A Framework for Software Reengineering Using Set of Software Metrics

Dr. Sumesh Sood
Department of Computer Applications
SSCMT
Amritsar, Punjab
sumesh64@gmail.com

***Abstract*— Software metrics support various reengineering tasks. A set of software metrics can be used to identifying the quality problems with the current system and the prioritization of applications that are candidates for reengineering according to their technical quality and business value. The metrics can be used in the measurements of the costs of changes in the software, if an increase in maintainability is one of the goals of the reengineering process. Also at the conclusion of the re-engineering effort the same metrics can be used to identify the quality of the new system and the return on investment. The collection of metrics on the new system can continue throughout development. To demonstrate, a metric framework has been used for reengineering process. This framework is used in different phases of Rainfall model to make reengineering process easy, economical and efficient. Case study of software has been undertaken to validate this metric framework.**

Keywords- Maintainability Index; Partial Reengineering; RRC; RRCM; SourceMonitor

I. INTRODUCTION

Reengineering legacy systems become a vital matter in today's software industry. When software becomes obsolete, some companies decide to retire the software and redevelop the whole system, while other go through a process of upgrading the software for various reasons, to be able to resale the software again in the new market or to use the software in the work place [15]. This process of upgradation of software is called reengineering.

Sometimes it is more cost effective not to reengineer the whole software but the part of the software. This process is called partial reengineering. The process of partial reengineering provides an opportunity to look at the existing design and to identify opportunities for improvements. Hence, Partial Reengineering is the process of identifying parts of the system to be changed (candidate system), creating an abstract description of a system, reason about a change at the higher abstract level, re-implements the candidate system and integrate the whole system i.e. old system (excluding candidate parts) and redeveloped candidate system [16].

II. REVIEW OF LITERATURE

In early years of the information revolution the need for reengineering was not acknowledged by the wider community, instead, attention was directed towards the new ways of creating better software.

In 1990 Chikofsky and Cross described the reengineering of software as 'the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form' [4]. The fact, that so much attention was given to reengineering that entire businesses were caught up in the excitement and had their entire business structure recognized according to the newly developed reengineering methodologies and patterns that were emerged.

Soon it became apparent that the reengineering of both business and its software was not as easy as the consultants had first believed. Over half of the reengineering processes of the time failed, mostly due to inexperience and lack of customer's involvement. With these failures resulting in huge lose for the companies and soon the reengineering boom was over and so was the interest in reengineering development. This shows that reengineering, the reorganization and redesign of a system is very important, since if these costs can be reduced, much will be gained for software users.

In 1998 Unified Modeling Language was adopted unanimously by the membership of OMG (Object Management Group) as a standard [6].

MORALE (Mission Oriented Architecture Legacy Evolution) is reengineering methodology developed in Georgia Institute of Technology in 1997 by Abowd et al. The MORALE address the problem of designing and evolving complex software systems [1].

In the late 90's Kazman, R. [8] and Woods, S. [18] developed a conceptual "horseshoe" model that distinguishes different levels of reengineering analysis and provides a foundation for transformations at each level, especially for transformations at the architectural level. This model describes the rich set of technical choices that reengineers make. However, because of its technical focus, it has not been accessible to decision makers in a form that can assist them in deciding on complex options regarding the future of their legacy systems.

Singh and Sood (2006) in their study presented four scenarios of software reengineering [14] and with this the need for partial software reengineering arises.

In 2007 Chiang in his work explained the connection between the stability modeling and reengineering process for legacy system [2].

In 2009 Mishra et. al. have designed a model CORE (Component Oriented Reverse Engineering) to identify and develop reusable software components [10]. By using the reverse engineering techniques they extracted architectural information and services from legacy system and later on convert these services into components that can be reusable later. But again in this paper they have explained reengineering of the software. But they were not able to explain that why they had preferred to reengineering the components, in place of developing the new components from scratch.

