International Journal of Soft Computing 7 (6): 264-270, 2012
ISSN: 1816-9503

Applying Game Theory to Restructure PL/SQL Code

[1]S. Vimala, [1]H. Khanna Nehemiah, [1]R.S. Bhuvaneswaran, [1]G. Saranya and [2]A. Kannan
[1]Ramanujan Computing Centre, [2]Department of Information Science and Technology,
Anna University, 600 025 Chennai, India

Abstract: Success of any organization is based on the quality of the information system which undergoes many alterations during its life cycle. Hence, it can be quoted as an example for a live entity. Database is a core component in any information system and it gets affected due to change in business logic. A popularly used data model in any organization is the relational model. Changes that are made in relational schema also modify the queries that access the relations. It is really difficult to identify the set of queries that access the same relation in the case of large information system. Similarity measures concept is one of the techniques that are applied in object oriented programming for refactoring is used in this proposed system for restructuring procedures into packages by taking PL/SQL code as input. The proposed system groups those queries or procedures that access the same relations into a single package. The objective of this research is to determine whether the proposed methodology can be used as a mechanism to improve the maintainability of PL/SQL code. This process of packaging is done by applying game theory so as to increase understandability and maintainability of the system.

Key words: Refactoring, restructuring, game theory, packaging, pay-off matrix, Nash equilibrium, PL/SQL, structural similarity

INTRODUCTION

In software engineering discipline and practice, the modules and their relationships are established in the architectural design before coding takes place. Modularization of code is analogous to data normalization which gives the benefits of reusability, manageability, readability and reliability. In order to adapt to the constantly evolving requirements, software designers and developers add new features or modify the existing design. Because of this, the software becomes more and more complex and drift away from its original design thereby the quality of the software is getting reduced and hence effort on maintenance is increased. A huge portion of the total software development cost is normally spent on the software maintenance (Coleman *et al*., 1994; Guimaraes, 1983; Mens and Tourwe, 2004). To ease the maintenance, the existing code is to be restructured so that the changes made in a module will not create any adverse effect in any other module which brings down the ripple effect. This reduces the software complexity by improving the internal software quality. In the research domain this process is referred to as restructuring (Arnold, 1986; Griswold and Notkin, 1993; Mens and Tourwe, 2004) or refactoring as in the special case of object oriented software development (Opdyke, 1992; Fowler, 1999; Mens and Tourwe, 2004).

The term refactoring was originally introduced by Opdyke in his Ph.D dissertation (Opdyke, 1992). Refactoring is the process of changing an object-oriented software system in such a way that it does not alter the external behaviour of the code yet improves its internal behaviour (Fowler, 1999). The key idea behind refactoring or restructuring is to encourage code reuse. Clean, modular, well written code is easy to reuse, reducing future programming efforts. Furthermore, refactoring aims to improve several factors of quality namely, understandability, portability, maintainability, testability, reliability, usability, reusability and adaptability.

In this context, restructuring is needed to convert legacy code or deteriorated code into a more modular or structured form. In order to obtain a system consisting of relatively independent modules in the legacy system, it gets remodularized or reengineered. Restructuring and refactoring are used in the context of software evolution and reengineering. It also makes developers to develop program faster and assists in identifying defects (Fowler, 1999). Identifying the restructuring opportunities is a difficult task since when and where to apply restructuring is based on human intuition. But some approaches in the literature have stated restructuring methodologies for object oriented programs, one such is to identify the refactoring opportunities by the presence of some bad smells like long method, god classes and feature envy in the code.

Corresponding Author: H. Khanna Nehemiah, Ramanujan Computing Centre, Anna University, 600 025 Chennai, India

In the proposed approach the restructuring of PL/SQL code is performed by the structural similarity measure that captures the relationship between procedures accessed by same attribute and relation respectively. The results are compared and conclusions are drawn. The PL/SQL code is restructured by grouping the procedure that accesses the same relation/attribute into one package by applying game theory to adapt a suitable strategy for the competing objectives which leads to better maintainability and understandability.

