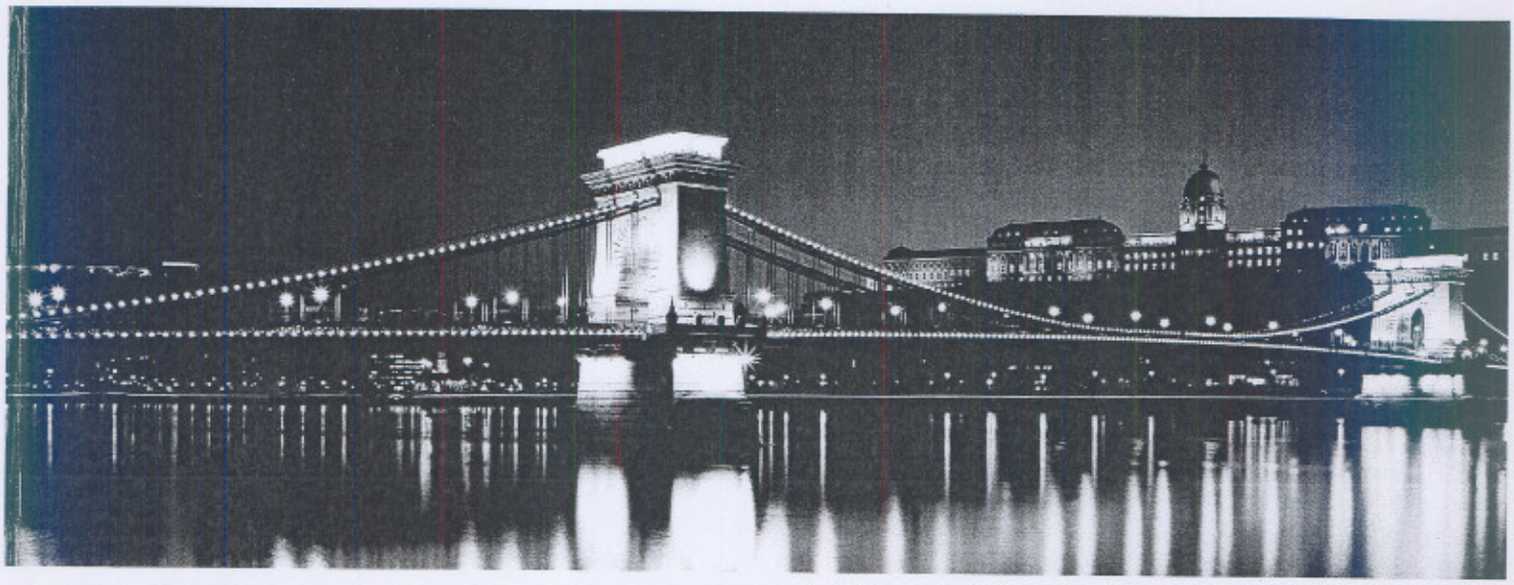


ICSM 2005

Proceedings

IEEE COMPUTER SOCIETY

IEEE

International Conference on Software Maintenance 2005

Budapest, Hungary,
26-29 September, 2005
www.inf.u-szeged.hu/icsm2005

Published by the IEEE Computer Society
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314

IEEE Computer Society Order Number P2368
ISBN 0-7695-2368-4
ISSN 1063-6773

ISBN 0-7695-2368-4
90000
9 780769 523682

Proceedings of the

21st IEEE International Conference on Software Maintenance

ICSM 2005

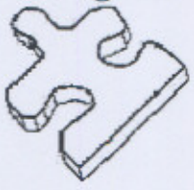
Proceedings of the

21st IEEE International Conference on Software Maintenance

ICSM 2005

Sponsored by
University of Szeged

In cooperation with
Reengineering Forum



IEEE COMPUTER SOCIETY

IEEE

Los Alamitos, California
Washington • Brussels • Tokyo

IEEE Computer Society Order Number P2368
ISBN 0-76952368-4
ISSN 1063-6773

Additional copies may be ordered from:

IEEE Computer Society
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314
Tel: + 1 800 272 6657
Fax: + 1 714 821 4641
http://computer.org/cspress
csbooks@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
Tel: + 1 732 981 0060
Fax: + 1 732 981 9667
http://shop.ieee.org/store/
customer-service@ieee.org

IEEE Computer Society
Asia/Pacific Office
Watanabe Bldg., 1-4-2
Minami-Aoyama
Minato-ku, Tokyo 107-0062
JAPAN
Tel: + 81 3 3408 3118
Fax: + 81 3 3408 3553
tokyo.ofc@computer.org

Individual paper REPRINTS may be ordered at: reprints@computer.org

Editorial production by JD Cantarella

Cover art design by Attila Matai, Kreativmuhely, Hungary

Cover art production by Joseph Daigle, Studio Productions

Printed in the United States of America by The Printing House

IEEE Computer Society
Conference Publishing Services
http://www.computer.org/proceedings/

