Design of Model For Restructure Transformation of Public Sector

Ashok Kumar [1], Anil Kumar[2]
1.Professor, Department of Computer Science & Application
Kurukshetra University, Kurukshetra, India

2. Asst. Professor, Computer Science & Engg.
Vaish College of Engineering, Rohtak, India

Abstract
Public sector such as Govt. University composed of many physical as well logical threads, which are very beneficial for public to provide services. Over times due to repeated modification of software modules, the structure of the system deteriorates and it become very complex to understand for further modification whenever requirement need to provide services to public, because it is universal truth after specific time period there is need of modification to fulfill the requirement for public. And if we repeat to modify the software module, then it is very complicated just like noodles in chowmin plate and program structure is twisted and tangled. Due to this program structure greatly decrease the scalability, reliability, efficiency, robustness and increased the complexity of software module. And it also increased the maintenance cost of s oftware module, therefore repeated modification is not a good choice. Reengineering is good choice for this.
Therefore, in this paper we will introduced a new methodology that is known as pattern based reengineer methodology[1], that is not only focus on only logical thread, but also focus on ph ysical entities - reduce overall complexity. It is proved that the transformation[2] does not alter the semantic of restructured program.

Keyword: *Restructure Transformation, Reengineering, Reverse Engineering, Forward Engineering, Composition and D ecomposition Design*

Introduction

The software communities has actively responded to the needs of m aintenance and it is very difficult activity, integrating of existing software components. As we know maintenance is not a good choice today, because it is very costly as well as repeated modification deteriorates structure of whole software modules.
Therefore, here we will introduce the new methodology that is known as pattern based reengineering methodology, that is analyzes not only existing system, where modification require, but also analyzes the people who are working in organization and that are involve with software module directly or indirectly. It will analyzes the responsibility of these people who responsible to provide service to public or customer.
It provides action for recommendation, and warranty for greater success in communication procedure.
The main chrematistics of this paper is:

- A description of each and every modules of public/private sector, i.e followed to reengineering, not only reengineering, but also complexity measurement
- It provide a framework, that resulted from our reengineering modules and validated by several case studies
- Detailed description of resulting architecture, which provide benefit in other way.

The proposed methodology based on de sign of existing software module. As we know that the design is silver bullet in software development and diamond bullet during reengineering of software modules. Even though, it has less useful throughout the lifetime of software system, then it should be. Design part of software modules are often large and monolithic and structure of design quite different from that requirement. As a result, developer tends to discard the design, especially, as the system evolve and due to this it is too difficult to keep the relationship to the requirement and software module programming, especially when both are changing.
The purposed methodology, provides flexibility to the decomposition and composition. The existing

decomposition mechanism (class, interface, object, method and package) are extended to includes decomposing designs in a manner directly aligning design and requirement specifications. Composition mechanism for design are extended to support the additional decomposition mechanism that is closely align with both requirement specified and with code.

It illustrate that how purpose methodology, permits the benefits of design to be reengineering throughout a system life time.

Background

Reengineering describes a process of reverse engineering[3,4,5], redesigning[6] and forward engineering[7,8,9] .

Reverse engineering, involves recovering and documenting a system for developers to understand how system works. The abstraction can be discovered by referring to the system experts, system documentation or the source code. In the legacy system the original system experts are often no longer available and system documentation quickly becomes out of date. *Reverse Engineering is focused on the challenging task of understanding legacy program code without having suitable documentation.*

Redesigning, is the process of changing the system abstraction to accommodate the system's present and future requirement.

Forward engineering, is the process of implementation of the new abstraction. Forward engineering practice informal requirements are somehow converted into a *semi-formal* specification using domain notations without underlying precise semantics like e.g. data-flow diagrams, entity relationship diagrams, natural language descriptions, or other problem specific informal or semiformal notations. The program then is constructed manually (i.e. in an error prone way) from the specification by a creative agent, the programmer. Hidden in this creative construction of the program from the specification are a s et of obvious as well as no obvious design decisions about how to encode certain parts of the specification in an efficient way using available implementation mechanisms to achieve performance criteria (the why of the design decisions). **As** an example, a specification fragment requiring associative retrieval using numeric keys may be implemented using hash tables, achieving good system reaction time. These decisions are usually not documented. Over time the program code is modified to remove errors and to adapt the system to change requirements. The requirements may change to allow usage of alphanumeric keys and to be able to handle large amounts of data. Unfortunately, often these changes take place without being reflected correctly in the specification. The gap between the original specification and the program becomes larger and larger. The result is a program code without a proper specification and with untrustworthy design information (such as comments describing the hash tables!). The code becomes difficult to understand and, thus, difficult to maintain. To overcome this deficiency, it is important to change the specification first and then reflect the changes in the program code. **A** necessary precondition for this is to have reliable information about the relationship between the specification and the program code. The design and its rationale describe the how and why of this relationship; however, they are not documented in current practice.