LITERATURE REVIEW

Over the last years several approaches have been proposed for code refactoring in order to improve the overall quality the software systems. Fowler (1999) made an invaluable contribution to object oriented software development by shedding lights on the refactoring process. They explained the principles and best practices of refactoring. They also stated when to apply and not to apply refactoring to the source code but failed to state where to apply refactoring and complicated this by stating that refactoring are based on human intuition and on subjective perceptions. They further stated that no set of metrics rivals informed human intuitions. According to them the main risk of refactoring is that existing working code may break due to the changes being made. They concluded by suggesting two golden rules namely, refactor in small steps and have test scripts available to existing functionality to mitigate this risk.

Simon *et al.* (2001) in their research proposed a metric-based visualization tool to support the identification of source code components that needs refactoring. They defined a distance-based cohesion metric which measures the cohesion between attributes and methods. The calculated distance are visualized in a three dimensional perspective. They showed that these special kinds of metrics can support the subjective perceptions and thus can be used to get support for the decision what technique of refactoring is to be applied and where it can be deployed in an effective and efficient way. In their research they demonstrated the concept for four usual refactoring namely move class, move attribute, extract class and inline class by presenting a tool supporting the identification and explaining the real world application with this concept. The drawback of Simon's approach is that the calculated distances are visualized in a three-dimensional perspective, forcing the developer to manually identify refactoring opportunities. So, the visual interpretation of distance in large systems can be a difficult and subjective task. Furthermore, the approach proposed by Simon *et al.* (2001) did not suggest any restructuring automatically; it just helps the developer to identify the restructuring opportunities.

Joshi and Joshi (2009) presented a method based on concept analysis which aims to identify less cohesive classes. Concept analysis for class cohesion has come up with the concept called Cohesion Lattices that captures the cohesiveness of the class and its members. Most metric based approaches for cohesion analysis compute a single value for the entire class and do not help to identify individual members responsible for the lack of cohesion. But the approach of Joshi and Joshi (2009) helps to identify the class members contributing to lack of cohesion. The concept based cohesion analysis gives a quick visual overview of class cohesion and helps guiding refactoring in localization of attributes, removal of instant variables and class extraction.

Fokaefs *et al.* (2009) proposed a methodology which identifies extract class refactoring opportunities by a class decomposition method. An agglomerative clustering algorithm is used based on the Jaccard distance between class members. In terms of cohesion this research facilitates to identify new concepts and rank the solutions according to their impact on the design quality of the system. Specific kind of bad smells called God class are considered for this purpose. Data God class and Behavioural God class are defined. A class which has many system's data in terms of number of attributes is called Data God class. When it has greater portion of the systems functionality in terms of number and complexity of methods is called Behavioural God class. Behavioural God class may be avoided by splitting the class by extracting a cohesive and independent piece of functionality. This refactoring is called extract class. Two projects namely eRisk, an electronic adaptation of the well known board game and Selfplanner (Refanidis and Alexiadis, 2008), an intelligent web based calendar application have been taken for this purpose and the result shows that this methodology identifies relatively large number of new concepts. To improve the understandability of the code and the process of maintenance many number of refactoring technique is applied in this research.

To perform clustering with different attribute weights Alkhalid *et al.* (2010) proposed Adaptive K-Nearest Neighbor (A-KNN) algorithm to facilitate the software developers in refactoring at the function or method level. Improvements of cohesion for ill-structured entities are suggested by them. Entities are code statements that needs to be grouped. There are two types: executable and non-executable statements. Comments and declarations that do not affect the functionality of the function;

therefore they are not used as entities are non-executable statements. The statements which include assignment, operation and iteration statements are the executable statements. This A-KNN is compared with three other function-level clustering techniques like Single Linkage algorithm (SLINK), Complete Linkage algorithm (CLINK) and Weighted Pair-Group Method using Arithmetic averages (WPGMA) to prove the improved performance of proposed system. Each entity has properties, based on which the similarity matrix is constructed. A-KNN reduces the amount of required computations by using this similarity matrix. Code restructuring at class and package levels are not discussed in this research.

Du Bois *et al.* (2004) proposed a concept of refactoring with metric oriented approach which helps improve metrics rather than apply them as an identification criterion. The best and worst-case impact of refactoring on coupling and cohesion dimensions are theoretically analyzed. The refactoring they studied are Extract Method, Move Method, Replace Method with Method Object, Replace Data Value with Object Method and Extract Class Method. They demonstrated that by exploiting the results from coupling or cohesion impact analysis, it is possible to achieve quality improvements with restricted refactoring efforts. This effort is restricted to the analysis and resolution of a limited set of refactoring opportunities which are known to improve the associated quality attributes.

