002/smr

Table II. General characteristics of the code generators.

Code generator	Presentation tier	Domain tier
OO2EJB	JSP pages	Enterprise Java Beans
OO2JSP	JSP pages	Standard Java classes
OO2JavaDesktop	JFrames, JPanels, JDialogs (from javax.swing)	(these two code generators share the domain classes)
OO2CSharpDesktop	Windows Forms	C# classes

on the class diagram, which in turns proceeds from the original database schema, it is central to perform such a validation to ensure that all models are representing the same problem.

Here, the process described is a complete re-engineering process in the sense of Chikofsky and Cross [1] mentioned in Section 1, since it includes reverse engineering (the class diagram produced from the database corresponds to the conceptual model of the database, thus being 'a more abstract description' of this), restructuring (the behaviour of the classes, obtained automatically, can be modified at the same abstraction level) and forward engineering (the class diagram, which represents the conceptual model, is used to generate the physical implementation of the system).

The *Code generator* component shown in the last swimlane of Figure 1 has four specializations, one for each type of target platform (Table II). Thus, the *OO2EJB* code generator builds an EJB-based application with JSP pages in its presentation layer; the *OO2JSP* and the *OO2JavaDesktop* generators share the component in charge of building the domain layer of the system, although they have different generators for the presentation layer; finally, *OO2CSharpDesktop* builds an application using C# classes in the domain layer and Windows Forms in the presentation one.

3.2. Obtaining the database instance

DBFactory is the class in charge of reading the database data dictionary to get a vendor-independent representation of the physical database (that is, to get the instance of the relational metamodel shown in Figure 6). It is implemented as an abstract factory pattern [32], with as many specializations as types of databases that must be processed. Today, the tool can manipulate databases implemented in Oracle, Caché (from Intersystems), SQL Server and Access (from Microsoft), meaning that four concrete factories exist (Figure 7).

Each concrete factory accesses the database data dictionary through the *Connection* object it knows. *Connection* is a standard interface provided by Sun Microsystems that is implemented by most database vendors. It includes the *getMetaData* operation, which returns a *DatabaseMetaData* object and, from this, Relational Web accesses and extracts the information saved in the data dictionary (Table III).

Figure 7 shows the hierarchy of factories and illustrates the case of the *SQLServerFactory* class, which instantiates the connection to a *SQLServerConnection* object, a class provided by Microsoft for connecting Java applications to SQL Server databases. *SQLServerConnection* implements all the operations required to access the database metadata and therefore, is useful when recovering the instance of the relational metamodel. Oracle and Caché also offer the corresponding classes; thus, the microarchitecture for these databases is similar to that in Figure 7. The case of Microsoft

DOI: 10.1002/smr

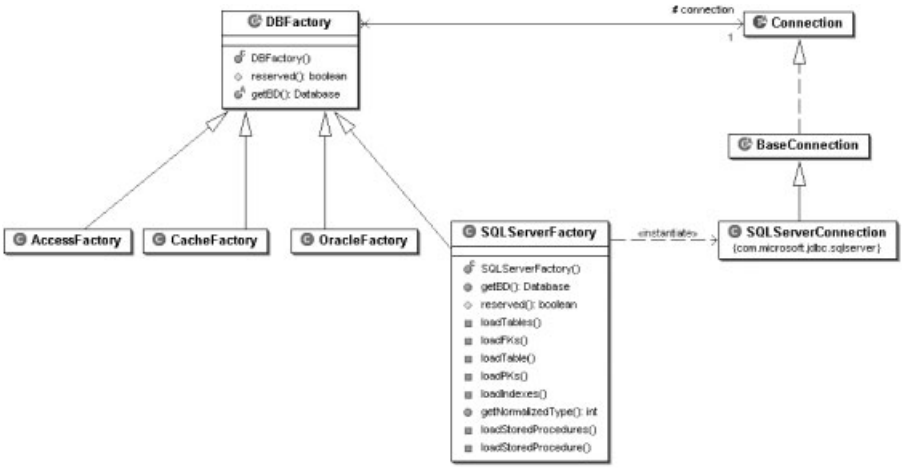


Figure 7. Factories to recover the database structure.

Table III. Some operations included in *java.sql.DatabaseMetaData*.

getCatalogs()
getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)
getExportedKeys(String catalog, String schema, String table)
getImportedKeys(String catalog, String schema, String table)
...

Access databases is different, since it must be used through the standard JDBC–ODBC bridge, which does not provide results for all operations in the Connection interface. In this case, the data dictionary must be accessed directly by reading from a set of system tables contained in the self-data file. Thus, the existence of a separated *AccessFactory* for reading Microsoft Access databases is completely justified. The three remaining factories can be grouped into a single one (as some tools do); however, the JDBC connection string is different for each vendor and there are also differences in the implementations given to their Connection classes and in the variety of data types provided (which has a strong influence on further code generation steps). Due to this, the implementation of the system in charge of reading the data dictionary as an abstract factory was chosen.

After reading the data dictionary, each *DBFactory* builds an instance of the relational metamodel, which is represented in the tool as the *Database* class. *Database* represents the relevant structures of a relational database from the point of view of this re-engineering process. Here then, *Database* saves information about tables and their columns, foreign keys and stored procedures, although it does not keep information about triggers or check constraints. Figure 8 shows the relationships of this class with the others.

The translation from the physical to the logical schema (i.e., the instance of *Database*) is the main function of *getBD*, an abstract operation included in the *DBFactory* class, which is redefined in each specialization. Excepting in the *AccessFactory*, the implementation of this operation uses

DOI: 10.1002/smr

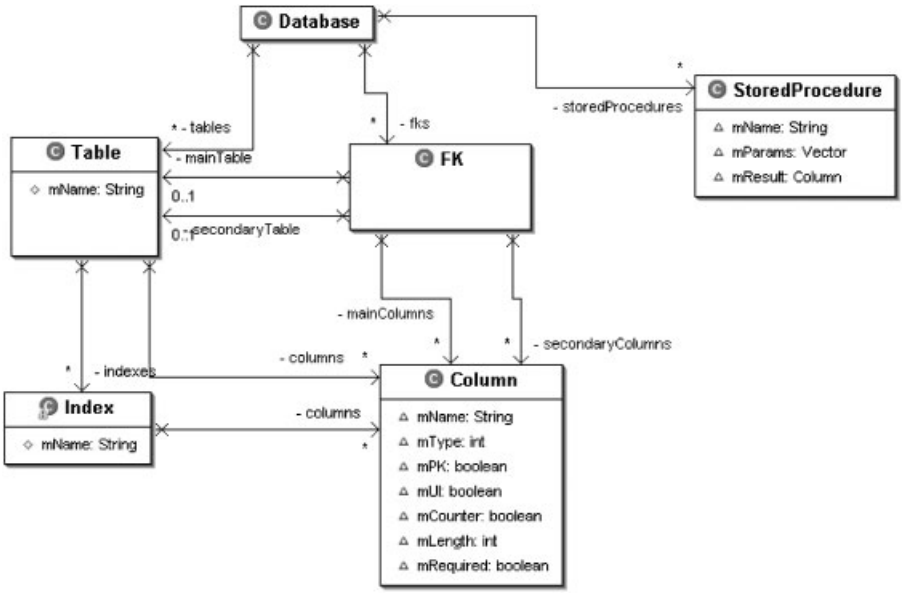


Figure 8. Structure of *Database*, which represents the relational metamodel.

```
- <database name="Journal" manager="rw3.domain.metamodels.dbms.SQLServerDB" user="sa"
      password="test" serverIP="localhost" port="1433">
   - <table name="Article">
       <column name="Id" type="4" isPK="true" isUI="false" isCounter="true" length="11"
           isRequired="false" />
       <column name="Journal" type="12" isPK="false" isUI="false" isCounter="false" length="50"
           isRequired="false" />
           ...
     </table>
   + <table name="Journal">
       ...
   - <fk name="FK_Article_Journal" mainTable="Journal" secTable="Article" mainCardinality="1"
        secCardinality="*">
       <columns main="Name" sec="Journal" />
       </fk>
       ...
 </database>
```

Figure 9. XML fragment of a *Database* instance.

the *DatabaseMetaData* and *ResultsetMetaData* classes (defined in the *java.sql* standard package) to read the data dictionary. Any instantiated metamodel can be saved into XML for later recovery. Figure 9 includes a fragment of the XML representation of the database shown in Figure 3.

DOI: 10.1002/smr

3.3. Reverse-engineering stage

The reverse-engineering stage is carried out by the *RDB2OO* element shown in the second swimlane of Figure 6. It takes an instance of the aforedescribed *Database* and translates it into an object-oriented class diagram representing the possible conceptual model used during the development of the database. Initially, the algorithm for obtaining the class model builds a class for each table in the database (following the 'one table, one class' transformation pattern, one of the three choices mentioned in Section 1, to obtain an instance of an Object-oriented metamodel corresponding to the relational schema), adding to the class a field for each column in the corresponding table of a compatible data type. Then, it processes the foreign key relationships in the database to represent associations and inheritance, which may require removing fields. Let A, B be two tables related by a foreign key from A (referenced table) to B (referencing table):

(1) If all the foreign key columns in B are also the complete primary key (that is, there is a 1:1 relationship connecting B primary key with A primary key), then the foreign key is translated into an inheritance relationship, where the class proceeding from A is the superclass and the class proceeding from B is the subclass. In this case, the fields proceeding from the foreign key columns are removed from the subclass. This translation agrees with Premerlani and Blaha, who say that 'in order to determine generalizations, we generated a list of possible one-to-zero-one associations through queries involving every pair of primary keys' [20, p. 46].
(2) If foreign key columns in B are not its primary key, then the relationship is translated into an association, whose cardinality is determined by the cardinality of the foreign key. Once more, Premerlani and Blaha [20] state that 'the foreign key is usually buried on one end of the association [...]. Unique indexes can enforce multiplicity constraints, particularly for qualified associations'. In fact:

    (a) If all the foreign key columns in B constitute a unique index, then the relationship is 1:1 and will be translated into a 1:1 association. In the class corresponding to A, the fields corresponding to the foreign key columns are removed and A receives a new field of the B data type.
    (b) Otherwise, the foreign key is translated into a B-collection field that is added to the class corresponding to A. In B, a field of type A is added and the fields proceeding from these columns are removed from both classes. Thus, a 1:n foreign key is translated into two associations: 1:0..n and 1:1.