In 2010 Tucker and Simmonds presented a paper in Seventh International Conference on Information Technology [17]. In this paper they describe a case study in perfective and adaptive reengineering. But again there was no comparison why they choose reengineering in place of maintenance or developing software from scratch.

In 2011 Hong Zhou proposed knowledge based software reengineering approach in the context of software reengineering and knowledge representation [19]. It is an application of description logic and ontology to the task of constructing computable models for the software reengineering domain. His thesis aims to improve the traditional software reengineering methods by proposing a knowledge based software reengineering approach via ontology and description logic.

Cholakov and Birov (2012) in their article represented a model for automated reengineering of legacy software systems [5]. It describes in details the processes of software translation and refactoring and the degree of automation that these processes may achieve.

From the review of literature, it can be concluded that a few studies discussed the circumstances where reengineering is preferred over maintenance and the various parameters on the basis of which the partial reengineering may be helpful to maintain the legacy software. So, stress to be given to study and discuss the need of partial reengineering to reduce the effort and cost, and to increase the efficiency during reengineering of a legacy system.

III. RESEARCH METHODOLOGY

In this paper Rainfall Model is used as candidate model. A metric framework has been proposed for reengineering process. This framework is used in different phases of proposed model to make reengineering process easy, economical and efficient. Case study of software, used to manage the stock of iron in an iron refinery, has been undertaken, to validate this metric framework. After applying the metric framework on the software, results are used to validate the techniques used in the paper. To make reengineering process automated and easy, a tool SourceMonitor is used.

IV. METRICS USED IN REENGINEERING PROCESS

In this section, some software metrics that have proven to be particularly useful will be discussed. These metrics fall into several categories depending on the aspects of a system they measure. The metrics presented in Table I. are of following categories: complexity metrics, coupling metrics, cohesion metrics, cost metrics and test metrics. Each metric is presented with its serial number, full name, abbreviation, a category (the metrics in this paper is grouped into certain categories) and a description about the working and use of metrics.

TABLE I. METRICS FOR REENGINEERING PROCESS

S.No.	Name	Category	Description
1.	*Defect Age (DA) [11]*	*Test*	*Defect Age (in Time) is the difference in time between the dates, a defect is detected and the current date (if the defect is still open) or the date this defect was fixed (if the defect is already fixed). Where 'fixed' means that the defect is verified and closed, not just 'completed' by the developer. The 'difference in time' can be calculated in hours or in days. The defect age is computed as shown* *Defect Age in Time = Current Date – Defect Detection Date*
2.	*Number Of Defects (NOD) [9]*	*Test*	*It is measure of total number of remarks found in a given time period/phase/test type that resulted in software or documentation modifications. Only remarks that resulted in modifying the software or the documentation are counted.*
3.	*Time To Solve A Defect (TSD) [9]*	*Test*	*It is effort required to resolve a defect (diagnosis and correction). It provides an indication of the maintainability of the product and can be used to estimate projected maintenance costs. It can be calculated by Divide the number of hours spent on diagnosis and correction by the number of defects resolved during the same period.*
4.	*Weighted Method Count (WMC) [3]*	*Complexity*	*Measures the complexity of a class by adding up the complexities of the methods defined in the class. Thus,* $WMC = \sum_{i=1}^{n} C_i$ *where Ci denotes a complexity measurement of method i.*