Problem Description

During reengineering of legacy system, there is structural mismatch between requirement specification and existing software system. Due to this, individual requirement are scattered, across the design and support for multiple requirements is tangled in individual design unit. This will reduces comprehensibility and traceability that making the software module design or existing software module code, difficult to understand, develop, r euse and extends.

And usually, while you fix a bug in one place, another bug is pop-pup somewhere else in the system. Long rebuild time make any changes difficult. All of these signs of software module close to breaking point. Many systems could be upgraded or simply thrown away, if they no longer serve their purpose

Related Work

Design patterns were discussed by Christopher Alexander, an architect, in order to describe techniques for town planning, architectural designs, and building construction

Techniques[10] each design pattern description contains a section where relationships to other patterns of a higher or of a lower granularity level are presented. These relationships influence the

construction process. A classification for the patterns was
given, however their mutual relationships have not been provided. In[11], a large collection of well described design patterns was presented. The relationships between design patterns are also described, but not classified. However a clustering of related design patterns was included. Such clustering according to *jurisdiction* (class, object, compound) and *characterization* (creational, structural, behavioral) is orthogonal to the one derived in this paper. In this context, patterns in a specific cluster can be considered as similar to another one which supports the selection of an appropriate design patterns for a certain problem. Frameworks[12,13] are also considered as high-level design patterns, usually consisting of many interrelated design patterns of lower levels.

In[14], it is indicated that "*Patterns can be used at many levels, and what is derived at one level can be considered a basic pattern at another level*". Furthermore, it is stated that
"*This is probably typical of most architects; some patterns will be generic and some will be specific to the problem domain*" which also confirms the organization depicted in
our proposed layers. Booch[15] also discussed that design patterns are ranging from *idioms* to *frameworks*. In, several design patterns[16]] are combined in an exemplary
application, but the relationships are not investigated further.
The relationships between object-oriented design patterns were first analyzed in [7] where three kinds of relationships between patterns are described. These include :
i) *use* - one pattern can *use* another pattern, ii) *variant* – one pattern can be *a variant of* another pattern, iii) *combine* - two patterns can be used *in combination* to solve a problem. Similarly, Mesazaros and Doble[[18] identified five relationships between patterns, a pattern can *use*, *be used by*, *generalize*, *specialize*, or provide an *alternative* to another pattern.

Research Goal

- **Reduction of Maintenance Costs:** the manually restructured software modules must be tested to ensure their behavior is not changed. This increased the cost of maintenance. Software modules restructured using our transformation need not be retested, since their external semantics is guaranteed to remain same.
- **Smooth Migration of Old Software Module Code to New Technology:** due to rapid changes of technology, there is a constant need to migrate software developed using one programming language or design paradigm to another. Our transformation may be used to restructure the old software module code, such that it effectively use the advantages offered by a new paradigm.

Purposed Work

The purposed methodology is based on Decomposition[19,20] and composition design[21,22]

Decomposition Design: matching the structure of requirements, during reengineering of software module by dividing up into separate module, that match the change structure. And each separate module, separately describes that part of a system or component that relates to a particular requirement, encapsulating its design and separating it from the design of rest of the system. It support with the requirement specification is to have a one-to-one match of requirement with modules. It is supported, while multiple requirements with single module.

The detailed process of decomposition consists of the following steps: 1) generation of functional-level component descriptions in Component from the source tree, 2) analysis of functional-level components in terms of modularity factors and modification of the descriptions to enhance modularity, 3) modification of actual source tree based on the refined component descriptions and generation of build-level component descriptions along with verification of the builds, 4) verification of the refined source tree against the component descriptions, and 5) testing the components.

Composition Design: decomposition of module design brings many benefits relating to comprehensibility, traceability, evolution and reuse. However, the design that have been decomposed must also be integrated later stage, in order to

understand the design of the system as a w hole. This required for the reasons such as verification or to support a developer to understand the semantics of the design and the impact of composition on the design.

Composition of module design, help to understand relationship between designed module to be composed. This will compare the specification of behavior of module to another.

The composition process consists of the following steps: 1) selection of components from the component repository, 2) construction of a source tree that combines the source trees of the selected components, and 3) generation of build scripts that build the combined source tree.

