QAOOSE 2006 Proceedings

10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering

3 July 2006 — Nantes, France

Edited by:

Michele Lanza, Fernando Brito e Abreu, Coral Calero, Yann-Gaël Guéhéneuc, Houari Sahraoui

Lugano
Universitá della Svizzera italiana
2006

Organizers

Fernando Brito e Abreu, *Univ. of Lisbon, Portugal*

Coral Calero, *Univ. of Castilla, Spain*

Yann-Gaël Guéhéneuc, *Univ. of Montreal, Canada*

Michele Lanza, *Univ. of Lugano, Switzerland*

Houari Sahraoui, *Univ. of Montreal, Canada*

Outline

QAOOSE 2006 is a direct continuation of nine successful workshops, held during previous editions of ECOOP in Glasgow (2005), Oslo (2004), Darmstadt (2003), Malaga (2002), Budapest (2001), Cannes (2000), Lisbon (1999), Brussels (1998) and Aarhus (1995).

The QAOOSE series of workshops has attracted participants from both academia and industry that are involved/interested in the application of quantitative methods in object-oriented software engineering research and practice. Quantitative approaches in the object-oriented field is a broad and active research area that develops andor evaluates methods, practical guidelines, techniques, and tools to improve the quality of software products and the efficiency and effectiveness of software processes. The workshop is open to other technologies related to object-oriented such as component-based systems, web-based systems, and agent-based systems.

This workshop provides a forum to discuss the current state of the art and the practice in the field of quantitative approaches in the fields related to object-orientation. A blend of researchers and practitioners from industry and academia is expected to share recent advances in the field-success or failure stories, lessons learnedand seek to identify new fundamental problems arising in the field.

ISBN 88-6101-000-8

Contents

Metrics, Components, Aspects

Visualization, Evolution

Quality Models, Metrics, Detection, Refactoring

A maintainability analysis of the code produced by an EJBs automatic generator

Ignacio García-Rodríguez de Guzmán, Macario Polo, Mario Piattini

ALARCOS Research Group
Information Systems and Technologies Departament
UCLM-Soluziona Research and Development Institute
University of Castilla-La Mancha
Paseo de la Universidad, 4 – 13071 Ciudad Real, Spain
{Ignacio.GRodriguez, Macario.Polo, Mario.Piattini}@uclm.es

Abstract. Design and development of Web applications is an increasinglydemanded topic. However, successive changes to their code and databases result in a progressive decreasing of its quality and maintainability. Because of that, we have built a tool for the automatic generation of multilayer web components-based applications to manage databases. The source code of these applications is automatically generated, being this one optimized, corrected and already pre-tested and standardized according to a set of code templates. This paper makes an overview of the code generation process and, then, shows some quantitative analysis related to the obtained code, that are useful to evaluate its maintainability. This study is important for developers since they will probably require to implement some changes for its adaptation to the final requirements.

1. Introduction

Reengineering is one of the most powerful tools offered by software engineering to maintain legacy systems (**Fig. 1**). According to [1], reengineering is composed in turn by other two techniques, the "forward" and the "reverse engineering"

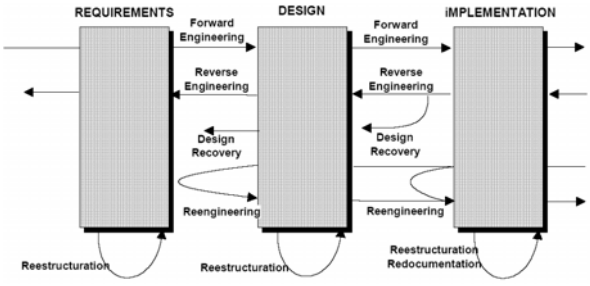

Fig. 1. Simplified reengineering model

Reverse engineering is the process of building abstract formal specifications from source code of a legacy system that can be later used to build new versions of the system, but now, using forward engineering [1].

In this context, we have developed a tool which generates Web applications from a relational database applying complete reengineering process. These Web applications are generated automatically, and support the management of a relational database. According to [2-5], the most usual practice when reverse engineering is applied to databases is to obtain an entity-relation scheme, although, other proposals get an object oriented representation from the database, usually as a class diagram [6, 7]. The use of class diagrams instead of ER schemas provides, from a reengineering point of view, the possibility of taking advantage of the object oriented paradigm constructions for the later steps.

Because of their nature, web applications have a complex development process, especially when a *middleware* must support the management of the database, and security of transaction constraints must be taken into account. *Enterprise JavaBeans* is a technology specifically designed for dealing whit this problems, but these characteristics (such as indirect relationship among classes and interfaces, that are managed by component containers) make difficult its development and maintenance.