Table of Contents

21st IEEE International Conference on Software Maintenance
ICSM 2005

Distributed Systems

Maintenance

Re- and Reverse Engineering

Source Code Analysis

Maintenance in Practice

Regression Testing

Theoretical Maintenance

Testing I

Evolution

Testing II

Web Maintenance & Reengineering

Short Papers

Maintenance & Evolution

Program Comprehension

AOP & Web

Testing III

PhD Dissertation Session

Panels

An integrated environment for reengineering

Ignacio García-Rodríguez de Guzmán[1], Macario Polo[2], Mario Piattini[2]
[1]*Escuela Superior de Ciencias Experimentales y Tecnología; Universidad Rey Juan Carlos*
c\ Tulipán, s/n, 28933 – Móstoles, Madrid (Spain)
[2]*Escuela Superior de Informática; Universidad de Castilla-La Mancha*
Paseo de la Universidad, 4; 13071-Ciudad Real (Spain)
ignacio.garcia@urjc.es

Abstract

This paper presents a tool specifically designed for database reengineering. As is well known, reengineering is the process of (1) applying reverse engineering to a software product to get 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 of artifacts involved in the whole process.

The starting point of the reengineering process is the physical schema of the database, which is translated into a vendor-independent metamodel; then, this is translated into a class diagram representing the possible conceptual schema used during the development 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.

Keywords: *reengineering, reverse engineering, model-driven reengineering.*

1. Introduction

The goal of reverse engineering is to obtain the representation of an existing software system in a higher level of abstraction than the original [2]. Nowadays, reverse engineering is often used to recover conceptual and architectural models of old systems that are later used as the basis for a "forward engineering" stage that produces a new version of the system adapted to other environments and paradigms, such as object-oriented, distributed computation, component-based software, etc.

According to [1], the whole process of reverse-and-forward engineering corresponds to the idea of "reengineering".

Reengineering can be understood as the composition of several transformation functions operating on different sets of artifacts, increasing or decreasing the abstraction level of the software product. Additionally, there can be intermediate stages of restructuring that change some intermediate software products with no exit from the abstraction level.

Usually, the initial set from which the reengineering starts (i.e., the set where the reverse engineering stage is used) is the program source code, with the final set of this first transformation function being class models or other types of diagrams. There have also been many works that apply reverse engineering to other software products, such as databases. The goal of most of these works is to produce a diagram representing the possible conceptual schema used during the initial development of the database and therefore, the target product of the reverse engineering is often an entity-relationship (ER) or extended entity-relationship (EER) diagram [6, 10].

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 management system). However, as databases are usually managed by external programs, obtaining the conceptual scheme in another format, such as a UML class diagram, would help the construction of both the new version of the database and of the program or set of programs in charge of managing it.

When a new information management 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 (which is a relational database in most cases,[9]), in this way approaching 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 [3]: the first possibility obtains a table from each class in the diagram, translating associations, aggregations and inheritance relationships into foreign key constraints (this is

1063-6773/05 $20.00 © 2005 IEEE

known as the "one class, one table" transformation pattern); with the second possibility, a table is built for each inheritance path in the class model ("one inheritance path, one table"); with the third one, a whole inheritance tree is translated into a single table ("one inheritance tree, one table"). Each possibility has some advantages and some drawbacks and it is therefore common to apply different combinations of these methods to the same class diagram.

In a three-tier 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: in fact, the presentation layer receives messages from the user and sends them to the adequate class in the business layer, which may require the execution of some operation with the database via the classes in the persistence layer. Obviously, other tiers or subtiers may be required depending on the type of application.

In this paper we present a tool, developed in Java, that integrates a complete process of reengineering, including the reverse engineering and the forward engineering stages. The starting point of the tool is a database with a physical schema reverse-engineered into a database-independent class diagram. This is passed as input to an automated code generation process, which builds a multitier system using different programming languages and platforms. The tool also includes the possibility of migrating a database from one to another database management system. The architectural design of the tool makes the easy addition of new code generators for other programming languages possible. The tool spurts as an evolution of the work we presented in [11].

The method is based on mathematical descriptions of all sets of artifacts and functions involved in the reengineering process, as Broy claimed in [4], Software Engineering requires mathematical descriptions for its models aspects, description techniques and development methods. For this author, such a mathematical theory should give semantics to usual description techniques (such as class diagrams, state charts, etc.) and should explain methods from a mathematical rationale more in the form of a "Formal Description Technique" than as a "Formal Method". Formal Description Techniques should lead to a deeper understanding of software processes and to a more powerful tool support.

The paper is organized as follows: Section 2 is divided into a set of epigraphs, each one explaining a different step in the reengineering process. The code generation process is presented and explained in Section 3. In Section 4 we analyse the correspondence of the proposed reengineering process with the model-driven architecture and with how we adapt the MDA approach to our tool. In Section 5 we present our conclusions and future lines of work..