			Complexity measurements for methods are usually given by code complexity metrics like LOC or the McCabe cyclomatic complexity. The McCabe cyclomatic complexity measures the complexity of some code by taking into account the decision structure of the code, i.e. code that contains a lot of loops or if-then-else-constructs is considered more complex.
5.	***Lines Of Code (LOC) [7]***	***Complexity***	***Measures the size of a piece of source code by counting its lines. Since the size of some source code can be seen as an indicator of its complexity, LOC is used as a complexity metric or as an indicator on how much effort required implementing that piece of code. The line counting is usually done with respect to a certain coding standard which defines precisely what constitutes a line of code in a particular programming language. This is necessary for obtaining comparable, well-defined measurement results.***
6.	***Tight Class Cohesion (TCC) [14]***	***Cohesion***	***Measures the cohesion of a class as the relative number of directly connected methods, where methods are considered to be connected when they use at least one common instance variable. More formally TCC for a class C is defined as follows: Let NDC = |{(m, n) | methods m, n access a common instance variable }| be the number of connected methods and NPC = n(n-1)/2 is the possible number of connected methods, then TCC = NDC/NPC***
7.	***Number of Children (NOC)[3]***	***Complexity***	***The Number of Children (NOC) represents the number of immediate subclasses subordinated to a class in the class hierarchy. This metric may be used in order to detect misuses of subclassing, and in many cases this means that the class hierarchy has to be restructured at that point during the redesign operation.***
8.	***Coupling between Objects (CBO) [3]***	***Coupling***	***Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class. The uses relationship can go either way: both uses and used-by relationships are taken into account, but only once. CBO = number of classes to which a class is coupled***
9.	***Maintainability Index [12]***	***Cost***	***The Maintainability Index is one of the most mathematically challenged metrics that has proven to be useful. A program with a maintainability index under 65 is hard to maintain. It is also used to calculate maintainability of the software after changes to compare it with previous software. Maintainability of the software can be calculated by using following Eq.*** $$MI = 171 - 5.2\ln(aveV) - 0.23aveV(g') - 16.2\ln(aveLOC) + 50\sin\left(\sqrt{2.4 * aveC}\right) \quad (1)$$
10.	***Defect Cost (DC) [15]***	***Cost***	***DC can be calculated by multiplying ratio between defect age in years and software age in years with lines of code affected by defect and total no of lines of codes. DC is directly proportional to defect age, as the age of defect increases, cost of DC increases, because it may be possible that this defect is difficult to remove. Also DC is inversely proportional to the age of software, as the software is older that means defect is not severe, as is it has not affected the software for long time. Hence its cost decreases. Defect Age and Software Age are taken as upper integer in years because if software is new then this ratio do not fluctuate and remain one. DC also depends on ratio of number of lines of code affected by defect to total number of lines of codes. If there is requirement to add other functionality then lines affected by defect is number of statements that will be added for increasing the functionality and can be calculated by using Fuzzy Logic Method [13].*** $$DC = \frac{\text{Ceil(Defect age in years)}}{\text{Ceil(Software age in years)}} \times \frac{\text{Lines affacted by defect}}{\text{Total number of lines}} \quad (2)$$
11.	***Fault Cost (FC) [15]***	***Cost***	***FC can be calculated by calculating ratio between mean time to maintenance and time between last two maintenance tasks. When failures become frequent above ratio increases and hence value of FC increases. FC is taken as 0 when software is maintained first time.*** $$\text{FC} = \frac{\text{Mean time to maintenance}}{\text{Time between last two maintenance}}$$

12.	*Reengineering Requirement Cost (RCC & RRCM) [15]*	*Cost*	*The value of RRC can be calculated by adding twenty times value of DC of each bug (number of bugs can be calculated by the metric Number of Defects [79]) and FC.* $$\mathbf{RRC} = \mathbf{20}\sum_{\mathbf{i-1}}^{\mathbf{n}} \mathbf{DC_i} + \mathbf{FC} \qquad (\mathbf{3})$$ *where n is number of defects.* *To find whether there is requirement to maintain, reengineering or retire the system or its modules two cases arise.* *a. Software consists of one module only.* *b. Software consists of more than one module.* *a. Case 12 A:* *If the software consist of one module only then* • *If RRC is less than 3.0 then there is no requirement of reengineering.* • *If RRC is between 3.0 and 6.0 then there may be requirement for reengineering.* • *If RRC is between 6.0 and 10.0 then there is high requirement for reengineering.* • *If RRC is greater than 10.0 then reengineering cost will be very high and system must retire and redesign using new architecture and techniques.* *b. Case 12 B:* *Reengineering Requirement Metric calculates whether there is requirement to maintain the software, reengineer software or retire software. It also calculates if there is a requirement to reengineer software then whether whole software requires reengineering or some part of software require reengineering. In this metric four variables, efect Cost (DC), Fault Cost (FC), Reengineering Requirement Cost (RRC) and Reengineering Requirement Cost of Module (RRCM) are used.* *If software consists of more than one module and if the value of RRC is between 3.0 and 6.0 then each module of the software is checked by adding half of RRC of whole software to RRC of ith module to calculate Reengineering Requirement Cost of Module i (RRCM).* $$\mathbf{RRCM_i} = \frac{\mathbf{1}}{\mathbf{2}}\,\mathbf{RRC} + \mathbf{RRC}(\mathbf{M_i}) \qquad (\mathbf{4})$$ • *If RRCM is less than 6.0, then there is no requirement of reengineering.* • *If RRCM is between 6.0 and 10.0, then there is requirement to reengineer the module.* • *If RRCM is greater than 10.0 then reengineering cost will be very high and new module should be redesign.*