Our proposal is based on a tool which automatically generates distributed component-based applications (specifically EJB components and Web Services, both written in Java), using some principles of software engineering inside them. Some of these principles are the use of design patterns (which provide great consistency, extensibility and understandability to the application). As a result, applications can be easily extended adding new features which implement new. Furthermore, some technical documentation is generated when the web application is generated. This documentation helps us in the afore-mentioned maintenance process, making easier the modification of the source code. In addition, automatic development of these kind of web applications lets to the development team to save a lot of time. In order to analyze the maintainability of the generated code, in this paper we make a quantitative analysis of the generated source code by means of the use of some object-oriented metrics. A quantitative way, an overview of such easy is to maintain these applications.

This paper is organized as follows: Section 2 contains an overview of some related technologies and metrics; in Section 3, some metrics are applied to an example web application obtained from a relational database by our tool. Result are shown and commented in the same section; finally, we draw our conclusions and future lines of work in Section 4.

2. Web Application technologies and Metrics for source code evaluation

From a relational database, our tool generates a multilayer application [8] based on EJB components and JSP pages. **Fig. 2** shows the general architecture of the automatically web applications.

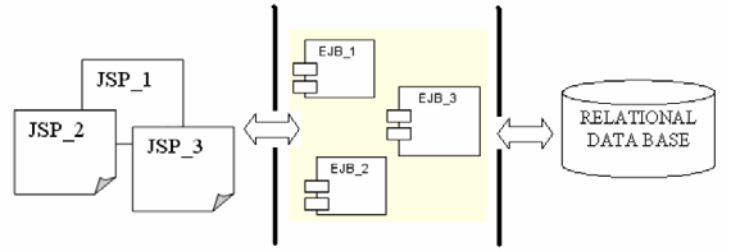

Fig. 2. Basic architecture of the automatic generated applications

In **Fig. 2** we can distinguish a layer made up of JSP pages whose goal is to offer a friendly interface to the user in order to manage the database. The middle layer is a middleware composed by EJB components which implements the logic to perform the management of the database. The third layer is made up of the relational database and, maybe, some additional classes. We do not provide an analysis of neither the tool nor of the generated web application from since this points are out of the scope of this paper.

The most important elements of the generated web applications are the EJB (*Enterprise Java Bean*) components, which are components written in Java language. An EJB component has a couple of interfaces, a class which implements the methods (of the interfaces and others) and a set of additional classes which gives support for some features that could be necessary implement. Actually exits three types of EJB components (*Entity Beans*, *Message Driven Bean* and *Session Bean*. For our proposal, the most interesting EJB type is the *Entity Bean*, because this one referents a persistent entity existing in the relational database.

As we said in the beginning, these applications carry a substantial complexity, because there are some technologies involved in the development process in order to implement all the features, and also, to delegate the database management to the component-based *middleware* requires an additional effort. This is due to the fact that we have to program the necessary logic to orchestrate all the components in such way that the database integrity be respected. That corresponds to define the choreography among the EJB components.

After studying the problem, we notice that the development process of such applications could be performed in an automatically way, because the generated source code could be predicted. The preliminary analysis let us to generate free-error code, and as far as possible, this code is already optimized by means of the use of design patterns. In this manner, we obtain the basic number of classes, with the basic number of methods per class for each component, being written both classes and methods in a clearly and concise. This allows the possibility of realize task of adaptive and perfective maintenance in the future, when new features and requirements have to be added to the web applications in order extend the offered services.

To check these assertions, we will use some well-known software metrics to verify the quality of the source code of the generated applications. The used metrics are the following:

- LOC (*Lines Of Code*): This metric is the sum of lines of the source code of the
- class.
- WMC (*Weighted Methods Class*): This metric is the sum of the complexities of methods of a class, this is, the sum of the ciclomatic complexities.
- CBO (*Coupling Between Objects Classes*): This metric measures coupling among classes.

According to several studies, high coupling is the best predictor of the fault proneness of classes [9]. When the coupling or complexity understandability and testability of the system decreases, and any attempt of change something in maintenance task will be hard and difficult. So, these metrics are good predictors of the quality of our generated Web applications.

A database example (see **Fig. 3**) has been designed to illustrate the results of applying these metrics to the obtained source code. The database schema is very simple but enough for our illustrative goal.