2. Description of the reverse engineering stage

Figure 1 presents a general view of the complete process implemented in the tool: (1) by means of a Connection object, a factory (the DBFactory class) accesses the database's data dictionary and (2) builds an instance of Database, a class containing a metamodel for representing relational databases with no dependence of the vendor. Then, (3) the BDR2OO class applies a reverse engineering algorithm to the Database instance and (4) obtains an instance of *OOS* (*O*bject *O*riented *S*ystem), which is a metamodel to represent UML class diagrams. The user can manipulate the set of classes included in the OOS instance, adding new operations by means of state machines; then, (5) one of the code generators is used to generate the final application according to Table 1. Section 3 depicts the algorithms used for generating code.

2.1. Obtaining the Database instance

As Figure 1 shows, a DBFactory accesses the data dictionary of the database used as starting point of the reengineering process by means of a java.sql.Connection object. Connection is an interface including several operations to recover many database metadata, such as the table names, table structure (name, label, type and length of columns, for example), foreign key relationships, etc. However, not all database vendors implement all the operations in this interface or, if they do, the structure of the results of each operation can vary one from another.

Therefore, the use of a different way to access the database is required, depending on the relational database management system we are dealing with. To handle this, we consider *DBFactoy* as an abstract factory pattern [7] that has as many specializations as

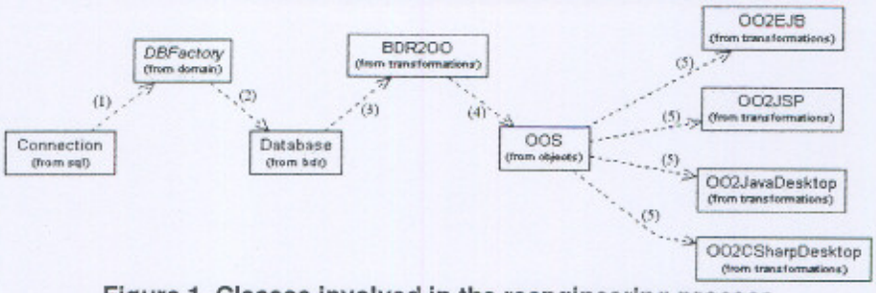


Figure 1. Classes involved in the reengineering process

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

Tabla 1. General characteristics of the code generators

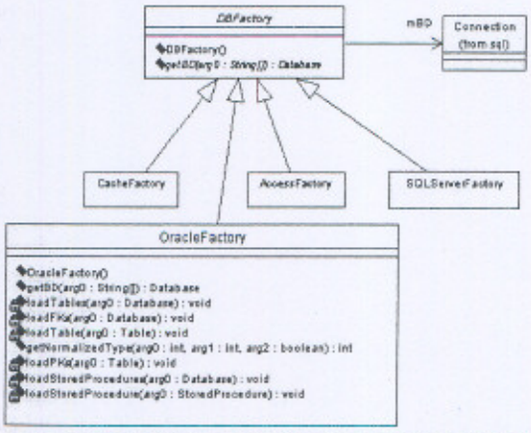


Figure 2. Factories to recover the database str ucture

types of databases that must be processed: today, the tool can manipulate databases implemented in Oracle, Caché Intersystems, SQL Server and Microsoft Access,
meaning that four concrete factories exist (Figure 2). Some of these use vendor implementations of the standard connection interface.

Database constitutes a metamodel to represent the relevant structures of a relational database from the point of view of our reengineering process. Thus, *Database* saves information about tables and their columns, foreign keys and stored procedures (Figure 3).

2.2. Reversing the database

The reverse engineering stage is carried out by the *BDR2OO* classifier shown in Figure 1. Its goal is to obtain an object-oriented class diagram representing the possible conceptual model used during the development of the database.

It takes one instance of the above-described *Database* and gets one instance of *OOS*, which represents the metamodel of a generic set of related classes (Figure 4).

Each *Class* may include a state machine that is added to the class by the tool user to give their instances a behaviour different than the default one. This is explained in Section 2.4.

Database = (Tables, FKs, SPs)

t ∈ Tables = (name, Columns)

c ∈ Columns = (name, type, isPartOfPK, isPartOfUniqueIndex)

fk ∈ FKs = (mainTable ∈ Tables, secTable ∈ Tables,

mainCols⊆ mainTable.Columns, secCols⊆ secTable.Columns)

sp ∈ SPs = (String)

Figure 3. Database metamodel

This definition of "Object oriented system" is valid to define each tier of the final application that the tool generates, since each tier is a subset of the final object-oriented application. Supposing three tiers in the final application, this one could be described as follows:

FinalApp=(PresentationTier, DomainTier, PersistenceTier)

...being *PresentationTier*, *DomainTier* and *PersistenceTier* instances of *OOS*.

The three basic methods (briefly described in Section 1) to translate a conceptual schema into a relational database can be used for the opposite process: our tool produces the class diagram supposing that the "one class, one table" pattern was used during the database development.

The function in charge of making the translation is shown below (Algorithm 1).