Bavota *et al.* (2011) have proposed a methodology for identification of extract class refactoring opportunities using structural and semantic cohesion measures. They have used Maxflow-Mincut algorithm to suggest Extract class refactoring. The approach takes as input a low cohesive class and parses the input class in order to extract a weighted graph representation of the class. The methods of the class form the nodes of the graph and the edges of the graph are the interaction between those pair of methods connected by the edge. Maxflow-Mincut algorithm is used to obtain the partition of the original graph in two subset of nodes. Such a partition is obtained by cutting the minimum number of edges with low weight. Without increasing too much coupling to build two new classes that have cohesion higher than the original class the two sub-graphs can be used. They have applied their methodology on seven less cohesive classes of GanttProject System. The drawback of this approach is that they have used semantic measure to perform refactoring and this semantic measure will work only if the system has proper commands, meaningful identifiers and method name.

Bavota *et al.* (2010a, b) have proposed an approach using game theory to identify extract-class refactoring opportunities. It is very common in software engineering to find a solution often for problems having competing goals, like integrity versus efficiency, testability versus efficiency, reusability versus reliability, reusability versus integrity, quality versus cost, cohesion versus coupling by the developers and managers. Game theory techniques can often be used to deal contrasting goals. Indeed, game theory is successfully used to propose solutions to strategic situations in which an individual's success in making choices depends on the choices of others. Bavota *et al.* (2010a, b) applied game theory in software engineering. The results achieved in a preliminary evaluation, supported the applicability and superiority of game theory in refactoring. Other software engineering problems having competing goals may also be solved using game theory.

The works discussed in the literature deals with various refactoring techniques used for object oriented systems. In the proposed research, the concept of refactoring applied to object oriented systems is extended to PL/SQL code which there by improves the understandability and maintainability of PL/SQL code. To achieve this, extract procedure refactoring opportunitiesproposed by Bavota *et al.* (2010a) based on game theory is used.

Existing literature do not suggest any restructuring method that can be directly applied for PL/SQL code to the best of the knowledge.

PROPOSED SYSTEM

The proposed approach extends the concept of refactoring applied by Bavota *et al.* (2010a, b) in object oriented domain to PL/SQL code where by maintainability and understandability is made easy. To achieve this, refactoring is performed by using game theory.

Restructuring of PL/SQL code is guided by the structural similarity between methods which measures the relationship between procedures accessing the same relations. The PL/SQL code is restructured by grouping the procedures that access the same relations into one package by applying game theory. Traditional applications of game theory focusing on non-cooperative games attempt to find equilibria in such games. Nash equilibrium (Nash, 1951) which is the most famous equilibrium provides the solution to any game where two or more players are involved. Nash equilibrium is the state in which every player is playing a best move based on the strategic choices of the opponents. It is not at all possible for a player to do better by choosing a different move. The role of each player in the research is to build a package of high cohesion and low coupling. The

constructions of the packages are iterative. For each iteration a player can select atmost one procedure of the package to be refactored. At the first iteration, the pair of procedures with lowest similarity is assigned to the two players. In successive iteration each player selects a procedure that is highly cohesive with the procedure it has selected in the previous iteration. If adding a procedure to the package build by a player decreases the cohesion of the package then the player on that iteration adds no procedure to the package and this action of the player is termed as null move. So, in particular during an iteration of the process, a player can perform one of the following moves: in the first move a player selects the procedure mi and yield the procedure mj to the opponent player (i, j move) in the second move the player selects the procedure m_i while the opponent does not select any procedure (i, null move) and in the third move the player does not select any procedure while the opponent selects the procedure m_j (null, j move).

The move to be performed during an iteration of the process is chosen by finding the Nash equilibrium in the payoff matrix. The Nash equilibria are obtained using the algorithm defined by Mangasarian (1964) which enumerates all the equilibria in a two-player finite game. If there are more than one Nash equilibria then the one having the highest sum of the payoffs of both players are selected. The process stops when the each procedure of the packages is assigned to any one of the two players. The computation of Similarity Matrix and Pay-off Matrix are given.

Process 1: Computation of the Similarity Matrix.