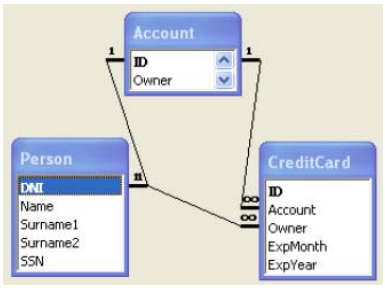

Fig. 3. A simple database

Once we have the database schema, the last step is generating the source code of the Web application. The following sections deal with the measure of this generated source code.

3. Source code quality measure

With out tool, an EJB is generated for each table. **Fig. 4** shows the signature of the operations generated for the *CreditCard* table (**Fig. 3**).

Next sections concrete present the calculus of the values of these metrics for these EJBs; Section 3.4 includes the description of the equations for predicting their values from the database schema.

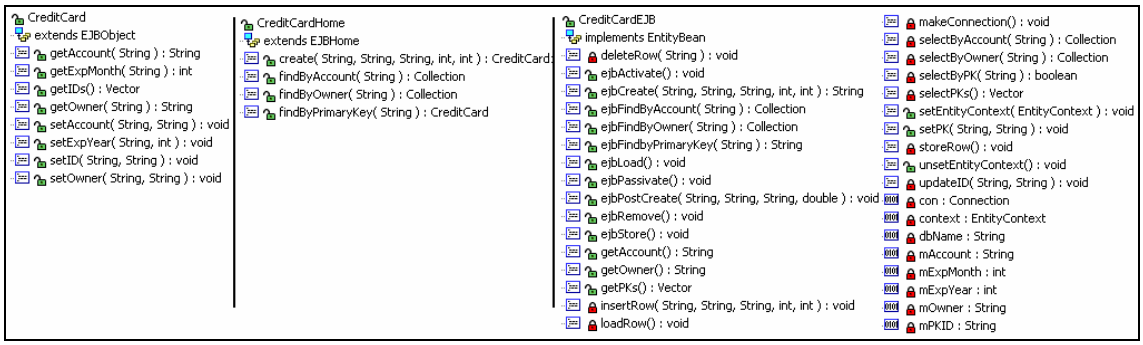

Fig. 4. Classes and interfaces automatically generated from the table CreditCard

3.1. Lines of Code

This metric measures the total number of lines ended with a semicolon in classes and interfaces. Below, we show the results for each element of each *Enterprise Java Bean* generated from the original data base:

Account (LOC)		
Account.java	AccountHome.java	AccountEJB.java
12	7	114
CreditCard (LOC)		
CreditCard.java	CreditCardHome.java	CreditCardEJB.java
12	8	151
Person (LOC)		
Person.java	PersonHome.java	PersonEJB.java
12	7	125

The number of lines of code generated depends on the schema of the database, the number of columns and tables, foreign keys, indexes and stored procedures. Also the number of LOC source code generated is very predictable, because lines of code generated from a table are directly proportional to the elements related with it.

3.2. Coupling between objects classes

The high coupling is a non-desirable characteristic in an OO system that can be measured using the *Coupling Between Object Classes* metric (CBO). CBO is a count of the number of classes a class is coupled to. It is measured by counting the number of related class hierarchies on which a class depends [10].

Inside the source code generated by our tool, coupling depends directly on the scheme of the database too. So coupling is directly proportional to the number of foreign keys existing among tables. For example, if there is a table in the database with three foreign keys to other tables, the EJB which represents this table will be

related with the other three EJBs epresenting the tables whose primary keys are foreign keys in the first table.

For this reason, the coupling measured here will be the existing coupling among components, not among classes, because coupling among classes automatically generated will be a constant. Other thing is the coupling caused by an external developer that modifies the source code in order to add some functionality or new features to the generated application. Because this relation among components depends on the number of foreign keys in the tables of the database, the level of coupling of the system will be also predictable.

```
package mypackage2;

import java.util.Collection;
import java.rmi.RemoteException;
import javax.ejb.*;

public interface CreditCardHome extends EJBHome
{

  public CreditCard create(String ID, String Account, String Owner, int ExpMonth, int Year)
    throws RemoteException, CreateException;

  public CreditCard findByPrimaryKey(String ID)
    throws FinderException, RemoteException;

  public Collection findByAccount(String Account)
    throws FinderException, RemoteException;

  public Collection findByOwner(String Owner)
    throws FinderException, RemoteException;

}
```

Fig. 5. CreditCardHome Interface with its 8 lines of code

Continuing with our example, the coupling from the component point of view is represented in **Fig. 6**. As we can see, the *CreditCard EJB* depends of the *Account* and the *Person* EJBs. This figure can be compared with **Fig. 3**, where we can clearly see the foreign keys.