V. CASE STUDY AND RESULT ANALYSIS

Software (Stock Management) use to manage the stock of cast iron foundry is taken as case study. Software is developed in C++. It is used to keep track of stock of hard coke and pig iron in an iron foundry. It consists of 3 modules and has 1366 LOC. The first module main is used to manage the other 2 modules. Second module Hard Coke is used to keep track of stock of hard coke and third module Pig Iron used to keep track of stock of pig iron. This software is developed with little documentation and lot of extra variables and statements. So there is requirement to look into the source code again. One more drawback of the software is that there are some methods which are not necessary and can be merged with other methods. Hence there is need to change the system by removing and changing some code and adding documentation. In the case study Rainfall model is used as candidate model. In the rainfall model process of reengineering is divided into five phases. So the whole case study is divided into five phases. In each phase some metrics are used to help the reengineering process as shown below.

Phase I: Identification of Candidate System

The software of Stock Management consists of 1366 statements, out of which 257 needs to be changed. DC (metric 10) can be calculated by multiplying ratio between ceiling value of defect age in years and software age in years, with lines of code affected by defect and total no of lines of codes (Eq. 2). In this case ratio between defect age in year and software age in year is 1 as the age of software is less than 1 year. Total lines affected by defect are 257 and total lines of the software are 1366. When calculated, DC value of the software is 0.188 (257/1366). Since software is changed first time, so FC (metric 11) value of the software is 0.0.

Now after putting the value of DC and FC in Eq. 3, the value of RRC (metric 12 A) is 3.76. So there is requirement for reengineering. Since it is between 3.0 and 6.0, hence, there requirement to calculate RRCM (metric 12 B) value of its each module independently, to find reengineering requirement of each module. After calculating RRCM of each module, results are shown in Table II.

TABLE II. RESULT OBTAINED AFTER APPLYING EQ. 4 IN MODULES OF SOFTWARE STOCK MANAGEMENT

S. No	Name	LOC	DC	FC	RCC	RRCM	Result
1.	*Main*	*137*	*.124*	*---*	*2.48*	*4.36*	*Maintain*
2.	*Hard Coke*	*668*	*.212*	*---*	*4.24*	*6.12*	*Reengineer*
3.	*Pig Iron*	*561*	*.174*	*---*	*3.49*	*5.37*	*Maintain*

From the Table II., it is clear that there is requirement to maintain the main and Pig Iron module and reengineer the Hard Coke module. Now only Hard Coke module is candidate system for software reengineering.

Phase II: Reverse Engineering

Now in the software some modules are more complex and are tightly coupled with other modules. These modules can be called important components of the software. Reverse engineering can be started from these components. In Hard Coke a coupling value (metric 8) of method disp() is 16, so it is a candidate for important components because it manages a lot of other methods. To understand how the system works, emphasis should be on understanding this component and its interactions by studying its source code. This will act as starting point during reverse engineering process.