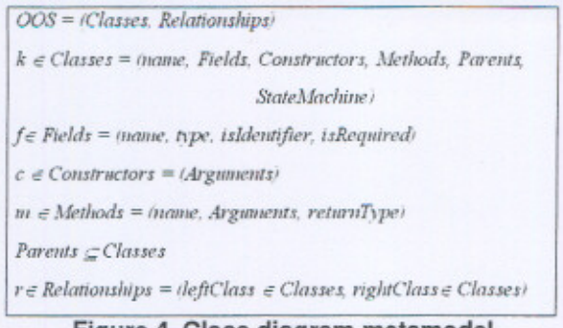


Figure 4. Class diagram metamodel

In lines 3-9, the algorithm builds a class for each table in the database. In lines 10-15, an association is created for each foreign key in the database. The function called in line 16 is in charge of removing the fields that are representing the association between both classes. Lines 17-26 translate into inheritance relationships, those associations that proceed from foreign keys whose primary and secondary tables are related through their respective primary keys, thus composing a 1:1 foreign key.

In line 27, the set of fields in each class is completely defined, and the algorithm has all the information to add constructors and methods. By default, each class receives:

- A constructor with no parameters, which builds "empty" instances assigning a default value to each field.
- A materializer constructor, which is used to build instances of the corresponding class from the records saved in the database. This constructor takes as argument the values of the primary key corresponding to the record to be materialized.
- Methods to insert, update and delete instances to and from the database.
- Set/Get methods to write and read field values.

For the conclusion, lines 31-33 process the set of relationships in the object-oriented system to add methods to navigate across related instances, corresponding to records of related tables.

The previous function, which operates on *Database* to produce an *OOS*, is implemented as a method called *getOOS* in the *BDR2OO* class shown in Figure 1.

2.3. Tuning the OOS for code generation

Figure 5 shows the main screen of the reengineering tool after having reverse-engineered the small banking database of Figure 6, used as example: the selected tab (JSP application) contains a tree showing the structure of the project to be generated (in the domain branch of the tree we can see a set of tables but many of them do not belong to the relational database of the Figure 6, because these tables are added in order to provide functionalities like security and access control to the end application); on the right side, a new tab is opened when the user double-clicks on a file name. This right side also includes a special tab to configure the code generation options. It is possible to add state machines for defining the behaviour of domain classes, as well.

The piece of code shown in Figure 5 corresponds to the *CreditCard* class that proceeds from the *CreditCard* table in the database. According to the *getOOS* algorithm previously described, the tool has translated the 1:1 foreign key relationship between *CreditCard* and Product into an inheritance relationship; the 1:n relationship between *CreditCard* and *Account* has been translated into a single field (*mAccount1)* of the *Account* type. As the remaining fields do not proceed from columns involved in foreign keys, they are translated into fields of compatible types.

2.4. Adding state machines to define the behavior

State machines can be attached to domain classes to provide them with non-default behaviour and non-default methods. Figure 7 describes a same state machine to describe the possible behaviour of *Account* instances using Rational Rose (top side) and our tool (bottom side). In our tool, each state receives:

- A name, such as *NegativeBalance*.
- A description, which is a boolean expression that denotes whether the instance is or is not in the state. Usually, this expression is written as a function of both the class fields and observer methods. In the bottom side of Figure 7, the state description denotes that accounts are in the sate *NegativeBalance* when *getBalance()* is less than zero.
- A set of output transitions. Each one has a triggering event, maybe a condition (which is evaluated after executing the event), one or more actions and a target state. Output transitions in the bottom side of Figure 7 denote that, after executing the *deposit(double amount)* method, a condition is evaluated; if the condition is true, then the instance will execute the corresponding action and will put the object in the target state.
- Optionally, a set of entry and exit actions, which are instructions to be executed every time that the instance arrives or goes out from the state.

In standard JSP, standard Java and standard C# applications, each domain class receives, by default, an "empty" constructor (which builds instances assigning

```
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.type, c.isIdentifier, c.isRequired) }
7           }
8           result.Classes = result.Classes ∪ { k }
9       }
10      ∀ fk∈db.FKs {
11          r : Relationships = (∅, ∅)
12                  r.leftClass= findClass(result, fk.mainTable)
13                  r.rightClass= findClass(result, fk.secTable)
14          result.Relationships = result.Relationships ∪[1]
15      }
16      removeColumns(result)
17      ∀ fk ∈db.FKs {
18          mainTable =fk.leftClass
19          secTable = fk.rightClass
20          allMainColumnsArePK(fk) ^ allSecColumnsArePK(fk) ^
21              |getPKs(mainTable)|=|fk.mainColumns| ^ |getPKs(secTable)|=|fk.secColumns| ⇒
22                  r.Relationships = r.Relationships – findRelationship(fk) ^ {
23                          c=findClass(result, secTable)
24                          c.Parents=c.Parents ∪ { findClass(result, mainTable) }
25                  }
26      }
27      ∀ k ∈result.Classes {
28          addConstructors(k)
29          addCRUDMethods(k)
30          addSetGetMethods(k)
31      }
32      ∀ r ∈result.Relationships {
33          addNavigationMethods(r.leftClass)
34      }
35      getOOS = result
36  }
```

Algorithm 1. Function to obtain an instance of the OOS metamodel

a default value to each field), a materializer constructor (builds instances from records saved in the database, the insert, delete and update methods and set/and ge methods for each field. In Enterprise Java Beans applications, the set of methods receive different names (there are some standardized names for entity EJBs, such as EJBLoad, EJBStore, etc.), although the functionality is essentially the same.