According to [10], coupling between objects should not be greater than 5 since higher CBO decreases system understandability, avoids the reuse of components and makes more costly maintenance. Our tool keeps the coupling between classes and components at the minimum level.

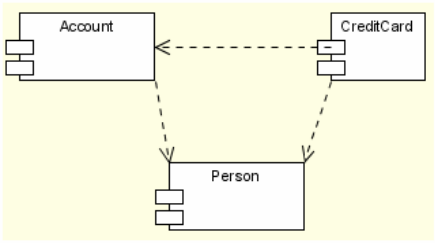

Fig. 6. Coupling between EJBs

3.3. Weighted Methods per Class

The last metric applied to the generated source code by our tool from a relational database is the *Weighted Methods per Class* (WMC). This metric is very similar to the *McCabe Ciclomatic Complexity* [11].

As Ciclomatic Complexity, [11] WMC gives the minimum number of test cases for a given system, supposing each decision condition as a different decision node; when the complexity is greater than 10, the probability of find faults in code grows, and so, we should raise again the architecture of the module which obtains this punctuation.

According to [10], WMC, must be lower than 100, so a class must have at most twenty methods per class and the ciclomatic complexity per method must be lees than 5. WMC is given by the following expression:

$$\sum_{i \in classes} \sum_{j \in Methods_i} Ciclomatic\ Complexity(j)$$

In our small example, WMC for each class and for all the components are the following:

Account (WMC)		
Account.java	AccountHome.java	AccountEJB.java
8	3	41
TOTAL: 52		
CreditCard (WMC)		
CreditCard.java	CreditCardHome.java	CreditCardEJB.java
8	4	44
TOTAL: 56		
Person (WMC)		
Person.java	PersonHome.java	PersonEJB.java
8	3	40
TOTAL: 51		

As we can notice see, none of the EJB in the example overcomes the limit imposed by [10]. In case, the code generated is fault-free.

In the case that other developers add some code generated by themselves, complexity of web applications could be increased depending on the ability of these developers, although to follow the code and design styles our tool adds some technical documentation in addition to the generated code, and so, developers can notice the design styles and follow them.

```
private void loadRow() throws SQLException
{
  String selectStatement="SELECT Account, Owner, ExpMonth, ExpYear "+
                         "FROM CreditCard WHERE ID = ?;";
  PreparedStatement prepStmt=con.prepareStatement(selectStatement);

  prepStmt.setString(1, mPKID);
  ResultSet r=prepStmt.executeQuery();

  if (r.next())
  {
    mAccount = r.getString(1);
    mOwner = r.getString(2);
    mExpMonth = r.getInt(3);
    mExpYear = r.getInt(4);
    prepStmt.close();
  } else {
      prepStmt.close();
      throw new NoSuchEntityException ("Row with ID:" + mPKID + " not found in database");
 }
}
```

Fig. 7. loadRow method with a 2 ciclomatic complexity level

3.4. Equations to predict metrics (and its maintainability)

Finally, in sight of the result afore-obtained and the source code generated, we have derived some equations. These equations allow to predict some characteristics of the web applications generated from a database.

To predict the *Number of Lines of Code* (LOC) for the EJB components, we can apply the following equation:

$$LOC = K_{LOC} + 13 * N^{o} OfIndexes + 8 * N^{o} Col + 3 * N^{o} ColFK \quad (1)$$

Where K_{LOC} is a constant representing the minimum lines to be always generated and its value is 90; *NºOfIndexes* is the number of indexes in the table associated to the EJB; *NºCol* is the number of columns in the table and *NºColFK* is the number of columns of the table which are foreign keys.

Coupling between objects (CBO), for a given EJB, can be predicted from the table by means of the following equation:

$$CBO_{EJB} = \sum_{i=0}^{FKs} N^{o} Cols(FK_i) \quad (2)$$

Where *FKs* is the set of foreign keys of the table represented by the EJB, *FK* is the foreign key that is being examined, *NºCols()* is a function that obtains the number of columns that targets to different tables inside de same foreign key. Note that a consequence to take in account when we realize this operation is that if columns belonging to the current foreign key are targeting to the same table, functions returns one.

To estimate the *Weighted Methods per Class*, we have obtained other equation:

$$WMC = K_{WMC} + 4 * N^{o} Col + 2 * N^{o} ColFK \quad (3)$$

Where K_{WMC} is a constant representing the minimum ciclomatic complexity to be always generated and its value is 20, *NºCol* is the number of columns of the table associated to the EJB, and *NºColFK* is the number of foreign key columns.