Input: PL/SQL code in text format.

Process logic: The Measure Structural Similarity between Methods (SSM) proposed by Gui and Scott (2006) is used to compute the Similarity Matrix. The measure is computed as follows:

$$SSM(p_i, p_j) = \begin{cases} \dfrac{|A_i \cap A_j|}{|A_i \cup A_j|} & \text{if } |A_i \cup A_j| \neq 0 \\ 0 & \text{Otherwise} \end{cases} \quad (1)$$

In the above measure A_i, A_j is the set of relations referenced by procedure p_i, p_j where i and j = {1, 2, ...,.n}. The value of SSM is in the range 0-1. A Similarity Matrix of n×n is constructed by substituting each SSM values in the elements of the matrix. Where n is the number of procedures of the package to be refactored.

Output: Similarity Matrix.

Process 2: Computation of Pay-off Matrix.

Input: Similarity Matrix (Output of the Process 1).

Process logic: The pay-off matrix is computed using the mathematical model (Nash, 1951) is as follows:

$$P(i,j) = \begin{cases} \begin{pmatrix} fcc(P_a, p_i) - fcc(P_a, p_j), \\ fcc(P_b, p_j) - fcc(P_b, p_i) \end{pmatrix} & \text{if } i, j \neq \text{null} \\ (fcc(P_a, P_i), \mu - fcc(P_b, P_i)) & \text{if } i \neq \text{null}, j = \text{null} \\ (\mu - fcc(P_a, P_j), fcc(P_b, P_j)) & \text{if } i = \text{null}, j \neq \text{null} \\ (-1, -1) & \text{if } i = j \end{cases} \quad (2)$$

where, $\mu > 0$ is a configuration parameter used to balance the payoff of null move with respect to other payoffs. P_a and P_b are the packages that are being built by the player A and player B respectively, p_i and p_j are the pair of procedures of the original package P, i and j are the two indexes in [1...p + 1] where p is the number of procedures of packages P that are yet to be assigned to the packages constructed by the players, since null move is also considered it is p+1. The value of the payoff for each player is in [−1, 1]. Thus, the diagonal of the payoff matrix is set to (−1, −1), the lowest payofff or both the players, to avoid that the two players play the same move, seek the same procedure or play both the null move which leads to an infinite loop.

To quantify the gain obtained by a player the function of cohesion and coupling (f_{cc}) is computed using the mathematical model (Nash, 1951) by selecting a procedure p_i and it is given as follows:

$$fcc(P_k, p_i) = \frac{1}{N} \sum_{p_i \in P_k} sim(p_i, p_j) \quad (3)$$

Where:

p_k = The set of procedures assigned at certain iteration to the player

N = The number of methods in P_k

sim (p_i, p_j) = The relationships among procedures p_i, p_j is computed as follows:

$$sim(p_i, p_j) = wssm.\ SSM(p_i, p_j) \quad (4)$$

where, wssm (weight of structural similarity between methods) have values with in (0, 1) and express the confidence (i.e., weight) in each measure.

Output: Grouping of procedures into packages.

EMPIRICAL EVALUATION

In this study, a PL/SQL code named first run (an operation system) in Pay Roll Management System developed using Oracle 10 g which is operational in Anna University, Chennai is used for the study. The first run contains twenty five procedures. The details of the relations name and attributes name accessed by twenty five procedures are given in Table 1.

The relation named deduction_specific stores deduction details of additional house rent, cooperative society, electricity charge and employee's recreation club, mit loan as well as vehicle maintenance. The relation named loan_sanction stores the details of cycle advance, loan sanction, festival advance, handloom recovery, marriage advance, medical advance and motor car loan. The relation house_loan stores house building advance and bank loan stores new bank loan. The relations named court_recovery, cps_recovery and rop stores new court recovery, contributory pension scheme recovery details and recovery of over payment details, respectively. The relation employee_gpf stores employee provident fund details, employee_cps stores employee contributory pension scheme details and gpf_loan stores gpf loan details. The relations pli, spf and income_tax_deduction store postal life insurance policy details, special provident fund details and income tax deduction details, respectively. The relations rec_deposit, licpolicy store the details of recurring deposits, details of life insurance policies, respectively. The relation variable_ded_std stores the details of family benefit fund, mediclaim, government health insurance scheme as well as professional tax.