The processing of domain classes with attached state machines produces the addition of new fields and methods into the generated class:

- For each state in the state machine, a boolean field is added.
- For each state in the state machine, a boolean method called *stateIs_STATE_NAME* (where STATE_NAME corresponds to name given to the state), which returns true if the expression used to describe the state is true and false otherwise.
- For each state in the state machine, a void *setState_STATE_NAME*, whose body includes the set of entry actions added to the state.
- For each event (taking into account all the events in all the states of the class), a new method is added to the class. The execution of this method will only be possible when the instance state has the event in its set of output transitions. For example, from the top side of Figure 7, it is clear that the *withdraw* event cannot be executed in the *BalanceNegative* state. Therefore, the generated method includes a set of conditional instructions to check the adequate state of the instance.
- The class also receives a *recalculateState* method, which checks all the *stateIS_*

methods to put the instance in the corresponding state. When one of the *stateIS_* methods returns true, then the corresponding *setSTATE_* method is executed. Note that it is important to preserve disjoint?? all the state descriptions.

For the example shown in Figure 7, the tool generates (besides the default code), the fields and methods shown in Figure 8.

3. Forward engineering stage

The instance of *OOS* got from the reverse engineer stage corresponds to the domain tier of the application to be generated. Moreover, this domain tier is used as the starting point to generate the remaining tiers.

Following the basis of the MDA, we use different templates and marks in order to generate suitable source code for different target applications. By means of these templates and marks, we can obtain a PSM from our PIM (see [9]).

3.1. Presentation tier

Basically, the presentation tier is composed of a set of screens that allows the user to manipulate instances. The format of each screen obviously depends on the target system, but all of them share the same design principles. So, for each domain class:

- One screen is added to manipulate its instances. By default, the window includes buttons to call the insert, delete and update methods. Moreover, the related instances/records of other classes/tables appear in a reserved area of the same screen.
- One screen is added to get lists of records of all tables. Interacting on these lists, the user can call the materializer constructor of the class for loading the instance into its corresponding manipulation screen.

Figure 9 shows some of the JSP pages and Frames generated by the tool to manipulate accounts. Since *Account* has been annotated with one state machine, additional screens are added to the final application for executing them. Screens for operations include the widgets corresponding to the "primary key" and "alternative key" fields (i.e., fields proceeding from the primary and alternative key columns) and as many widgets as the operation has primitive parameters.

3.2. Pure fabrication tier

As we said, our tool automatically generates multitier applications. In order to make these applications easier to maintain, modify or migrate, we have included another design pattern in the application´s architecture, the pure fabrication[8].