The database in Figure 3 contains eight tables and has several foreign relationships among them:

(1) *Author* and *Reviewer* are related to *Person* by means of their respective primary keys. According to the proposed algorithm, these relations will be interpreted as inheritance relationships: *Person* will be translated into the root class of the hierarchy, whereas *Author* and *Reviewer* will be subclasses.
(2) *Journal* is connected to *Person* via its *EditorEmail* column, which is a unique index, but not its primary key, to guarantee that each person does not edit more than one journal. Thus, the foreign key between *Journal* and *Person* has cardinality 1:1 in the database and will be translated into a 1:1 association. This implies the addition of a *Person* field to *Journal* and

DOI: 10.1002/smr

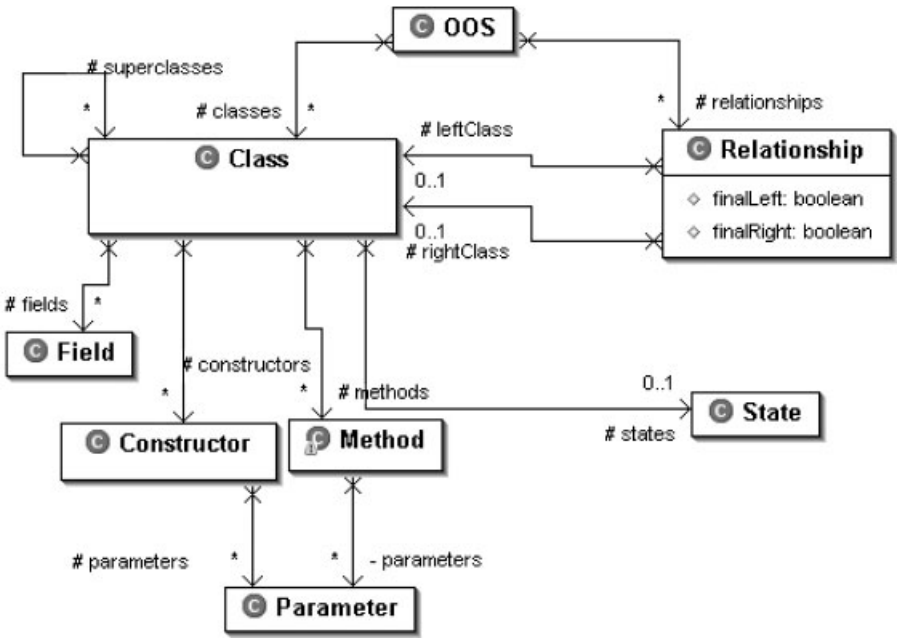


Figure 10. *OOS* metamodel (the structure of *State* is presented in Section 3.4).

the removal of the *EditorEmail* field in the *Journal* class, which was added in the first step of the algorithm.

(3) Since each journal receives many articles (foreign key between *Journal* and *Article*), a field to keep a collection of *Article* instances is added to *Journal*; in *Article*, the *Journal* column (a String field added in the first step) is substituted by a field of type *Journal*.

Figure 10 shows the metamodel used to represent *OOS* instances: for each class, the metamodel saves its set of fields, constructors, methods, superclasses and states (these are presented in Section 3.4).

When the tool is used later to generate the application, its source code can be annotated with comments (also called 'doclets') which can be interpreted by the Oracle JDeveloper development environment or by the Eclipse UML plugin. Figure 11 shows the representation made by Oracle JDeveloper of the domain layer of the application, obtained after having applied the reverse-engineering algorithm and having generated the basic code (that is, with no restructuring). For reasons of space, only partial details from three classes are shown. The following comments illustrate some issues in Figure 11:

1. The foreign key between *Person* (as referenced table) and *Journal* (which was relating the *Person*'s primary key to a unique index in *Journal*) is translated into a navigable 1:1 association from *Journal* to *Person*. Moreover, the field of *Journal* corresponding to the *EditorEmail* column in the *Journal* table has been removed, since this value is accessible through the *mPerson* role and the *getEmailFromPerson* method.

DOI: 10.1002/smr

2. 1:n Foreign keys are translated into two associations. For example, the foreign key between *Journal* and *Article* is translated into an association from one *Journal* to many instances of *Article* (note that the set of articles of the journal is represented by the 1:n association, named *mArticle*), and from each *Article* to its respective *Journal.* Note also that *Article*, which corresponds to the referencing table in this foreign key, has no field corresponding to the *Name* column (its foreign key in the database in Figure 3), since the whole object is now accessible through the *mJournal* role of the 1:1 association.
3. The *ArticleAuthors* table is actually representing an n:m association between *Article* and *Author* (since each article may be written by a set of authors, and each author may write several articles). In this example, *ArticleAuthors* has two foreign keys pointing to the other two tables and one additional column (*IsContactAuthor*). In the object-oriented system, this is represented as an associative class, which can be refactored as shown in Figure 11 [33]. If the table had no additional columns, then it would be replaced by two 1:n associations, one from *Author* to *Article* and another from *Article* to *Author*, which is semantically equivalent to one n:m association.
4. *Author* and *Reviewer* have been stated as specializations of *Person*, since the first two were related to the latest via their primary keys. *Author* and *Reviewer* have removed the fields corresponding to their respective primary keys (*Email* in both cases), because they are inheriting from *Person.*
5. All classes have an *oldPKs* field. When the instance is changed, this field saves (as strings) the previous values of the primary key columns in order to compose the *where* clause of the corresponding *update* SQL instruction.
6. By default, every class receives a basic set of operations: an 'empty constructor', with no parameters, used to build instances of the class with all the fields assigned to default values; a 'materializer' constructor, used to build instances of the class from the information saved in the database; the *insert*, *delete* and *update* methods, which work with the database to insert, delete and update instances; accessor methods (*getX*/*setX*) for each field (the signature of the method may change depending on where the desired field actually stays, such as the aforementioned *getEmailFromPerson* operation in *Journal*). Note that persistence operations receive a *Broker* as parameter: the *Broker* maintains the connection to the target database and plays the role of a Database Broker [34].

The algorithm in charge of making the translation from *Database* to *OOS* is shown in Figure 12, where the construction of classes from tables is made in lines 3–9. Classes receive the same names as the tables they proceed from. For fields, names are mapped to those of their respective columns (primary keys receive the prefix *mPK*, the remaining fields, *m*), and their types are assigned depending on the corresponding column data type. Once the initial set of classes exists, foreign keys are processed to arrange the inheritance relationships and associations. The addition of constructors and methods to classes is not made at this step but is postponed to the code generation step.

Figure 13 shows the main screen of the tool after having reverse engineered the database from Figure 3. It includes one panel for each database vendor, each connected to the corresponding concrete factory in Figure 7. This window shows some of the database metrics proposed by Calero *et al.* [35], which can be used to estimate some code metrics in the final application. In the figure, the tool already has the instance of *OOS* and the user can begin the restructuring.