Phase III: Architecture Transformation or Change

After reverse engineering two methods lostfocus and lostfocus1 are combined because both the methods are doing same work and only difference between them is list of parameters. Since, modules that are tightly coupled (metric 8) cannot be seen as isolated parts of the system and hence, difficult to change. Changes in such parts require a lot of work because a lot of other parts depend upon them. These modules should be carefully examined and tested after modification of the system. So method disp(), which has high complexity (LOC is 295, calculated using metric 5) and coupling value is examined for error and its interaction with other methods after changes are made. Also statements, which are not affecting the software, are removed and documentation is added.

Phase IV: Development of Candidate System

Now changes are made in the design and are implemented in the coding. To make sure system does not misbehave after the coding change sensitive parts (parts that are most likely affective by change in a system because they depend on lot of other parts) has been identified. Method disp() has highest coupling value i.e. 15 (using metric 8) and are mainly affected by the merger of two methods. Hence during testing of the whole module stress should be given to this method.

Phase V: Integration

During the integration, candidate system (Hard Coke module) is combined with Main and Pig Iron modules. To differentiate Stock Management software before and after reengineering, it is called SMold (before reengineering) and SMnew (after reengineering).

Now when the software is tested for complexity (using SourceMonitor) and maintainability (metric 9). It is found that average complexity of the software is decreased from 4.3 to 4.1 and value of maintainability is increased from 94.8 to 101.6.

Applying the SourceMonitor, it has been found that statements/method is decreasing from 18.8 to 16.6, average depth is decreasing from 7 to 6 and average complexity is decreasing from 4.3 to 4.1 as shown in Table III.

TABLE III. RESULT OF TOOL SOURCEMONITOR ON SMOLD AND SMNEW

Software	Statements(K)	% Branches	Methods/class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Complexity
SMold	*1.458*	*14.5*	*8.69*	*18.8*	*80*	*7*	*2.53*	*4.3*
SMnew	*1.180*	*12.5*	*8.34*	*16.6*	*74*	*6*	*1.8*	*4.1*

Table III. is represented in pictorial form in Fig. 1 and Fig. 2.