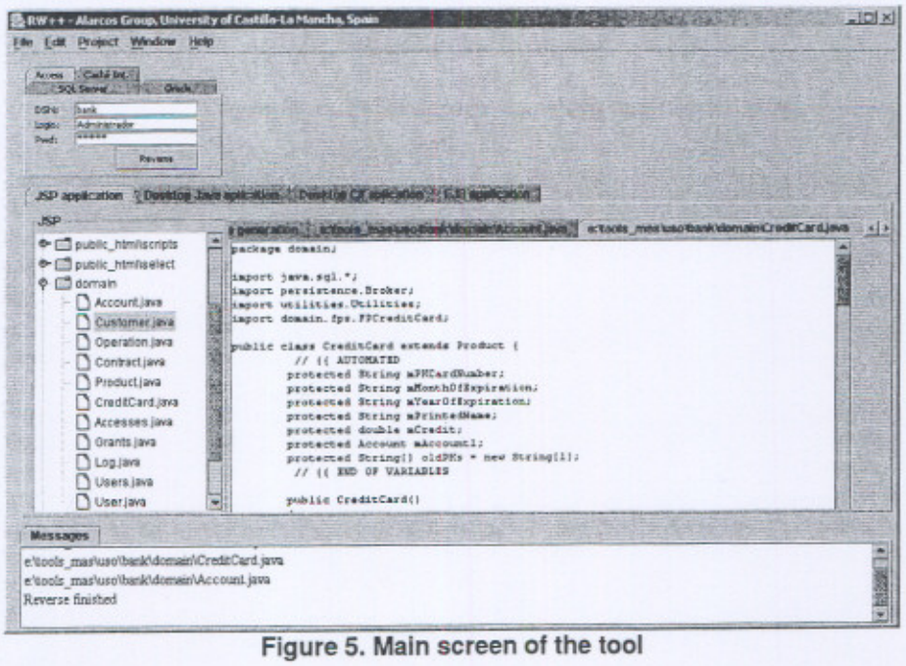


Figure 5. Main screen of the tool

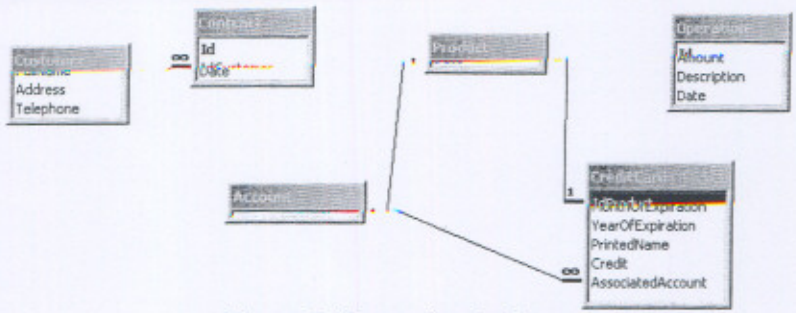


Figure 6. A sample database

```
private boolean _BALANCEZERO = false;
private boolean _BALANCENEGATIVE = false;
...
public boolean statels_BALANCEZERO() {
        return (getBalance()==0);
}

public boolean statels_BALANCENEGATIVE() {
        return (getBalance ()<0);
}

public void setState_BALANCEZERO() {
public void setState_BALANCEPOSSITIVE() {
}
```

```
public void setState_BALANCENEGATIVE() {

public void deposit(double amount) {
    if (statels_BALANCEZERO()) {
        Operation o=new Operation(this.mId_Product, amount), add(o);
        setState_BALANCEPOSSITIVE();
        setState_BALANCEPOSSITIVE();
    } else if (statels_BALANCENEGATIVE()) {
        Operation o=new Operation(this.mId_Product, amount); add(o);
        recalculateState();
    }
}

public void withdraw(double amount) {
        Operation o=new Operation(this.mId_Product, amount), add(o);
        recalculateState();
    }
}
```

Figure 8. New fields and methods proceeding from Figure 7

Our tool builds a pure fabrication class for each the relationships of the domain class with other domain classes. To generate the pure fabrications, the tool processes all the relationships contained in the *OOS* (corresponding to the main table in the original foreign key).

In our banking example, since *Account* has two 1:n relationships with *Operation* (inherited through *Product)* and with *CreditCard,* two methods called *loadRelatedOperation* and *loadRelatedCreditCard* are These methods are responsible for recovering the small lists of credit cards appearing in Figure 9.

3.3. Persistence tier

The persistence tier has only one class that centralizes *Pure fabrication classes.*

3.4. Other tiers

Depending on the target application, the final system may have additional tiers to include other functionalities. In the standard JSP and EJB applications, for example, when the user presses a button in the web page to execute an operation, the server, then, the servlet instantiates the corresponding class or EJB, calls the appropriate setter methods to give the instance fields the desired values and, then, executes on the instance the method corresponding to the button pressed (insert, update… or deposit, withdraw, etc.).