DOI: 10.1002/smr

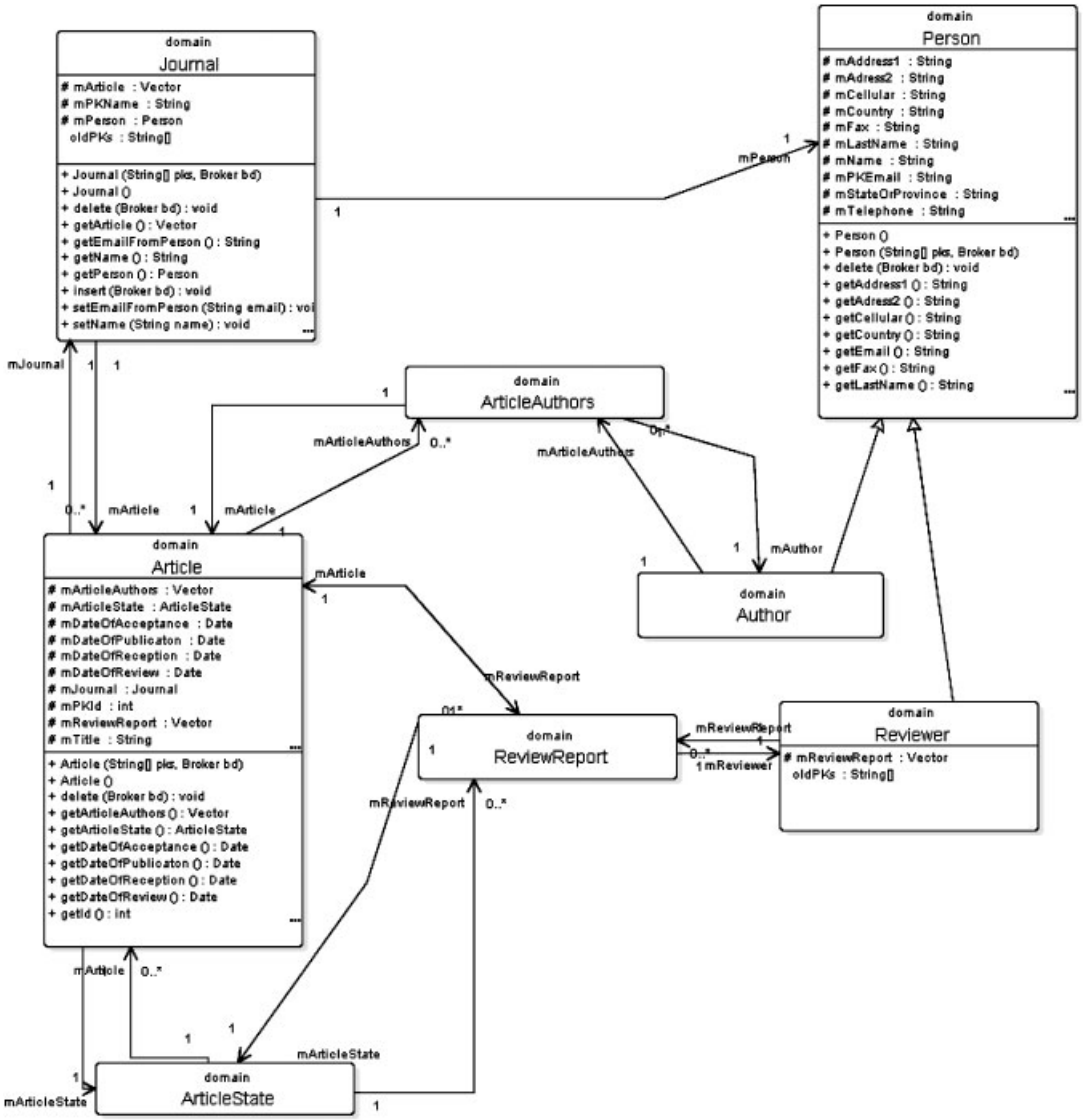


Figure 11. Class diagram proceeding from the database schema in Figure 3.

3.4. Restructuring

In the restructuring step, the software engineer completes class descriptions with state machines that modify the default behaviour (CRUD, get/set operations) of instances, which is determined by the default set of operations assigned to each class.

DOI: 10.1002/smr

```
1   getOOS(db : Database) : OOS {
2     result : OOS =(∅, ∅)
3     ∀ t∈ db.tables {
4       k : Class = (t.name, ∅, ∅, ∅, ∅, ∅)
5       ∀ c∈ t.columns {
6         k.fields = k.fields ∪ { (c.name, c.isIdentifier, c.isRequired, c.type) }
7       }
8       result.Classes = result.Classes ∪ { k }
9     }
10    ∀ fk∈ db.FKs {
11      r : Relationships = (∅, ∅)
12      if (fk.allMainColumnsAreThePrimaryKey() and fk.allSecondaryColumnsAreThePrimaryKey()) {
13        superclass = findClass(result, fk.secTable)
14        subclass = findClass(result, fk.secTable)
15        subclass.addSuperclass(superclass)
16      } else {
17        mainClass=findClass(result, fk.mainTable)
18        secondaryClass=findClass(result, fk.secTable)
19        r=(mainClass, secondaryClass, false, true)
20        arrangeFields(mainClass, secondaryClass, fk)
21        result.Relationships = result.Relationships ∪ { r }
22      }
23    }
24    getOOS = result
25  }
```

Figure 12. Reverse-engineering algorithm.

For each domain class, the user can define one or more state machines, which will be composed by states and transitions. Since the state of an instance depends on the values of its respective fields, the software engineer (see Figure 6) describes every state with a Boolean expression, which will be used to determine whether the instance is or is not in that state. Moreover, states are also annotated with *triggering events*, *entry* and *exit actions*. A triggering event (merely an *Event*) is mapped to a method which may cause the instance to change from a *source* to a *target* state. *Events* can also be annotated with actions, preconditions and post-evaluated conditions. A *Precondition* is a condition that must be fulfilled in order to trigger the execution of the *Event;* a *Post-evaluated condition* is checked after the execution of the operation and may vary the target state. Finally, an *Act ion* is an operation that is executed as a consequence of the event trigger, the input or the output of a state.

Thus, an *Event* will be able to be triggered if the instance is in a state which admits the event and the event precondition is true. Both the expression defining the state and the precondition therefore play the role of a single precondition. Figure 14 summarizes the set of classes implementing the State metamodel.

In order to illustrate the restructuring step, suppose that the desired behaviour for the *Article* class (depicted in Figure 11) is that shown in Figure 15: initially, an article is *Presented* and, when it has three reviewers assigned (note the post-evaluated conditions that use the *getNumberOfReviewers* operation), passes to the *UnderRevision* state; in *UnderRevision*, it may execute the *review* operation, which causes the instance to remain in the same state when the call to *getNumberOfRevisions* (a precondition) is less than 2; otherwise, the instance goes to *Reviewed*, which is left when the *accept*

DOI: 10.1002/smr

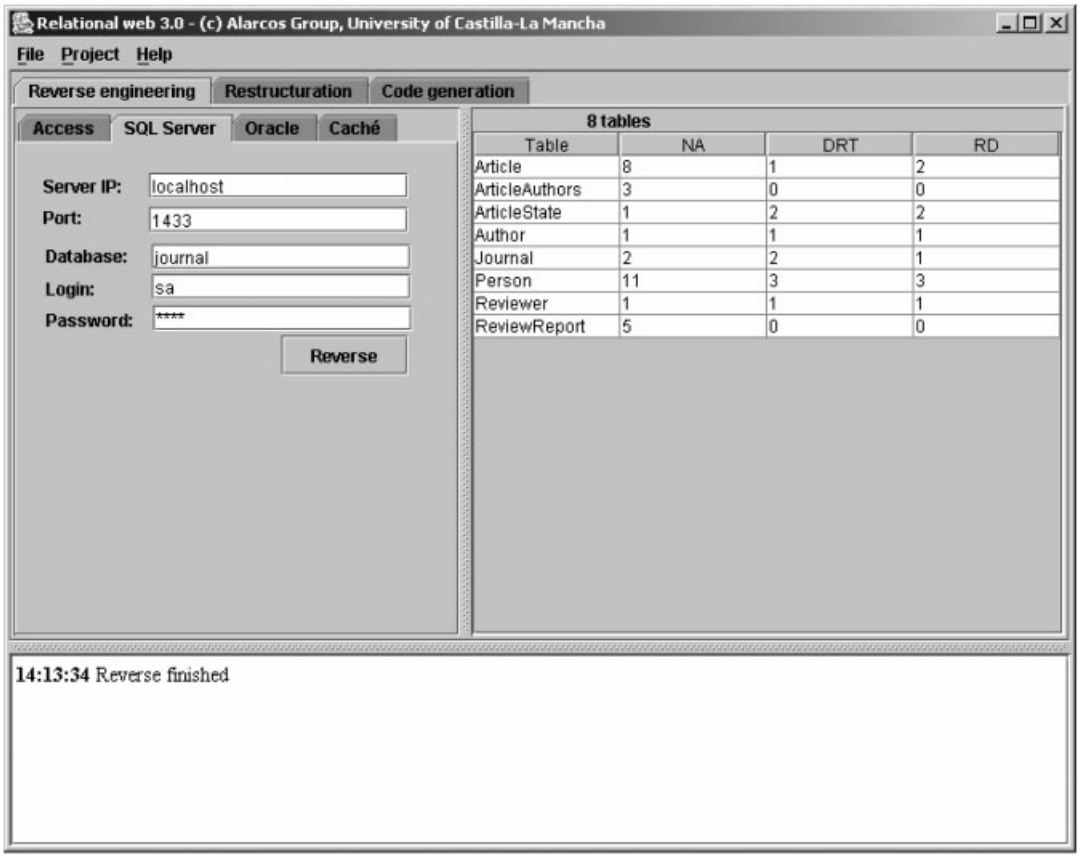


Figure 13. Reverse-engineering panel of the tool.

operation is called. Finally, and depending on several post-evaluated conditions, the article goes to one of the remaining states.

All these states, as well as the operations that have been mentioned in the previous paragraph (*getNumberOfRevisions*, *accept*, etc.), must be adequately described in the tool.

Figure 16 shows the restructuring panel of the tool applied to the *Article* class. The left side lists all the classes obtained after having reverse engineered the database (note that *Article* is selected). The middle column includes the states defined for the selected class. The right side provides the complete description of the state.

As noted above, all states must have a name and a description, which is a Boolean expression representing when the instance is in that state. Furthermore, each state may contain entry and exit actions, triggering events and transitions.

In Figure 16, the software engineer is describing the state *Presented*. Note that the annotation determines that the article is in this state when the date of review is null and three reviewers have not yet been assigned. The date of review corresponds to the *DateOfReview* field, which can be

DOI: 10.1002/smr

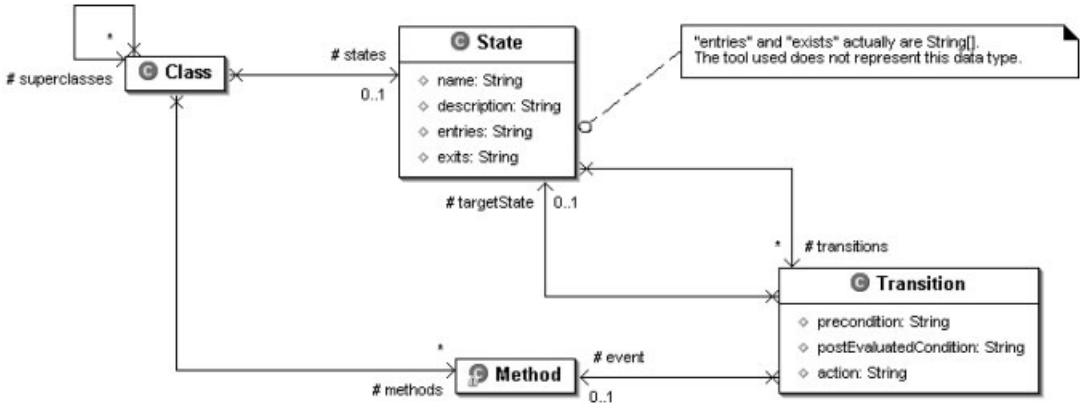


Figure 14. State metamodel.