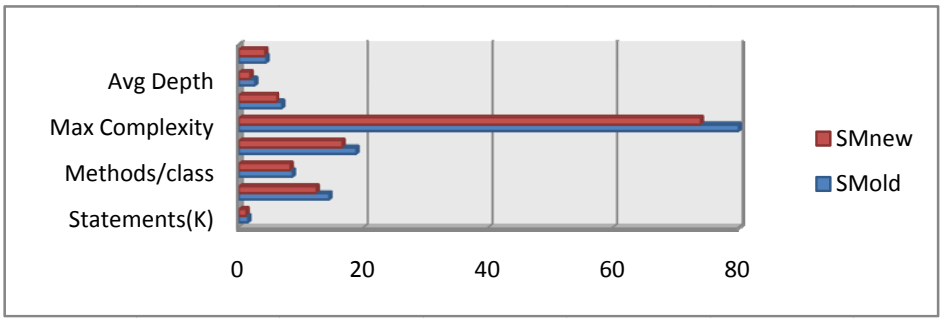

Figure 1. Bar Chart for SMold and SMnew Software

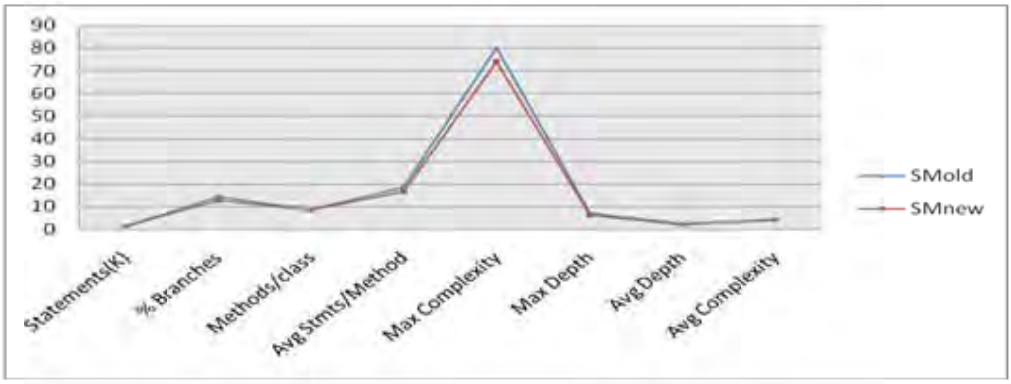

Figure 2. Line Chart for SMold and SMnew Software

A very high level quality goal for a software system could be maintainability. Now to find effect of reengineering on the maintainability of the software, maintainability index of the software is calculated by using Eq. 1. The value of maintainability index of SMold and SMnew are shown in Table IV.

TABLE IV. MAINTAINABILITY INDEX OF SMOLD AND SMNEW

Software	Functions	Vocabulary	Cyclomatic Complexity	Average LOC	Comments	Maintanability Index
SMold	***61***	***80.8***	***18.2***	***22.39344262***	***1.23811095***	***94.85178628***
SMnew	***60***	***65.4***	***13.3***	***17.66666667***	***1.987361869***	***101.6684362***

Fig. 3 and Fig. 4 showing pictorial representation of Table IV.

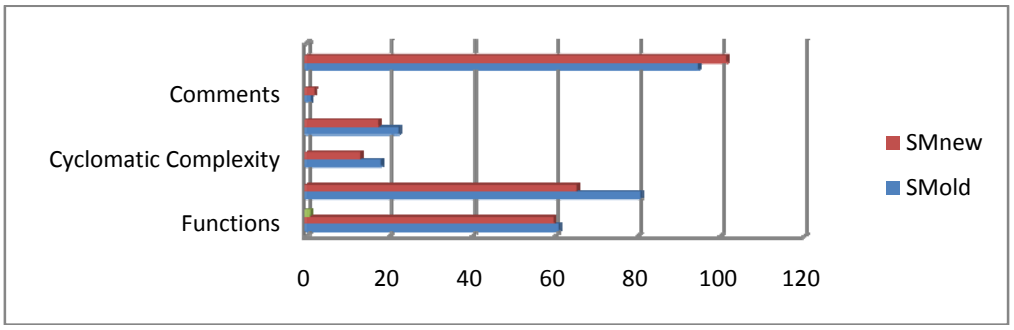

Figure 3. Bar Chart for Maintainability Index of Stock Management

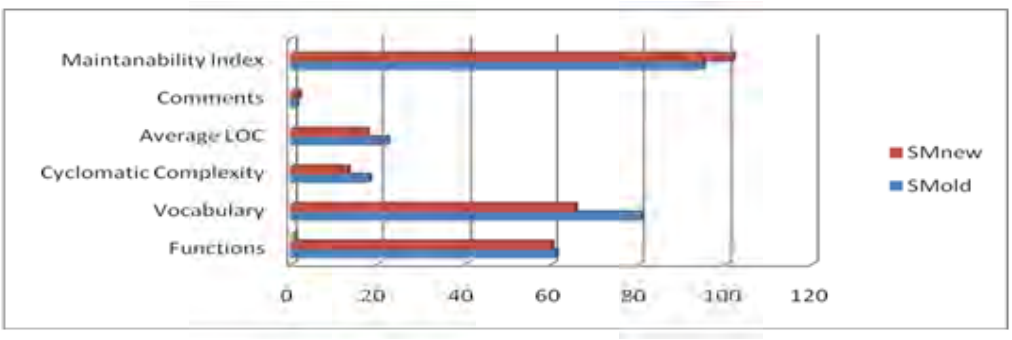

Figure 4. Line Chart for Maintainability Index of Stock Management

From the Table IV., it can be seen that number of functions are decreasing from 61 to 60, vocabulary is decreasing from 80.8 to 65.4, cyclomatic complexity is decreasing from 18.2 to 13.3 and average LOC is decreasing from 22.4 to 17.6 and hence, maintainability index increases from 94.85 to 101.67. So, SMnew is more maintainable as compare to SMold.