Additionally, the tool adds a set of fixed classes in charge of managing the

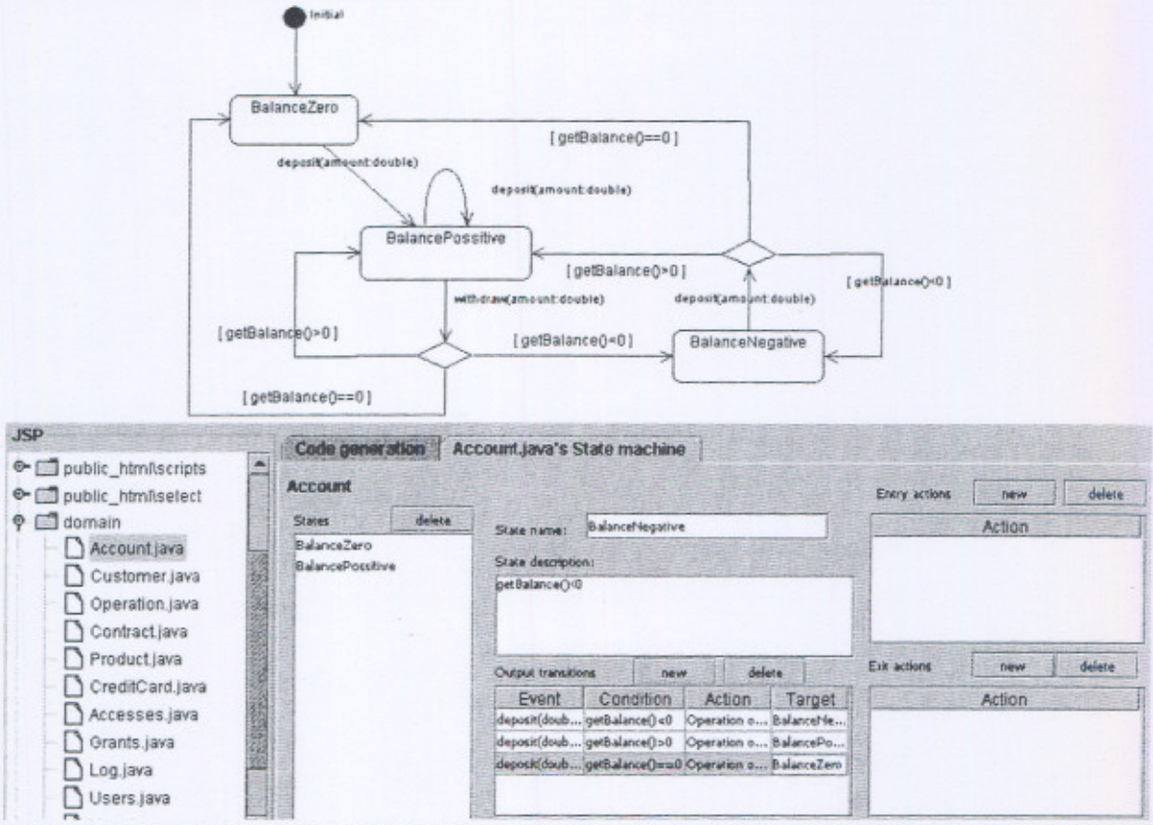


Figure 7. A possible state machine for Account, described with Relational Rose (top) and with out tool (botton)

user to create users, grant assignments, etc., always preserving the separation of business logic (the security logic, in this case) from view.

3.4. Platform-dependent metamodels

OOS encapsulates the required structures to represent generic class diagrams. However, each final platform has some particularities that advise using specific metamodels for each platform.

In EJBs, for example, an Entity EJB is added to the domain tier from each table. However, in order to make the bean findable and accessible for the remote clients, the addition of a "home" interface is required; in the same manner, a "remote" interface is needed to call the business methods. In this case, we note the presence of the "interface", a new element that we did not consider in the description of *OOS*. In C#, besides the fields and the methods, there exist "properties", a special type of methods that do not take parameters or have parenthesis.

As an example, Figure 10 shows the specializations built for Java. The *JavaClass* type includes a field call *isInterface*.

4. The reengineering process as a MDA process

According to [9], Model-Driven Architecture (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".

MDA suggests the use of three viewpoints in software systems, establishing that the construction of one of them can be seen as a series of successive transformations from one to another viewpoint, or from one to another of their corresponding models. The proposed viewpoints are the computation-independent,

the platform-independent and the platform-specific viewpoint.

- Our tool and the supported reengineering process
- Both the *Database* and the *OOS* metamodels represent relational databases and class diagrams in a vendor-independent way. With these metamodels, the point of view of the described reengineering process) of any relational database or class diagram. Thus, these metamodels match with the computation-independent viewpoint, and *OOS* represent actual databases and class diagrams. As a matter of fact, all the factories appearing in Figure 2 translate their respective physical databases to our specific metamodel; later, the obtained instance of *OOS* must be translated into a specialization of these metamodels match the platform-independent viewpoint.
- Finally, the resulting database and the multilayer required to build the new version of the database are quite similar, but they have small differences from one to another RDBMS. Also the JSP pages, classes, etc. are different when the application is based on standard Java classes or Enterprise Java

5. Conclusions and future work

This paper has presented some characteristics of and from relational databases by means of a complete process of reengineering. The successive transformations are based on formal description

Based on an excellent architectural design, the tool can generate four different types of applications from four different types of databases. Its design also facilitates both the addition of new database management systems to be used as input products and The tool has been used in the development of at least 12 projects. The uniform structure of the code generated and the adequate location of responsibilities allow the developers to decrease the learning curve of the generated applications, achieving important savings in the maintenance of the

standard Java desktop application. In this case, both applications share the domain and persistence tiers, The mean time of the resolution of corrective and perfective interventions requires less than 1.5 hours.

Currently, we are implementing a generator to obtain .NET web applications on the development of several components to save all models in XMI standard formats, which will make it possible to manipulate them using other tools.

description techniques with model-driven architecture is an excellent combination to improve the automation of software processes.

This work is partially supported by the MÁS project (Mantenimiento Ágil del Software), Ministerio de

7. References

[2] Biggerstaff, B.G. y D.E. Mitbander, *Program understanding and the concept assignment problem.* Communications of the ACM, 1994. **37**(5). pp. 72-83.

[3] Brown, K. y B.G. Whitenack, *Crossing Chasms, A Pattern Language for Object-RDBMS Integration.* K.S.

[4] Broy, M., *Towards a Mathematical Foundation of Software Engineering Methods.* IEEE Transactions on Software Engineering, 2001. **27**(1). pp. 42-57.

[5] Buschman, F., et al... *A System of Patterns.*

[6] Chiang, R., T. Barron, y V.C. Storey, *Reverse engineering of relational databases: extracting of an EER model from a relational database.* Journal of Data and Knowledge Engineering, 1994. **12**(2). pp. 107-142.