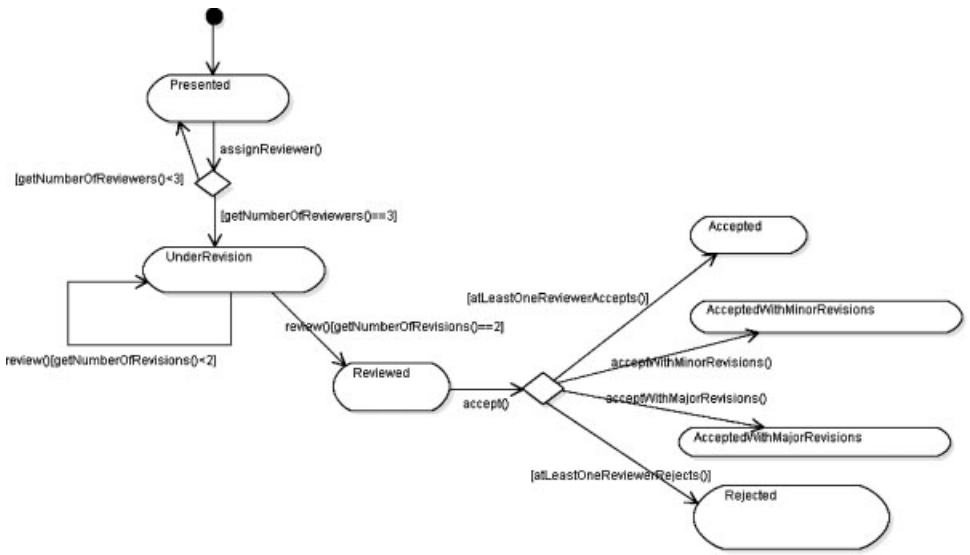


Figure 15. Desired behaviour for *Article* instances.

read using the *getDateOfReview* method (one of the default methods assigned by the tool); the second part of the condition (*getNumberOfReviewers (bd)<3*) requires a new method to be added to the class. Note that a parameter (*bd*) has been added to the signature of this operation. Since the generated application must access the database to know the actual number of reviewers assigned, this operation must receive a *Broker* parameter for connecting.

Required methods (*getNumberOfReviewers*, *accept*, etc.) can be added to the class at once using the class viewer (Figure 17), although its actual implementation must be given by the software engineer. The design of the final application imposes the use of the *bd* parameter, which is a *Broker*

DOI: 10.1002/smr

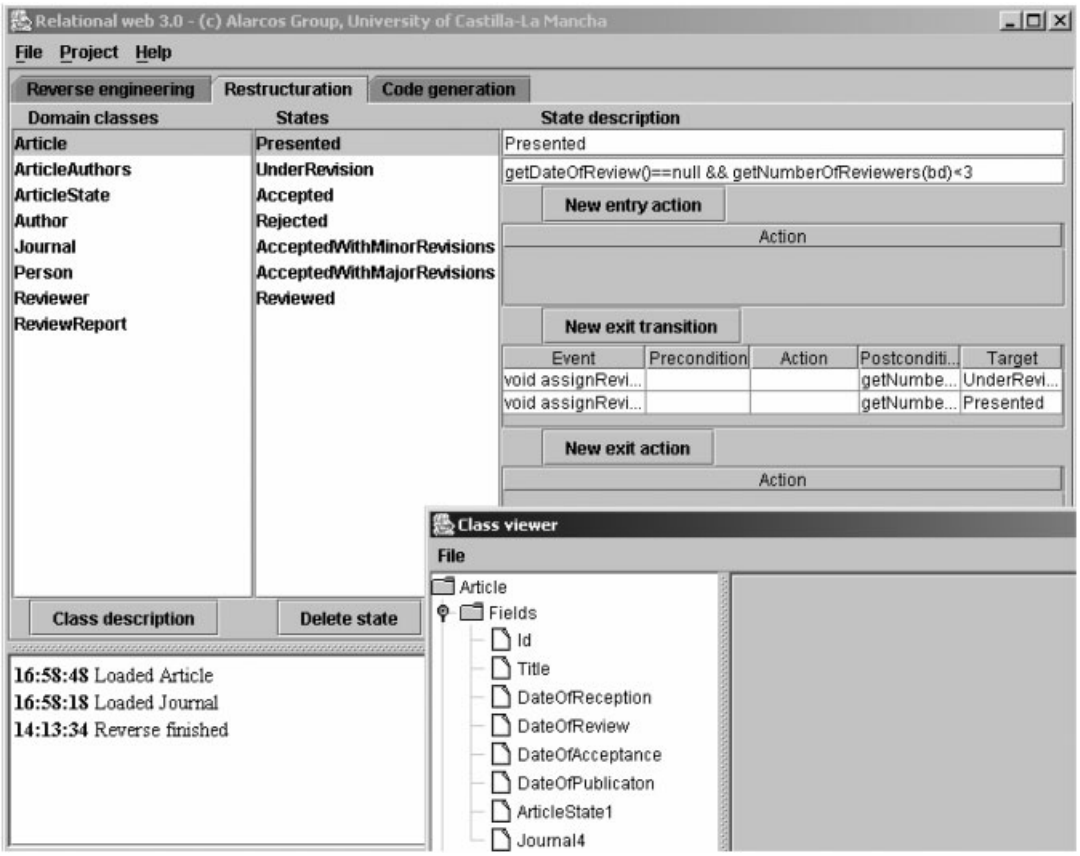


Figure 16. Restructuring panel with the class viewer in the foreground.

used by all methods accessing the database. The *Broker* is an implementation of the *Proxy* pattern [32] specialized as a *Database Broker* [34] representing the main access point to the database. The *Broker* class is generated for the four final platforms considered (*desktop Java*, *Java JSP*, *EJB* and *C# applications*). Thus, the state description shown in the previous figure (*getDateOfReview()==null && getNumberOfReviewers(bd)<3*) would be valid for any of them.

Exit transitions correspond to methods that can be executed on the class being described. The transition will be triggered if the instance is in a compatible state and if the possible precondition is verified. The triggering of a transition may imply the execution of one or more actions and, depending on post-evaluated conditions, a state change. The left side of Figure 18 shows an exit transition going from *UnderRevision* to *Reviewed* and is triggered when the *review* method is executed. *Reviewed* is reached if this is the third revision of the article. As a consequence of its execution, the system will report completion of the article revisions to the editor journal.

Since each state is described with a Boolean expression, the lack of preconditions does not invalidate the transition (the self-description of the state is actually a precondition).

DOI: 10.1002/smr

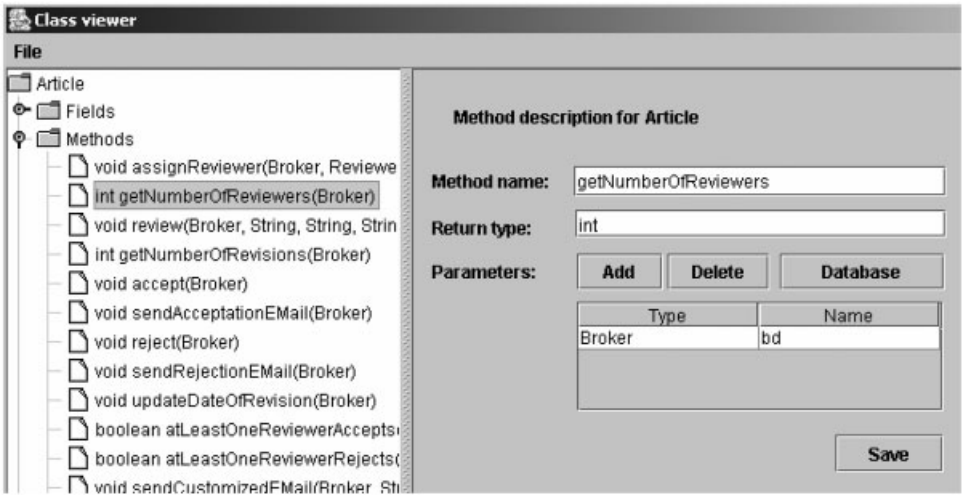


Figure 17. Addition of methods.

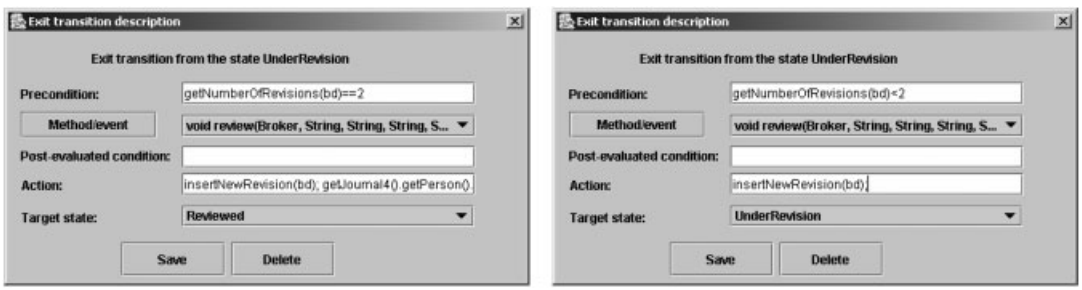


Figure 18. Description of exit transitions.

To generate the code, the tool adds a method for each method description given in the class viewer. In Figure 17, the operation *int getNumberOfReviewers(Broker bd)* is being described and translated into the first method shown on the right side of Figure 19.