The first run is not modularized on any aspect and so, it is difficult to maintain. It is not easy to upgrade, customize with respect to changing requirements and debug. The concept of refactoring using game theory is

Table 1: List of procedures and relations used in first run

Procedure name	Relation name	Attribute name
Additional_house_rent	deductions_specific	{eno,dedcode,validity}
Aucse_loan	deductions_specific	{eno,dedcode,validity}
Electricity	deductions_specific	{eno,dedcode,validity}
Erc	deductions_specific	{eno,dedcode,validity}
Mit_loan	deductions_specific	{eno,dedcode,validity}
Vehicle_maintenance	deductions_specific	{eno,dedcode,validity}
cycle_advance	loan_sanction	{eno,loancode,validity}
loan_sanction	loan_sanction	{eno,loancode,validity}
Festival_advance	loan_sanction	{eno,loancode,validity}
Handloom_recovery	loan_sanction	{eno,loancode,validity}
Marraiage_advance	loan_sanction	{eno,loancode,validity}
Medical_advance	loan_sanction	{eno,loancode,validity}
Motor_car_loan	loan_sanction	{eno,loancode,validity}
House_building_ advance	house_loan	{eno,loancode,validity}
New_bank_loan	bank_loan	{eno,branchcode,validity}
New_court_recovery	court_recovery	{eno,validity}
New_cps_recovery	cps_recovey	{eno,validity}
Recovery_over_payment	rop	{eno,validity}
New_pf	employee_gpf, employee_cps, gpf_loan	{eno,validity}
New_pli	pli	{eno,validity}
New_spf	spf	{eno,validity}
Income-tax	Income_tax_deduction	{eno,validity}
Recurring_depost	rec_deposit	{eno,validity}
Lic	licpolicy	{eno,validity}
Newmediclaim	variable_ded_std	{eno,validity}

Table 2: The resultant set of packages by similarity relations

Package name based on relations	Procedure name	Relation name	Attribute name
P1	Additional_house_rent	deductions_specific	{eno,dedcode,validity}
	Aucse_loan	deductions_specific	{eno,dedcode,validity}
	Electricity	deductions_specific	{eno,dedcode,validity}
	Erc	deductions_specific	{eno,dedcode,validity}
	Mit_loan	deductions_specific	{eno,dedcode,validity}
	Vehicle_maintenance	deductions_specific	{eno,dedcode,validity}
P2	cycle_advance	loan_sanction	{eno,loancode,validity}
	loan_sanction	loan_sanction	{eno,loancode,validity}
	Festival_advance	loan_sanction	{eno,loancode,validity}
	Handloom_recovery	loan_sanction	{eno,loancode,validity}
	Marraiage_advance	loan_sanction	{eno,loancode,validity}
	Medical_advance	loan_sanction	{eno,loancode,validity}
	Motor_car_loan	loan_sanction	{eno,loancode,validity}
P3	House_building_advance	house_loan	{eno,loancode,validity}
P4	New_bank_loan	bank_loan	{eno,branchcode,validity}
P5	New_court_recovery	court_recovery	{eno,validity}
P6	New_cps_recovery	cps_recovey	{eno,validity}
P7	Recovery_over_payment	rop	{eno,validity}
P8	New_pf	employee_gpf, employee_cps, gpf_loan	{eno,validity}
P9	New_pli	pli	{eno,validity}
P10	New_spf	spf	{eno,validity}
P11	Income-tax	Income_tax_deduction	{eno,validity}
P12	Recurring_depost	rec_deposit	{eno,validity}
P13	Lic	Licpolicy	{eno,validity}
P14	Newmediclaim	variable_ded_std	{eno,validity}

Table 3: The number of packages obtained from similarity by relations

Methodology	No. of procedures	No. of packages	No. of relations
Similarity by relation	25	14	16

applied to obtain packages that offer several advantages such as modularity, easier application design, information hiding, added functionality and better performance. As per the concept of oracle PL/SQL code package, it is possible to have any procedures that may access different relations. But using this system, the first run is restructured by grouping the procedure that accesses the same relation into one package by applying game theory which leads to better maintainability and understandability.

The details of the procedures and relations accessed by them and the output of the system that gives the packages in which the procedures are grouped by relations are shown in the Table 2. The implementation was done in JAVA.

