SindhUniv. Res. Jour. (Sci. Ser.) Vol. 49 (004) 861- 868 (2017)

http://doi.org/10.26692/sujo/2017.12.0073

SINDH UNIVERSITY RESEARCH JOURNAL (SCIENCE SERIES)

Software restructuring for enhancing the Cohesion using Backward Slicing

M. K. SHAIKH, M. A. ANSARI, M. MEMON, M. R. MAREE*

Qaid-e-Awam University College of Engineering Science and Technology, Larkana, Pakistan

Received 12rd March 2016 and Revised 19th July 2017

Abstract: Software structure is characterized by high cohesion among modules. Repeated modification into code can adversely affect cohesive structure of software systems. Efficient code transformation is required to re-engineer the software system for enhancing the cohesion. In this paper we suggest restructuring process objectively for cohesive bond within the module using backward tracing of dependent instance variables. In proposed measure of cohesion output variables produce domain of restructuring. The approach of cohesion measure is further validated experimentally on open source software. Our transformation methodology application evolves from small software to large software. Our approach significantly improves the cohesion and reduces complexity of ill structured code resulting quality software.

Keywords: Software Restructuring, Cohesion, Backward slicing

1. INTRODUCTION

Software is composed of many small logical components. Over time, due to repeated modifications, the structure of a system deteriorates, causing its logical threads to get intertwined, like noodles in a bowl of spaghetti. Essentially state of art mechanism is required to comprehend the code efficiently.

Developing bug free software's have been significant domain of subject areas of software maintenance and Software quality assurance over the years. However, key factors like efficiency and of process has never been priority. Binkley (Binkley, 1998). targeted lines of code for software quantification. (Binkley *et al.*, 2007) towards software refactoring is considered as landmark in our research work. (Weiser, 1981) (Sward *et al.*, 2004).

- Importance of Time and Cost in software maintenance was always ignored.
- Real model development for potential risk forecast was never formed.
- Fragile software design and analysis has complicated the code understanding.

Poor Empirical statistics often results in extra testing and maintenance effort. Reliable code comprehension becomes inevitable to further validate the software. The main motivation of this research is to develop an appropriate code refactoring model with the combination of quantitative measures and static inspection of software code. Initial phase of our work is subject to general aspects software code then it evolves for structured and object oriented features too. Our study establishes the fact that Software metrics statistics can be used to achieve Code Optimization and Re-engineering objectives. Analysis and Application of software code set the core domain of this research. Programming loopholes can't always be reason to deteriorate the overall code structure. It is a law of software evolution (Arnold, 1986). Software Design is not judged on predefined criteria. Implication of good software design is constraints dependent. Consequently, repeated modifications are carried out in order to insure optimum level of constraints satisfaction. Such modifications become restrict integration of new constraints and leave no room for further optimization. Successive changes become major cause of structural decay of code thus adversely affecting the functional requirements too. The major objectives of this research are outlined below:

1. Propose a restructuring methodology that is objectively focused to improve overall cohesion measure of software systems.
2. Use backward tracing strategy of output variables which could be faster and precise.
3. Validate the proposed idea experimentally with available automated tools support.
4. Explore the futuristic dimension on the basis of results obtained.

2. THEORITICAL FOUNDATION

In this section, we briefly outline the key concepts of restructuring, cohesion and slicing, particular in context of our study and their effective application towards software quality improvement.

2.1 Dynamics of Cohesion

Software Engineering as a recognized discipline covers broad spectrum of quality assurance activities

++Corresponding author: M. K. SHAIKH engr_mohsin@quest.edu.pk

*Institute of Mathematics and Computer Science, University of Sindh, Jamshoro,

like quantification, measurement, testing, consumer acceptance and prototype re-engineering. Descriptive analysis of process, product and people can form the basis of adequate and smooth software development life cycle. Elimination of frequent complexities in software should be focal objective of good software practitioner. Software metrics have been pivotal scale to detect and improve quality measures. Software metrics are mainly used for quantification of certain code characteristics. An analytical and empirical analysis shows that the design-level measures correspond closely with code-level cohesion measures.

2.2 Application of Program Slicing

In today's economically motivated world, software systems are backbone of all the business operations to insure swift and secure Since the inception of software engineering, maintenance of source code has remained concerned area for quality assured software system. Program slicing is known as an effective technique to understand the source code in decomposed structure (Binkley *et al.,* 2007) Application of slicing the program is aimed at extracting out source code parts that having defined functionality and exclusive existence. Program slicing essentially focuses to identify portion of a source code having meaningful semantics with particular slicing criteria.