For each state defined for a class, the tool generates a method to check whether the instance is in that state. The second method listed in the right-hand column in Figure 19 determines whether the instance state is *UnderRevision.* The method body is made up of the *return* keyword followed by the description given to the state in Figure 17:

return (getNumberOfReviewers<2).

Triggering events are also added as methods: according to the state machine shown in Figure 15, the *review* method can be triggered from the *UnderRevision* state, although the target state depends on two preconditions. On the left side of Figure 19, the code generated for the *review* method (whose two possibilities of execution were described in Figure 18) is shown. The given implementation checks, the first time, whether the instance is in an appropriate state (*stateIs_UNDERREVISION*)

DOI: 10.1002/smr

```
public void review(Broker bd, String email,
   String recommendation, String commentsToAuthors,
   String commentsToEditor)
{
     if (stateIs_UNDERREVISION(bd) &&
                  getNumberOfRevisions(bd)==2) {
          insertNewRevision(bd);
          getJournal4().getPerson().sendCompletedReview();
          setState_REVIEWED(bd);
          return;
     }
     if (stateIs_UNDERREVISION(bd) &&
                  getNumberOfRevisions(bd)<2) {
          insertNewRevision(bd);
          setState_UNDERREVISION(bd);
          return;
     }
}
```

```
public int getNumberOfReviewers(Broker bd)
{
   // TODO: complete method
   return 0;
}

public boolean stateIs_UNDERREVISION(Broker
   bd)
{
     return (getNumberOfReviewers(bd)<=2);
}

public void setState_REVIEWED(Broker bd)
{
   // TODO: entry actions for REVIEWED
}
```

Figure 19. Some of the methods generated.

and, the second time, the precondition (*getNumberOfRevisions(bd)==2* or *getNumberOfRevisions (bd)<2*).

3.5. Code generation

The previous sections have explained some concepts of the strategy for code generation, but focused on the domain layer of the final application. This section describes the policies followed to give the final application a suitable architectural design.

The tool builds a multilayer application from the instance of OOS annotated with state machines. The four main layers are *domain*, *controllers*, *presentation* and *pure fabrications*. The code generation step considers both classes and the relationships between classes. Thus, for each class K in *OOS*:

(1) A class K is added to the domain layer. K will have the default operations and those that have been added by the user in the state machine definition (restructuring) of K.
(2) A class FK is added to the presentation layer (F comes from 'form' or 'frame'). FK will have the adequate widgets (textbox, checkboxes, etc.) to manipulate instances of K, as well as the suitable buttons to execute the default operations on K. Moreover, if K has states, there will also be buttons that make it possible to execute operations depending on the state of K. For example, if an article is in the *UnderRevision* state, then its corresponding window will show a button to execute the *review* operation.
(3) A class *ListK* is added to the presentation layer, which will show listings of records of K. Selecting one record, the application will show the instance of K in the corresponding FK window.
(4) Depending on the final platform, a class CK that plays the role of a 'use case controller' [41] is added to the controller layer. If the application is JSP or EJB (where the presentation layer is composed of web pages), messages passed from FK (a web page) will be captured by CK, which translates them into calls to methods of the associated instance of K.

DOI: 10.1002/smr

(5) A *KPF* class is added to the pure fabrication tier. The pure fabrication pattern [23] advises delegating those operations that are not part of the business goal of the class to associated classes. Thus, if *domain.K* has the business methods, methods in *KPF* are in charge of allowing navigation among records of related tables (in other words, instances of related classes), obtaining listings in HTML, etc.
(6) Moreover, for each operation *O* in *K* described in the class viewer (Figure 17), the tool adds an *FO* class and sends (to the corresponding controller class, *CK*) the adequate instructions to deal with these messages. *FO* includes suitable widgets to send parameters to the corresponding operation.

Figure 20 illustrates some of the classes generated for the *Article* class after having reverse engineered the JSP application: *_formArticle* is the corresponding *FK* class (a web page); the remaining classes starting with *_form* correspond to some of the *FO* web pages generated to execute some of the operations defined for *K*. *_gestArticle* represents the *CK* class (controller), which will receive parameters and message calls from the presentation layer and will pass them to the associated domain instance (*Article*). *_listArticle* corresponds to the aforementioned *ListK* class that, for example, obtains HTML listings from the pure fabrication.

The background of Figure 21 shows the web page generated for the *review* operation in *Article.* In Figure 17, the *review* operation had three String parameters (*recommendation*, *commentsToAuthors* and *commentsToEditor*). The corresponding class in the presentation layer shows the complete record and includes three widgets to give values to the three parameters (in this example, and due to the lengths of these fields in the database, the tool has added three 'text areas' that have been manually changed by three text fields to reduce the size of the figure) and a button to send the *review* message instance. The figure also shows the possibility of selecting values from related classes: if the user presses the button highlighted in the background, the list of possible values is extracted by the pure fabrication and is shown on a new page. Then, any value ('Accepted' or 'Accepted

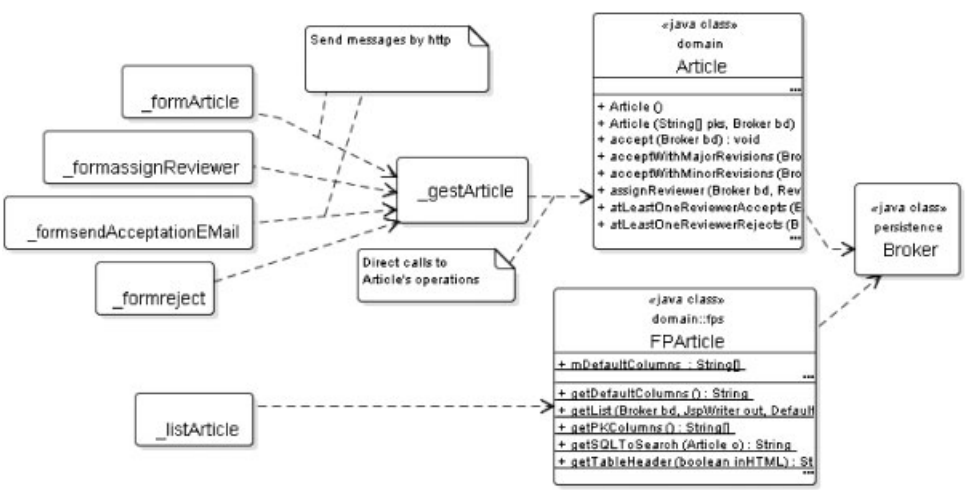


Figure 20. A view of the final application class diagram.

DOI: 10.1002/smr

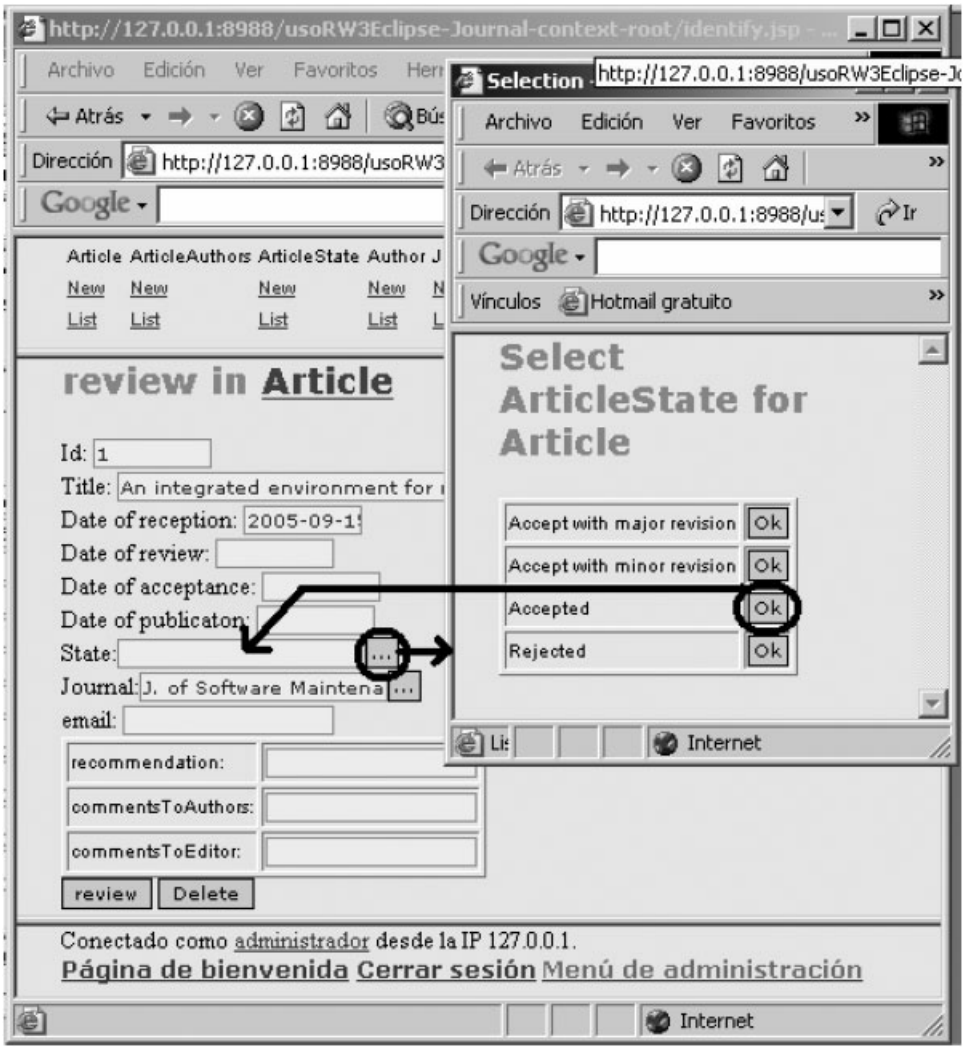


Figure 21. The *FK* class corresponding to the *review* operation in *Article*.

with minor revisions', for example) can be selected in the foreground and transported to the *State* widget.