VI. CONCLUSIONS

Reengineering projects can benefit from metrics. Applied with well documented scenarios, they make reengineering tasks more organized and focused. They provide an abstraction mechanism from the huge amount of source code of the legacy system, thus allowing us to concentrate our work on the important or critical parts of the system that have been identified by the measurements.

With this work the partial reengineering technology using metrics will be promoted. Organization will be aware of this technology and they can go for partial reengineering. Old software are renovated as new one instead of developing the new one. The organizations can apply the new technology to their old software product / part of software product and performance of the product is increased without much cost and effort. Hence, by promoting partial reengineering there is a lot of saving of time and efforts of the organization.

REFERENCES

[1] Abowd G, Goal A, Jerding DF, McCracken M, Moore M, Murdock JW, et al. MORALE, Mission Oriented Architectural Legacy Evolution, Proceeding of International Conference on Software Maintenance, Bari (Italy), 10/01/97-10/03/97; 150-9.
[2] Chiang CC. Software Stability in Software Reengineering, Proceeding of IEEE International Conference on Information Reuse and Integration, Las Vegas, 13/08/07-15/08/07; 719-23.
[3] Chidamber SR, Kemerer CF. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 1994; 20(6).
[4] Chikofsky EJ, and Cross JH. Reverse engineering and Design Recovery: A Taxonaomy, IEEE, 7(1), 1999; p.15.
[5] Cholakov T, Birov D. Automated Software Reengineering Model and Framework, Proceedings of the Forty First Spring Conference of the Union of Bulgarian Mathematicians, Borovetz, 2012; 225-31.
[6] Framinghan MA. Unified Modeling Language Specification Version 1.1, Object Management Group. 1998; Available from: www.omg.org.
[7] Humphrey WS. Introduction to the Personal Software Process, SEI Series in Software Engineering, Addison Wesley; 1997; p. 22.
[8] Kazman R, Woods SG, Carriere SJ. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II, Proceedings of WCRE 98, Honolulu, HI, 1998; 154-63.
[9] Konda KR. Measuring Defect Removal Accurately, Proceeding of The Enterprise Development Conference, 1998; p. 35.
[10] Mishra SK, Kushwaha DS, Misra AK. Creating Reusable Software Component from Object-Oriented Legacy System through Reverse Engineering, Journal of Object technology, Jan-Feb 2009; 133-52.
[11] Morrison J. Software Testing Fundamentals, 2009. Available from http://www.softwaretestingfundamentals.com /2009/05/defect-age.html.
[12] Oman P, Hagemeister J. Constructing and Testing of Polynomials Predicting Software Maintainability, Journal of Systems and Software, 1994; 24(3), 251-66.
[13] Putnam L H, Myers W. Measure for Excellence: Reliable Software on Time, within Budget, Englewood Cliffs, NJ, 1992.
[14] Singh H, Sood S. Reengineering Process and Methods: A Study, Proceedings of the conference Innovative Application of IT and Management for Economic Growth, Jalandhar, India, 2006; 392-404.
[15] Singh H, Sood S, Kaur R, Ratti N, Metrics Framework for Reengineering Process, Punjab University Research Journal, 2007; 57, pp. 251-255.
[16] Sood S. Use of Metrics for Identifying the Framework of Reengineering Process, Submitted as dissertation of M. Phil. to MKU, Madurai, (2006).
[17] Tucker DC, Simmonds DM. A Case Study in Software Reengineering, Proceeding of Seventh International Conference on Information Technology, Las Vegas, Nevada, USA, April 12- April 14, 2010; 1107-12.
[18] Woods S, Carriere SJ, Kazman R. A Semantic Foundation for Architectural Reengineering and Interchange, Proceedings of the International Conference on Software Maintenance (ICSM-99) Oxford, England, 1999; 391-8.
[19] Zhoug H. A Knowledge Based Reengineering Approach via Ontology and Description Logic, Ph. D. Thesis, Software Technology Research Laboratory, De Montfort University. (2011).