[7] Gamma, E., R. Helm, J. Johnson, y J. Vlissides, *Design Patterns.*

[8] Larman, C., *Applying UML and Patterns.* 1998, New York: Prentice Hall, Upper Saddle River.

[9] OMG, *MDA Guide Version 1.0.1.* 2003, Object Management Group. p. 62.

[10] Pedro de Jesus, L. y P. Sousa. *Selection of Reverse three-tier applications from relational databases: a formal*

and practical approach. Information & Software Technology, 2002. 44(15). pp. 923-941.

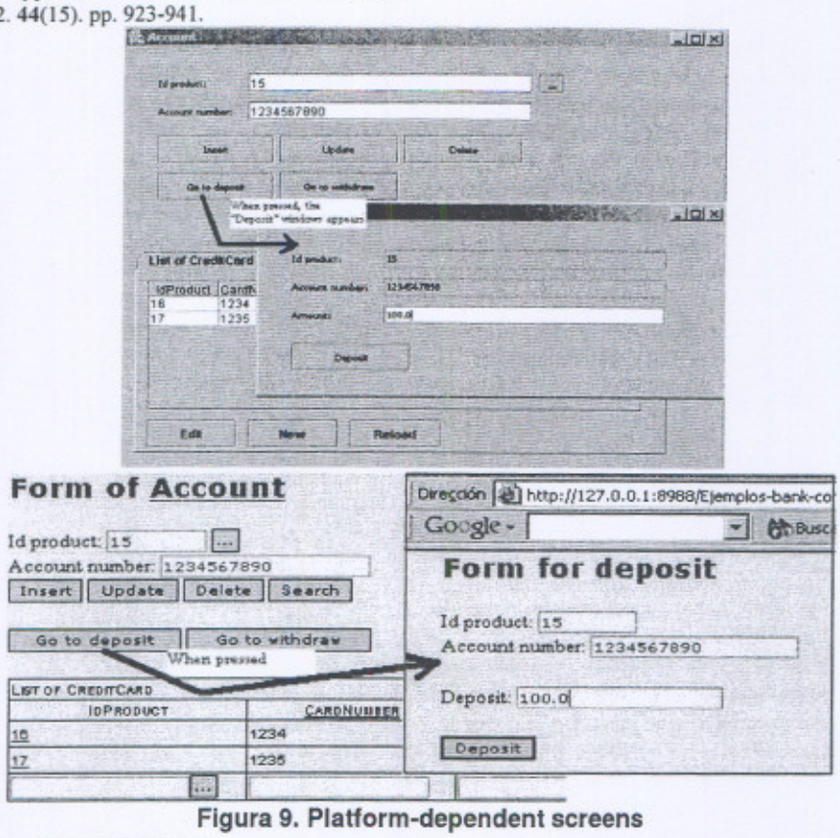


Figura 9. Platform-dependent screens

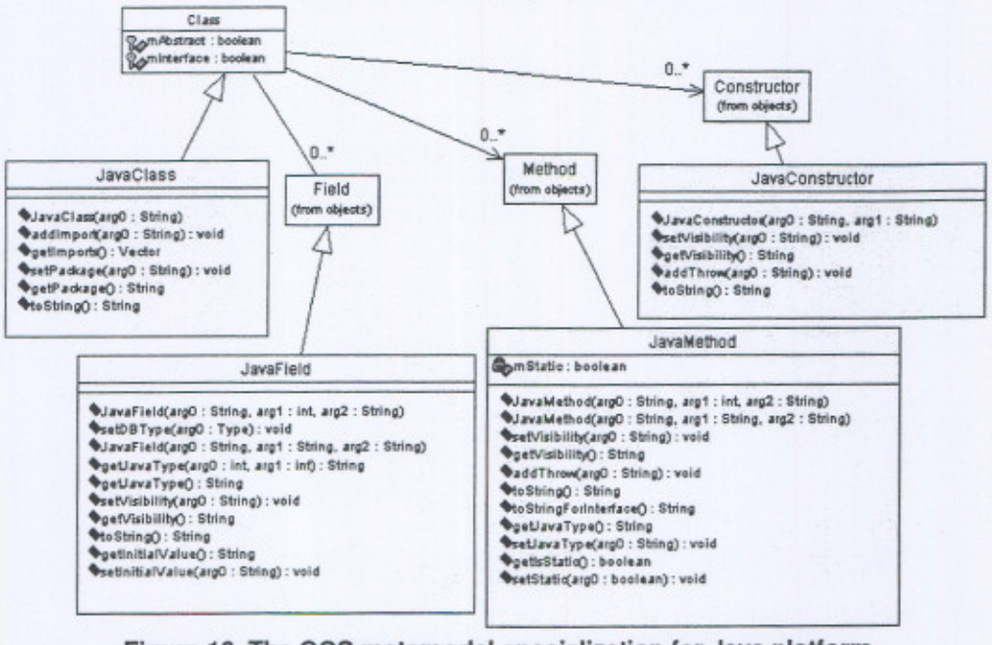


Figura 10. The OOS metamodel specialization for Java platform