Relationships in the *OOS* instance are used to add methods to the pure fabrication classes corresponding to the related classes. For example, the listing shown in the foreground window of the previous figure is obtained by the *selectArticleStateForArticle* static method, which has been automatically added to *FPArticle* by means of the relationship between *Article* and *ArticleState*, which in turn proceeds from the foreign key relationship existing between the corresponding two tables.

DOI: 10.1002/smr

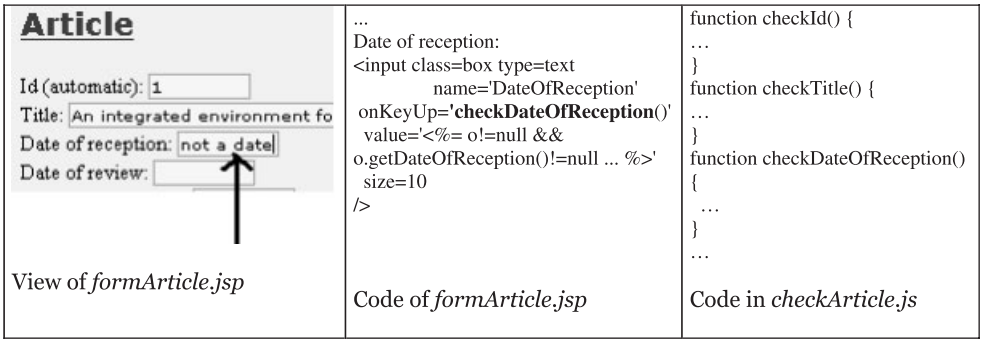


Figure 22. Scripts are added to validate parameters.

Additionally, the tool adds a set of fixed classes in charge of managing security, roles, users, etc. to the final application. In both standard JSP and EJB applications, for example, a set of web pages and servlets is added to the presentation layer to make it possible to create users, grant assignments, etc., always preserving the separation of business logic (the security logic, in this case) from its view. Also, if the original database had stored procedures, a specific window (or JSP page) is added for each one, containing the suitable widgets to assign values to the parameters. For web applications, the tool also adds a small Javascript program to each class with a set of functions to validate the values of the parameters before sending them to the server (Figure 22).

The code generation process may include the generation of the SQL code to migrate the original database to another manager (from Access to SQL Server, for example). The main difficulties in this step concern the conversion of data types, since each manager has a different set of data types (i.e., Access and Caché have the *boolean*, which is a *bit* both in SQL Server and in Oracle). The conversion is guided by a suite of configuration tables.

The persistence tier only has a class (the Database Broker) that centralizes access to the database from the domain tier. Persistence services are requested by the user, passing from the Presentation tier to the Broker through domain and pure fabrication classes.

3.6. Platform-specific metamodels

OOS (Figure 10) encapsulates the structures required to support class diagrams representing conceptual schemes for databases. However, each final platform has some particularities that make it advisable to use specific metamodels for each platform.

Thus, for example, the tool is capable of generating EJB 2.0 applications. In this case, an entity EJB is added to the domain tier. However, in order to make the bean findable and accessible for remote clients, the addition of a 'home' interface is required; in the same manner, a 'remote' interface is needed to call the business methods. Thus, the EJBs metamodel includes the presence of the 'interface', a new element that we did not consider in the description of OOS. C#, in addition to having fields and methods, also contains 'properties', a special type of operation that does not take parameters or have parentheses.

These metamodels, which are adapted to the target platform, correspond to the MDA's platform-specific metamodels.

DOI: 10.1002/smr

3.7. Customizing code generation

The code generated can be customized by adding new templates. Thus, for example, it is easy to connect the domain with the presentation layer through an intermediate layer of observers, by applying the MVC pattern: when any user changes the state of any instance, this transmits its new state to all the screens that are observing it. Figure 23 shows a class diagram where each domain class (*Article*, *Journal*) is associated with a collection of observers (defined as interfaces), which are implemented by the corresponding classes in the presentation layer.

If the complete application must be generated following this pattern, the template files corresponding to the presentation and domain classes should be modified adding the following: in presentation, the template will include an *implements* instruction to denote the implementation of the interface; the instructions to be altered in the domain template are highlighted in Figure 24; in Relational Web, patterns are written between the # symbol. Each time the tool finds a pattern, it substitutes it with the corresponding text. For example, the pattern #CLASS# is substituted by the class name (*Article*, *Journal*, *Person*...); thus, according to Figure 24, an *import* statement will be added to each domain class to import its corresponding observer, which will be placed into the *observers* package. The #COLLECTION_COMMENT# pattern is substituted by a comment (*doclet*) that is redacted according to the target IDE (JDeveloper or Eclipse), which will correctly draw the 1:n association.

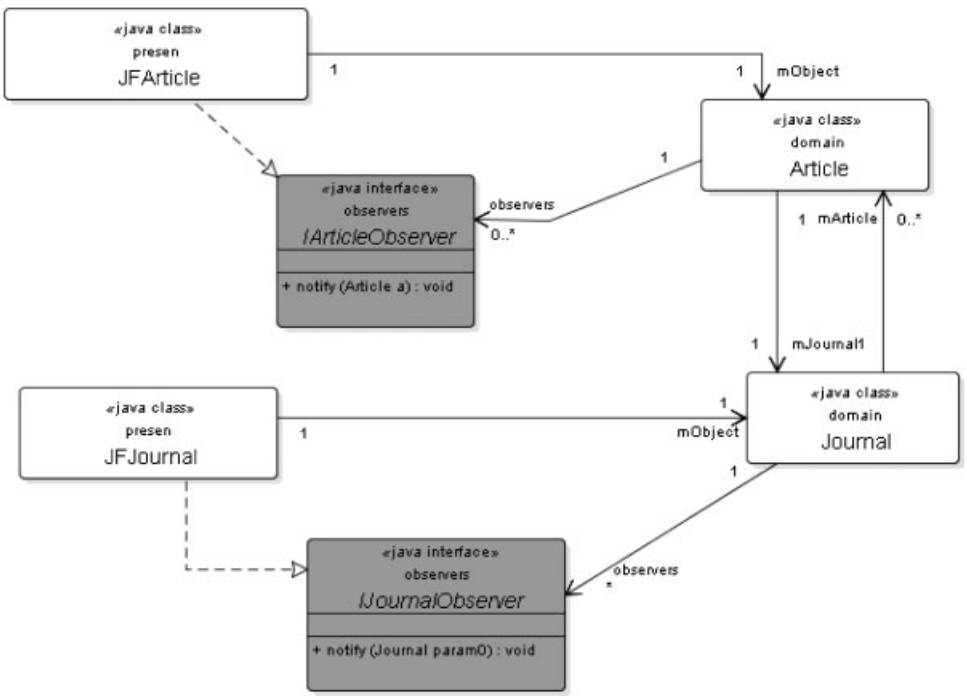


Figure 23. Addition of an *observers* layer.

DOI: 10.1002/smr

```
package domain;
import persistence.Broker;
...
import observers.I#CLASS#Observer;

public class #CLASS# {
    #DOMAIN_FIELDS#
    protected String[] oldPKs=new String[#PKS#];

    /**
    * #COLLECTION_COMMENT#
    */
    protected Vector observers[];

    ...

    public void update(Broker bd) throws SQLException {
            #UPDATE_CODE#
            for (int i=0; i<observers.length(); i++) {
                    I#CLASS#Observer o=(I#CLASS#Observer) observers.get(i);
                    o.notify(this);
            }
    }
}
```

Figure 24. Modifications to the domain class template.

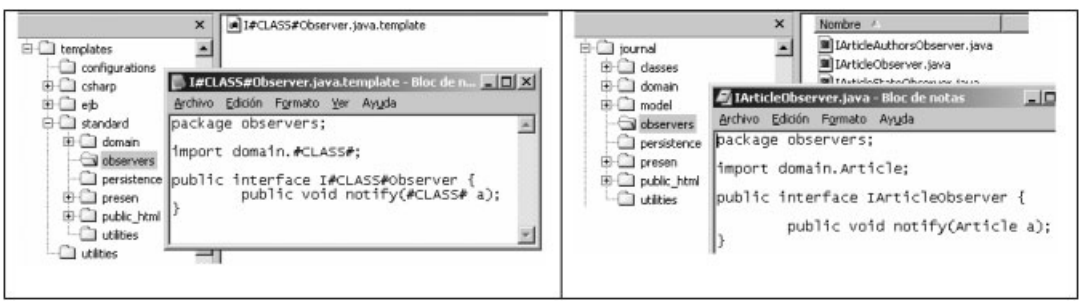


Figure 25. Original (left) and final structures of folders.

The template for the observer interface, which is not in the default set of templates for Relational Web, must be placed in a directory under the templates folder and must have 'template' as extension. The tool processes these files and substitutes the corresponding patterns. The left-hand side of Figure 25 shows the structure of the templates folder and the contents of the generic template added for building observers; the right-hand side shows the folders making up the structure of the generated application, the set of observers generated and a sample of the code obtained for one of them.

4. THE RE-ENGINEERING PROCESS AS AN MDA PROCESS

Section 2.3 reviewed some key concepts of MDA and summarized two common approaches for model transformation (rewriting rules based, and marks and templates based). The section concluded

DOI: 10.1002/smr

by affirming that the technique and tool proposed in this article is an intermediate proposals between both approaches.