Also, if there is stored procedures in the database, an additional EJB is generated containing methods to call them. In this case, this EJB is not an *Entity Bean* but a *Session Bean*. As well as an *Entity Bean* materializes a record from a table, a *Session Bean* only interacts with the client. For our purpose, the *Session Bean* will allow us to invoke the stored procedures of the database. In order to estimate the effect caused to the calculated metrics, we derive two very simple expressions which give us a measure of LOC and WMC (coupling is not affected). The estimated metrics for the *Session Bean* representing the stored procedures are:

$$LOC = K_{LOC} + 12 * N^{o} StorProc \quad (4)$$

$$WMC = K_{WMC} + 3 * N^{o} StorProc \quad (5)$$

In the *LOC* equation, K_{LOC} is the minimum number of lines always included in the bean, and *NºStorProc* is the number of stored procedures the database In the *WMC* equation, K_{WMC} is a constant which value is 7, and *NºStorProc* is the number of stored procedures in the database. For the stored procedures owned by the system, the tool does not generate code.

As it is seen, the design of the database has a strong influence on the quality of the application that manages it. Using the thresholds proposed by NASA [10] together to equations 1-6 (as predictors of the quality of the application), it is possible to determine, before the application development, that a change in database design is required in order to keep adequate values of maintainability and fault proneness in the application.

4. Conclusions and future work

Development of component-based web applications constitutes a complex process which involve some technologies. For this reason, a tool has been developed in order to automate this process. The fact of generating correct web applications is so important that writing of optimized, easily understandable and documented source code.

The tool presented, give us a very simple method to develop web applications to support the management of a relational database. This management is realized by means of a set of EJB components which constitutes the middleware that implements

all the necessary logic. As the generated application must probably be modified to adapt it to the actual requirements, we have studied the quality of the generated source code from the maintainability point of view. Thus, we have analysed some features of the code as predictors of maintainability. As our prediction method has demonstrated, the developed tool generates code which is easily to maintain and understand.

Other lines of work could consist in develop other techniques which optimize more the source code obtained, reducing the number of EJB components in the systems. Some of these techniques could be the implementation of any heuristic to optimize the number of tables represented by an EJB, or the choreography defined to coordinate the operations of the EJB during the management of the relational database.

5. Acknowledgements

This work is partially supported by the MÁS project (Mantenimiento Ágil del Software), Ministerio de Ciencia y Tecnología/FEDER, TIC2003-02737-C02-02, and the ENIGMAS project, Plan Regional de Investigación Científica, Desarrollo Tecnológico e Innovación, Junta de Comunidades de Castilla La Mancha, PBI-05-058

References

1. Arnold, R.S., *Software Reengineering*, ed. 0-8186-3272-0. 1992: IEEE Press. pp. 675.
2. Andersson, M. *Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering.* in *13th International Conference on Entity-Relationalship Approach*. 1994. Berlin: Loucopolous.
3. Pedro de Jesus, L. and P. Sousa. *Selection of Reverse Engineering Methods for Relational Dabases.* in *Proceedings of the Third European Conference on Software Maintenance*. 1998. Los Alamitos, California: Nesi, Verhoef.
4. Chiang, R., T. Barron, and 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)): p. pp. 107-142.
5. Hainaut, J.-L., et al. *Database Design Recovery*. in *Eighth Conferences on Advance Information Systems Engineering*. 1996. Berlin.
6. Polo, M., et al., *Generating three-tier applications from relational databases: a formal and practical approach.* Information & Software Technology, 2002. **44**(15): p. pp. 923-941.
7. García-Rodríguez de Guzmán, I., M. Polo, and M. Piattini. *An Integrated Environment for Reengineering*. in *Proceedings of the 21st International Conference on Software Maintenance (ICSM 2005)*. 2005. Hungary, Budapest: IEEE Computer Society.
8. Larman, C., *Applying UML and Patterns*. 1998, New York: Prentice Hall, Upper Saddle River.
9. Briand, L., J. Wuest, and H. Lounis. *Using Coupling Measurement for Impact Analysis in Object-Oriented System*. in *IEEE International Conference on Software Maintenance (ICSM´99)*. 1999. Oxford.
10. Rosenberg, L., R. Stapko, and A. Gallo, *Applying Object Oriented Metrics*. 1999, NASA.
11. Piattini, M.G., et al., *Análisis y diseño de Aplicaciones Informáticas de Gestión: Una perspectiva de Ingeniería del Software*. 2004, Madrid: RA-MA. 710.