Purposed Methodology, Pattern Based Reengineering methodology(PBRM), helpful to software engineering, software analyst, software designer and software programmer to understand the existing software modules where reengineering required, just like a doctor examine patient, to know which type treatment is given to patient.

PBRM, examine existing software system just like a doctor examine patient. E.g

First of all while doctor examine the patient, he/she try to know which type of infection/disease infected to patient, then examine nervous system, heart beat, blood circulation and finally make a case history and the start its medicine, once case history is prepared that is used for further check up p atient. And time to time checkup of patient takes place to know either there is need to change medicine or guide for physical exercise to recover. This recovering process of patient just like reengineering of software modules.

In case of PBRM, first of all try to understand what is the actual requirement of services that is need to be implemented in existing software modules. Here, requirement engineering is helpful for this purpose. Then, decomposition design and composition design takes place to design the software module, then verification and validation of software modules takes place, through examine the flow of control in software modules.

PBRM, support restructuring transformation of software modules, restructuring is based on decomposition and composition. While restructure transformation of software module takes place PBRM, software module is change its internal structure without affecting its external behavior. This transformation should separate the inter wined logical threads of an old program, to reduce its complexity.

PBRM provides a model that is known as a model, that is helpful for restructuring transformation of software modules. This model is iterative model, where each and every task is performed iteratively, until or unless it is not requirement of reengineering and depending upon requirement, it help full to assign priority, depending upon requirement need.

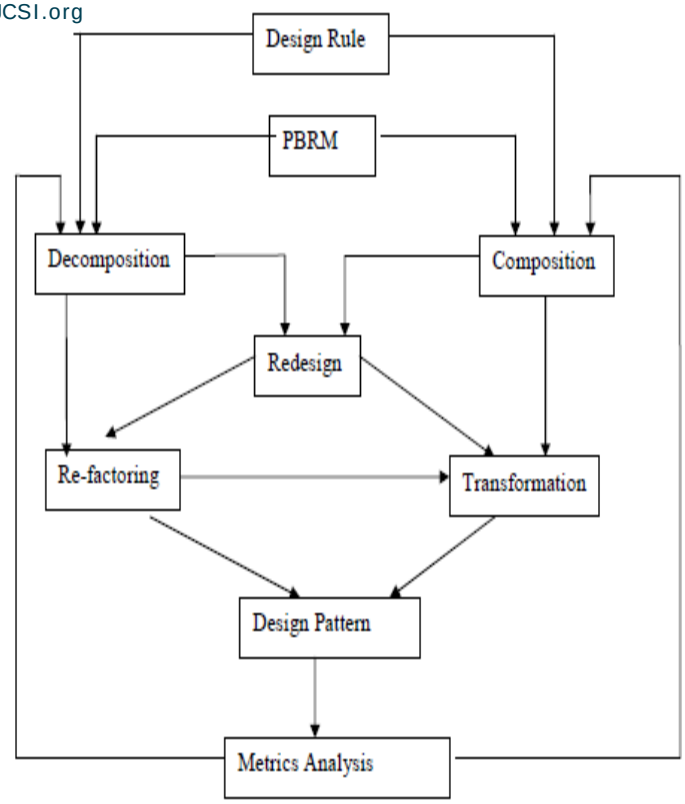
A Model has following activities:

1. Identify each task
2. Identify the ‘depend on’ relations between each pairs of task
3. Determine the order in which the task are to be restructured
4. Assign priorities among task, according to requirement
5. Restructured each task
    i. Identify the computation that influence the given task
    ii. Collect all these computation in a new module and create a function call to the new function in the appropriate position of the original procedure.

Each step in the above model may be considered to be independent of other step.

This model is used for two purpose : Sketch and Blueprint

Sketch is used as a thinking tool, which help developer to communicate some aspects of a system and alternatives about, what are to be done. Blueprint is used for guiding for implementation.



'Pattern Based Reengineering Model'

PBRM, provides following type of metrics, that is used to determine complexity of software module during reengineering.

1. Number of Attributes of Pattern of class/interface: measure the ratio of the total number attribute of pattern of class in a model to be implemented
2. Size of Attribute of pattern of Class/interface: measure ratio of attributes of pattern with a signature to the total number of attribute of pattern of class
3. Number of Operation of class/interface: measure the ratio of total number of operation of a class/interface in a model to be implemented
4. Operation with Parameter of class/interface: measure the ratio of operation with parameter of a cl ass in a m odel to that implemented
5. Operation with Return of Class/interface: measure the ratio of operation which return value of a class in a model to that in the implementation
6. Association Label of Class/interface: measure total number of association of class/interface
7. Association Rule of Class/interface: measure total number of association attached to a class/interface

Conclusion

Pattern based reengineering methodology, is successful technique in planning where reengineering are takes place, what is actual requirement, which one activity performed first according to need. It also helpful in problem detection, migration strategies and software redesign. PBRM, provide suitable documentation i.e. helpful to understand the system, in future, after reengineering is completed.