Both the tool and the re-engineering process fulfill the basic ideas of MDA:

(1) Both the database and the OOS metamodels represent relational databases and class diagrams in a vendor-independent way. With these metamodels, one can represent the relevant characteristics (from the point of view of the described re-engineering process) of any relational database or class diagram. These metamodels then match with the computation-independent viewpoint.
(2) The instances of database and OOS represent actual databases and class diagrams. Indeed, all the factories appearing in Figure 7 translate their respective physical databases to our specific metamodel; later, the code is not generated from the OOS instance but translated into a platform-specific metamodel (Section 3.6). Thus, *Database* and *OOS* match with the Platform-independent viewpoints.
(3) Finally, the resulting database and the multilayer web application are completely platform dependent. For example, the SQL instructions required to build the new version of the database are quite similar, but they have small differences between one RDBMS and another. Also the JSP pages, classes, etc. are different when the application is based on standard Java classes or Enterprise Java Beans. Code generation is based on template files and patterns, concepts that correspond to the MDA ideas of templates and marks. For example, marks are used to specify the elements of the model for both relational and object-oriented systems by means of metamodels (as the definition of 'mark' explains [26]). Templates are used to specify how the elements (belonging to one metamodel) are transformed into elements of another model (belonging to another metamodel). Thus, templates contribute to specifying the transformations between instances of different metamodels.

5. CASE STUDIES

Relational Web has been successfully used by the authors in at least 12 projects and has been given to several software development companies for their own developments. Below we give brief descriptions of some of these projects, which are summarized in Table IV.

Alar.Net is the intranet of the Alarcos Research Group, where the authors of this article are undertaking their current research, and is one of the first projects developed with the tool. It is accessible to members from the web page of the group (http://alarcos.inf-cr.uclm.es). The intranet started from a relational database (Figure 26) which holds all the tables storing the information concerning the research group (members, publications, projects granted, theses, trips, participation in committees, etc.). According to the discussion of previous sections, Relational Web generates a complete multi-tier application (Figure 27). This one has a wide set of characteristics to manage all the information saved in the database. However, the users expected other functionalities, such as the generation of *curricula vitarum* for the members and the group in several formats (pdf, doc), sharing of the information to be published in the public site, granted projects summaries, etc. Also, the default user identification system provided by Relational Web required to be changed in order to use the Active Directory of the University. For this case study, the code generated by Relational Web is ready to be quickly modified to fulfill these requirements: actually, it is quite easy to add new functionalities, being less than 1.5 h the time spent in perfective maintenance tasks.

DOI: 10.1002/smr

Table IV. Some of the projects developed with Relational Web.

Project	Type	Database manager	# of tables	Platform	Additional functionalities
Alar.NET (alarcos.inf-cr.uclm.es)	Internal	MS Access	65	JSP	Users' identification with POP_3 server
					Generates personal, group and project curricula in several official formats
					Discussion forums
Escuela Superior de Informática (www.inf-cr.uclm.es)	Mixed	Oracle, later migrated to SQL Server	133	JSP	Users' identification with POP_3 server
	(supervised students)			Standard Java	Feeds the institution's public web page
					Generates .doc file with the Student Guide Broadcasts news
Printing house	External	SQL Server	107	JSP	
				C#	Control of productive process
					Control of times
Sports club	External	MySQL	32	JSP	Small adaptations

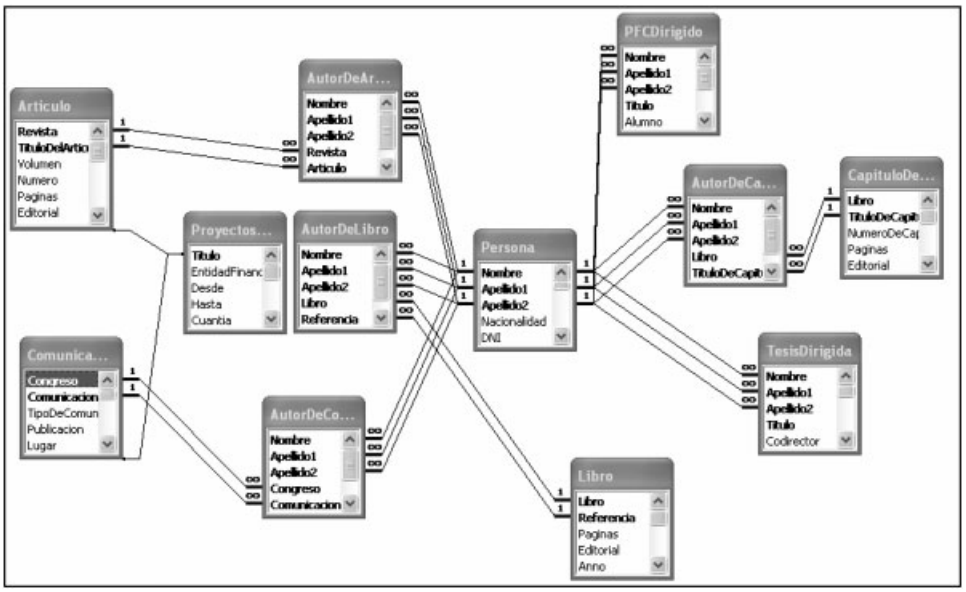


Figure 26. A small subset of tables and relationships.

DOI: 10.1002/smr

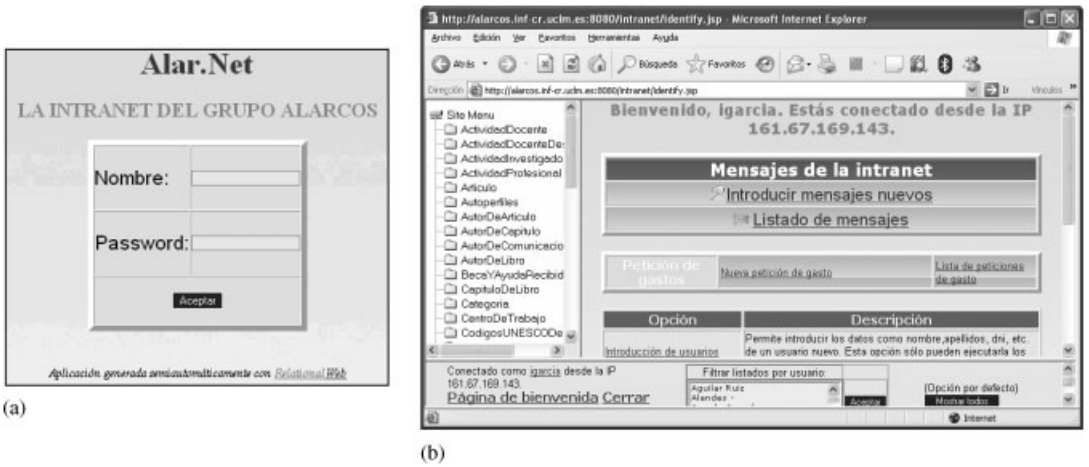


Figure 27. (a) Alar.Net welcome screen and (b) main screen.

The *School of Computer Science of the University of Castilla-La Mancha* has a set of dynamic web pages (*http://www.inf-cr.uclm.es*) whose information is extracted from an SQL Server database. (Oracle was initially used, but it was migrated due to several requests from programmers, who find it more comfortable to work with SQL Server.) These are maintained via an intranet built with Relational Web. The database has 133 tables and saves information about the school staff (personal information, and also departments, jobs, etc.), subjects of study, course scheduling, research groups, activities, scholarships, news, theses, etc. The intranet is used by all people working at the school and thus has almost 200 users. The identification procedure has also been adapted to use the university's Active Directory. One of the main uses of the system is the maintenance of the academic information of the subjects. Related to this feature, the system maintains a log with historical information. Thus, the tool is capable of generating the Student Guide in several formats and corresponding to different years. It also contains a module that sends news inserted by the intranet users by e-mail to subscribers. This module has been improved to send notifications to mobile phones as well. The person responsible for this project was, from 2001 to 2004, one of the authors of this article, despite now being a professor outside this research group. Each semester, the school offers two scholarships to collaborate in maintaining the web application, which are assigned to undergraduate students. They perform all types of maintenance tasks, such as improving the Web page, adding new functionalities to the intranet, data maintenance, user support, etc. The project has a high turnover rate of students, but the uniform and high-quality structure of the code generated requires little apprenticeship. Because of this, the time spent in comprehending the architecture of the final application, the code style, etc., is very short.

One of the first projects developed by an external organization manages the *productive process of a printing house*. Here, the generated code was modified to calculate the machines that each submission had to pass through. This application works with an SQL Server database containing 107 tables. Initially, the system was generated as a web application, although it was expanded with an additional C# program with a small set of tables to control the time that the staff dedicates to tasks, also making it possible to control the time consumed by each job.

DOI: 10.1002/smr

Very recently, Relational Web was used to build an application for managing a sports club. The application allows members to reserve courts, to know the state of their accounts, etc., and has additional administrative functionalities for the club staff. The application was generated from an SQL Server database, although the developers decided to migrate it to MySQL, which meant performing some very small changes in the generated source code.

It is important to note that, in all projects (both internally and externally developed), the mean time to perform either a corrective or a perfective maintenance task was less than 1.5 h. When the programmer knows how Relational Web structures the final application, he or she quickly detects where the changes to the code must be implemented. This good coding and architectural style also helps to preserve the high quality of the modifications. This opinion is shared by external developers, who have made interesting suggestions to the functionalities that can be added to the generated code. For example, the possibility of customizing the code generation process (Section 3.7) with the arbitrary inclusion of new templates was proposed by the developers of the sports club application.

Therefore, the technical scenarios where Relational Web can be used include all those related to the development of database-intensive applications. On the basis of the relational schema and the annotations made to the class model using state machines, the tool generates a ready-to-deploy application. The architecture of the application is organized according to the well-known architectural and design patterns. This fact leads to high understandability and maintainability of the code generated. Thus, even though the generated application does not fulfill all the expected requirements, these can be achieved and implemented in very short times. Today, the tool is not applicable to non-relational databases or to relational databases with a poor design. (For example, if the database has no foreign keys, Relational Web will not be capable of detecting relationships among tables and objects, thus limiting the functionalities of the generated application.)