It is noticed that using similarity by relations the number of packages obtained is 14 which is given in Table 3.

CONCLUSION

The proposed system achieves modularity by restructuring the code that accesses same data into single package thereby it bringing out the advantage of encapsulation that is commonly achieved in object oriented programming into PL/SQL code by using packages. The modularization is done by applying game theory. The modularized code improves the maintainability and understandability of the system and so reduces the overhead whenever the real life scenario changes. Since, the procedure accessing the same relations are grouped into a single package any change that is done to the schema of the relation will be reflected only in one package and thereby reduces the ripple effect. Application of this approach in other empirical studies may result in generating related metrics.

REFERENCES

Alkhalid, A., M. Alshayeb and S. Mahmoud, 2010. Software refactoring at the function level using new adaptive K-nearest neighbor algorithm. Adv. Eng. Software, 41: 1160-1178.

Arnold, R.S., 1986. An Introduction to Software Restructuring. In: Tutorial on Software Restructuring, Arnold, R.S. (Ed.). IEEE Computer Society Press, USA., ISBN-13: 9780818606809, Pages: 365.

Bavota, G., A. de Lucia and R. Oliveto, 2011. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. J. Syst. Software, 84: 397-414.

Bavota, G., A.D. Lucia, A. Marcus and R. Oliveto, 2010b. A two-step technique for extract class refactoring. Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, September 20-24, 2010, Antwerp, Belgium, pp: 151-154.

Bavota, G., R. Oliveto, A. de Lucia, G. Antoniol and Y.R. Gueheneuc, 2010a. Playing with refactoring: Identifying extract class opportunities through game theory. Proceedings of the 2010 IEEE International Conference on Software Maintenance, September 12-18, 2010, Timisoara, pp: 1-5.

Coleman, D.M., D. Ash, B. Lowther and P.W. Oman, 1994. Using metrics to evaluate software system maintainability. Computer, 27: 44-49.

Du Bois, B., S. Demeyer and J. Verelst, 2004. Refactoring-improving coupling and cohesion of existing code. Proceedings of 11th Working Conference on Reverse Engineering, November 8-12, 2004, Delft, The Netherlands, pp: 144-151.

Fokaefs, M., N. Tsantalis, A. Chatzigeorgiou and J. Sander, 2009. Decomposing object-oriented class modules using an agglomerative clustering technique. Proceedings of the IEEE International Conference on Software Maintenance, September 20-26, 2009, Edmonton, AB., USA., pp: 93-101.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, New York, USA., ISBN-13: 9780201485677, Pages: 431.

Griswold, W.G. and D. Notkin, 1993. Automated assistance for program restructuring. Trans. Software Eng. Methodol., 2: 228-269.

Gui, G. and P.D. Scott, 2006. Coupling and cohesion measures for evaluation of component reusability. Proceedings of the 2006 International Workshop on Mining Software Repositories, May 22-23, New York, USA., pp: 18-21.

Guimaraes, T., 1983. Managing application program maintenance expenditure. Communi. ACM, 26: 739-746.

Joshi, P. and R.K. Joshi, 2009. Concept analysis for class cohesion. Proceedings of the 13th European Conference on Software Maintenance and Reengineering, March 24-27, 2009, Kaiserslautern, Germany, pp: 237-240.

Mangasarian, O., 1964. Equilibrium points in bimatrix games. J. Soc. Ind. Applied Math., 12: 778-780.

Mens, T. and T. Tourwe, 2004. A survey of software refactoring. IEEE Trans. Softw. Eng., 30: 126-139.

Nash, J., 1951. Non-cooperative games. Ann. Math. Second Ser., 54: 286-295.

Opdyke, W.F., 1992. Refactoring: A program restructuring aid in designing object-oriented application frameworks. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Illinois, USA.

Refanidis, I. and A. Alexiadis, 2008. SelfPlanner: Planning your time! Proceedings of the Workshop on Scheduling and Planning Applications, September 2008, Sydney, Australia.

Simon, F., F. Steinbruckner and C. Lewerentz, 2001. Metrics based refactoring. Proceedings of 15th European Conference on Software Maintenance and Reengineering, March 14-16, 2001, Lisbon, Portugal, pp: 30-38.