The proposed methodology helps software engineers to

: i) better understand the complex relationships between design patterns,

ii) organizes existing design patterns as well as categorizing and describing new design patterns,

iii) build a model which supports the application of design patterns during restructuring transformation and complexity measurement.

References

1. *K. Beck. Patterns and software development. Dr. Dobbs Journal, 19(2):18–23, 1993.*
2. *H.A. Partsch. SpecGcation and Transformation of Programs: A Formal Approach to Software Development. Springer, 1990.*
3. *H. Muller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem identification. Software Maintenance and Practice, 5:181–204, 1993.*
4. *4 H. Muller. Rigi as a reverse engineering tool. Technical Report DCS-160-IR, University of Victoria, Victoria, BC, Canada, 1991*
5. *H. Muller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineeringapproach to subsystem identification. Software Maintenance and Practice, 5:181–204, 1993.*
6. P. Tollena and G. Antoniol. Object oriented design patterns inference. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM), pages 230–238, September 1999
7. *H.A. Partsch. Specification and Transformation of Programs: A Formal Approach to Software Development. Springer, 1990.*

8. *S. Rugaber, K. Stirewalt, and L. Wills. The Interleaving Problem in Program Understanding In: Working Conference on Reverse Engineering, pp. 166-175, 1995*

9. *S. Rugaber. White Paper on Reverse Engineering. Georgia Institute of Technology, 1994.*

10. C.Alexander. *A Pattern Language*.Oxford University Press, 1977

11. E. Gamma, R. Helm, R. Jahnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

12. R. E. Johnson and V. F. Russo. Reusing object-oriented designs. Technical report uiucdcs 91- 1696, University of Illinois, May 1991.

13. R. J. Wirfs-Brock and R. E. Johnson. Surveying current research in object-oriented design. *Communications of ACM (CACM)*, 33(9):105–123, September 1990

14. K. Beck. Patterns and software development. *Dr. Dobbs Journal*, 19(2):18–23, 1993.

15. G. Booch. Patterns. *Object Magazine*, 3(2), 1993.

16. P. Coad. Object-oriented patterns. *Communications of ACM (CACM)*, 35(9):153–159, September 1993.

17. F. Buschmann et al. *Pattern-Oriented Software Architecture : A System of Patterns*. JohnWiley and Sons, 1999.

18. G. Meszaros and J. Doble. A pattern language for pattern writing. In R. Martin, D. Riehle, and B. F., editors, *Pattern Languages of Program Design*, volume 3, pages 529–574. Addison-Wesley, 1998.

19. R.J. R. Back and K. Sere, "Stepwise refinement of parallel algorithms," *Sci. Comput. Programming,* vol. 13, pp. 133-180, 1990.

20. Y. Liu, A. K. Singh, and R. L. Bagrodia, "A decompositional approach to the design of efficient parallel programs," Dep. Comput. Sci., Univ. Califomia at Santa Barbara, Tech. Rep., Sept. 1994.

21. M. Abadi and L. Lamport, "Composing specifications," in *Stepwise Refrnement* of *Distributed Systems,* J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., LNCS 430, Berlin: Springer-Verlag, 1990.

22. M. D. Jonge, "Multi-level Component Composition," 2nd Groningen Workshop on Software Variability Modeling (SVM'04), Reseach Institute of Computer Science and Mathematics, University of Groningen, Dec. 2004.

Dr. Ashok Kumar has received his Ph.D degree from Agra University, Agra, India. He has joined as a Professor in the Department of Computer Science & Application, Kurukshetra University, Kurukshetra – 1361199 (Haryana), India, in June 1982. He has published more than 60 national and international papers. He has attended more than 30 na tional and international seminars. His area of interests are software engineering, operational research, networking and operating system.

Anil Kumar received his Master degree from IGNOU, India and M.Tech in Computer Science & Eng. From Kurukshetra University, Kurukshetra, India in year 2002 a nd 2006. He is pursuing Ph.D in Computer Science from the Department of Computer Science & Application – Kurukshetra University, Kurukshetra, India. Currently he is working as an Asst. Professor in Computer Science & Engineering Department in Vaish Engineering College, Rohtak, Haryana, India since September, 2006. He has also worked in software industries for more than three years . His research area includes Software engineering, Reengineering, Software Metrics, Object Oriented analysis and design, Reusability, Reliability.