6. CONCLUSIONS AND FUTURE WORK

This article has presented a complete re-engineering process that includes reverse engineering, restructuring and forward engineering, as well as a supporting tool for the automated generation of multilayer applications from relational databases.

On the basis of the architectural design and metamodels described here, the tool can generate four different types of applications from four different types of database managers (Caché Intersystems, Microsoft Access, Oracle and SQL Server). Its design also facilitates both the addition of new database managers to be used as input products and the implementation of new code generators.

The code generation process is based on the use of well-known design and architectural patterns (separation of layers, business operations in business classes, pure fabrication, database broker) that provide the code in its final application with a uniform, high-quality structure where responsibilities are adequately distributed in different classes and the architectural design is easily understandable. This means little effort to understand the structure and behaviour of the final application, as well as short maintenance intervention times, even when maintenance workers do not belong to the original development team.

The tool has been used in the development of several projects. The mean resolution time for corrective and perfective interventions is less than 1.5 h. Moreover, classical risks of software

DOI: 10.1002/smr

projects, such as the leaving of the expert domain staff is mitigated, due to the aforementioned characteristics of the code generated: this means that the new staff can be promptly familiarized with the ongoing projects.

As explained in Section 3, the core metamodel of Relational Web is based on object orientation. However, according to the MDA paradigm, the obtained instance of the OOS metamodel (Figure 10) could be translated into any other kind of representation, such as a structured application in, for example, C. Obviously, for this type of transformation, both the target metamodel and the suitable transformation rules should be defined and implemented.

Currently, the most important efforts are devoted to the implementation of new functionalities for: (1) generating .NET web applications that will share the domain tier with the existing C# generator; (2) exporting class models and state machines in XMI to make manipulation with Eclipse possible; and (3) importing Eclipse models to generate code.

ACKNOWLEDGEMENTS

This work has been possible, thanks to the Junta de Comunidades de Castilla-La Mancha (ENIGMAS project, PBI-05-058), Ministerio de Industria, Turismo y Comercio (FAMOSO project, FIT-340000-2005-161) and Ministerio de Educación y Ciencia (ESFINGE project, TIN2006-15175-C05-05).

The authors wish to thank the anonymous referees, who have made very important contributions to improve the work during the revision process.

REFERENCES

1. Chikofsky EJ, Cross JH. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 1990; **7**(1):13–17.
2. Chan K, Liang Z, Michail A. Design recovery of interactive graphical applications. *International Conference on Software Engineering*. IEEE Computer Society: Silver Spring MD, 2003; 114–125.
3. Tonella P, Potrich A. Static and dynamic C++ code for the recovery of the object diagram. *International Conference on Software Maintenance* (*ICSM'02*). IEEE Computer Society: Silver Spring MD, 2002; 54–65.
4. Borne I, Romanczuk A. Towards a systematic object-oriented transformation of a merise analysis. *Second Euromicro Conference on Software Maintenance and Reengineering* 1998; 213–225.
5. Alhajj R, Polat F. Reengineering relational databases to object-oriented: Constructing the class hierarchy and migrating the data. *Eighth Working Conference on Reverse Engineering* (*WCRE'01*). IEEE Computer Society: Silver Spring MD, 2001; 335–344.
6. Andersson M. Extracting an entity relationship schema from a relational database through reverse engineering. *Thirteenth International Conference on Entity-relationalship Approach* (*Lecture Notes in Computer Science*, vol. 881). Springer: Berlin, 1994; 403–419.
7. Chiang R, Barron T, Storey VC. Reverse engineering of relational databases: Extracting of an EER model from a relational database. *Journal of Data and Knowledge Engineering* 1994; **12**(2):107–142.
8. Hainaut JL, Henrard J, Hick JM, Roland D, Englebert V. Database design recovery. *Eighth Conferences on Advance Information Systems Engineering*. Springer: Berlin, 1996; 463–480.
9. Henrard J, Englebert V, Hick JM, Roland D, Hainaut JL. Program understanding in database reverse engineering. *Database and Expert Systems Applications (DEXA)* 1998; 70–79.
10. Pedro de Jesus L, Sousa P. Selection of reverse engineering methods for relational databases. *Third European Conference on Software Maintenance*. Nesi, Verhoef: Los Alamitos CA, 1998; 194–197.
11. Henrard J, Hick J-M, Thiran P, Hainaut J-L. Strategies for data reengineering. *Ninth Working Conference on Reverse Engineering*. IEEE Computer Society: Richmond VA, 2002.
12. Blaha M. A retrospective on industrial database reverse engineering projects—Part 1. *Proceedings 8th Working Conference on Reverse Engineering* (*WCRE'01*). IEEE Computer Society: Stuttgart, Germany, 2001; 136–147.
13. Blaha M. A retrospective on industrial database reverse engineering projects—Part 2. *Proceedings 8th Working Conference on Reverse Engineering* (*WCRE'01*). IEEE Computer Society: Stuttgart, Germany, 2001; 147–156.
14. Leavit N. Whatever happened to object-oriented databases? *IEEE Computer* 2001; **33**(8):16–19.

DOI: 10.1002/smr

15. Brown K, Whitenack BG. *Crossing Chasms*: *A Pattern Language for Object-RDBMS Integration*, 1995. http://www.smalltalktraining.com/articles/staticpatterns.htm [15 August 2007].
16. Shoval P, Shreiber N. Database reverse engineering: From the Relational to the Binary Relationship model. *Journal of Data & Knowledge Engineering* 1993; **10**:293–315.
17. Castellanos M. A methodology for semantically enriching interoperable databases. *Eleventh British National Conference on Databases*, 1993; 58–75.
18. Tari Z, Bukhres O, Stokes J. The reengineering of relational databases based on key and data correlations. *Searching for Semantics: Data Mining, Reverse Engineering, etc.*, Spaccapietra S, Maryanski FJ (eds.). Chapman and Hall, 1997.
19. Soon L-K, Ibrahim H, Mamat A. Constructing object-oriented classes from relations. *International Symposium on Information and Communication Technologies (M2USIC)* 2005; 313–316.
20. Premerlani WJ, Blaha MR. An approach for reverse engineering of relational databases. *Communications of the ACM* 1994; **37**(5):42–49.
21. Pérez J, Ramos I, Anaya V, Cubel J, Domínguez F, Boronat A, Carsí J. Data reverse engineering of legacy databases to object oriented conceptual schemas. *Electronic Notes in Theoretical Computer Science* 2002; **74**(4):1–13.
22. Yeh D, Li Y. Extracting entity relationship diagram from a table-based legacy database. *Ninth European Conference on Software Maintenance and Reengineering* (*CSMR'05*). IEEE Computer Society: Silver Spring MD, 2005; 72–79.
23. Larman C. *Applying UML and Patterns*. Prentice-Hall: Upper Saddle River, NJ, 1998.
24. Keller W. Mapping objects to tables. A pattern language. *European Pattern Languages of Programming Conference*, 1997.
25. Polo M, Piattini M, Ruiz F. Reflective Persistence (Reflective CRUD: Reflective Create, Read, Update and Delete). *Sixth European Conference on Pattern Languages of Programs* (*EuroPLOP*). Universitätsverlag Konstanz GmbH: Irsee, Germany, 2001; 69–85.
26. OMG, *MDA Guide Version 1.0.1*. 2003.
27. Bézivin J. Model engineering for software modernization. *Guest Talk in the 11th IEEE Working Conference of Reverse Engineering*, 2004.
28. Boronat A, Carsí JÁ, Ramos I. Automatic reengineering in MDA using rewriting logic as transformation engine. *Ninth European Conference on Software Maintenance and Reengineering* (*CSMR'05*). IEEE Computer Society: Manchester, U.K., 2005; 228–231.
29. McCombs T. *Maude 2.0 Primer. Version 1.0*. 2003. http://maude.cs.uiuc.edu/primer/maude-primer.pdf [15 September 2005].
30. Allilaire F, Idrissi T. ADT: Eclipse development tools for ATL. *Second European Workshop on MDA*, 2004; 9.
31. Bézivin J, Jouault F, Valduriez P. *An Eclipse-based IDE for the ATL Model Transformation Language*. University of Nantes, Nantes, 2005.
32. Gamma E, Helm R, Johnson J, Vlissides J. *Design Patterns. Elements of Reusable of Object-oriented Software*. Addison-Wesley: Reading MA, 1995.
33. Ambler SW. *The Object Primer*: *Agile Model-driven Development with UML 2.0*. Cambridge University Press: Cambridge, 2004.
34. Buschman F. *A System of Patterns*: *Pattern-oriented Software Architecture*. Addison-Wesley: Reading MA, 1996.
35. Calero C, Piattini M, Genero M. Empirical validation of referential integrity metrics. *Information & Software Technology* 2001; **43**(15):949–957.

AUTHORS' BIOGRAPHIES

Macario Polo has a PhD in Computer Science from the University of Castilla-La Mancha and a MSc degree from the University of Seville. His research areas include automation of software processes, especially testing and maintenance.

DOI: 10.1002/smr

Ignacio García-Rodríguez has a PhD and a MSc degree in Computer Science from the University of Castilla-La Mancha. His main research areas include migration of legacy systems towards SOA architectures.

Mario Piattini is a full professor at the School of Computer Science in the University of Castilla-La Mancha. He received his PhD and MSc degrees in Computer Science from the Politechnical University of Madrid, as well as an MSc in Psychology from the National University of Distance Education.

DOI: 10.1002/smr