There has been significant advancement to explore different aspects and applications of program slicing over the years and is still ongoing in research literature of software engineering. Program Slicing has emerged as utility component in restructuring and refinement of program code to understand technical dimensions of software systems (Ishio *et al.,* 2003) Application of program slicing is covered in major areas of software engineering, e.g., debugging, testing, software maintenance, safety and re-engineering. Program slicing mainly simplifies large source code by eliminating those parts of the program which are meaningless for particular context. The main aim of program slicing is to identify and extract relevant parts of a software program from a more complicated code. Being able to extract these subprograms and view the source code makes it possible to identify a wide range of potential bugs and thus make the software run with more efficiency (Karhu, *et al.,* 2009)

2.3 Software Restructuring

Software restructuring is the process of re-organizing the logical structure of existing software systems in order to improve particular quality attributes of software products (Kataoka *et al.,* 2002). Some examples of software restructuring are improving coding style, editing documentation, transforming program components (renaming variables, moving expressions, abstracting functions, so on), and enhancing functional structures (Relocating functional components into other or new modules).

Noncompliance with coding standards, improper documentation, and inappropriate development coordination often result in inadequate source code structure (Kataoka *et al.,* 2002) Even the structure of well-designed software tends to deteriorate due to maintenance hazards (Shaikh *et al.,* 2016) The restructuring of old and new software systems can potentially make them easier to understand, comprehend and allow reuse for future operational purposes. Other savings include reduced maintenance costs, increased. Component reuse, and extended software lifetimes (Kataoka *et al.,* 2002) software systems are generally far too large and complex to be effectively restructured on an ad hoc basis. Analysts need rigorous techniques and tools to restructure large software systems. 60% of total software cost is incurred over evolution process during entire life cycle development of software's (Gallagher *et al.,* 1991) Program Comprehension consumes 50-90% of evolution process (Saleem, *et al.,* 2009) (Mohsin and Kaleem 2010) Complicated and multiple tasks within the function make its code ill structured and beyond casual comprehension. In such situation, maintaining the good quality throughout development process becomes very tough and objective of high priority. Over the time, upgrading of technological and customer's needs can enforce modification even in well-designed software system. As a result, software fails to retain its originality in terms of its operation and structure.

Program restructuring is an integral part of software evolution to improve deteriorated structure and to cover the maintenance overheads. Transformation of poorly design software to well-designed software is usual practice in today's software engineering (Gallagher *et al.,* 1991). Restructuring in early days only focused on data and control flow of code but now its application dominates entire software code (Karhu, *et al.,* 2009) Developing the cohesive bond within function by integrating its relevant components is challenging task. However separating the unrelated fragments from same function is even more challenging.

3. RESEARCH THEME

In order to validate theoretical perspective of proposed idea and determine whether backward slicing can be effective software restructuring, experimental study is carried out.

3.1 Describing Approach

We have established theoretical framework and introductory base for our approach. Now we can elaborate its practical dimensions with effectiveness

towards domain of Software quality assurance. Comprehension Process for large code base is very difficult. Experts in program analysis divert their focus on selection function or inter related components of functions from entire code inspection techniques. Such practice does not only help in comprehension but also guide towards developing different metric attributes for further quantification. In this thesis we have set out enhanced cohesion to be foremost criterion for code restructuring. We utilized sophisticated technique called backward slicing to analyze the dependencies on output variable. Below we explain different dimensions of approach.

3.2 Enhancing the Cohesion

For enhancing the cohesion, precise analysis of code and evaluation of dependencies is foremost objective. To properly evaluate the cohesiveness of software, dependent instance variables should be identified from module, and implicit interaction via dependent variables should be considered in computing the cohesion metrics. For example consider code shown in **(Fig. 1).**

```
proceedure sale_pay_Profit

 (days: integer; cost: float;
  var sale: int_array;
  var profit: float;
  var process: boolean;
  var i,j: int; total_sale, total_pay: float;)

  begin
   i:=0 while i>days do begin i:=i+1
   readln(sale[i])
  end;

 if process=true then begin
    total_sale=0; total_pay=0;
 for J:=1 to days do begin
 total_sale=total+sale[i];
```

Fig. 1. Example of a module with lack of cohesion

This program as its name suggests computes the sale the pay and profit which are its eventually instance variable of module. Program structure suggests that instance variables or modules have complex dependencies among them.

One problem with software development is that frequently modules continue to be in state of flux beyond design phases. Redesigning complete system would be disastrous task to handle poor software design. Hence static analysis of software systems can be important approach towards maintenance and cohesion enhancement. Cohesion for this program can be improved if it is restructured using following objective.

- Slice the program into main computation and partial Computation.
- Each task should be isolated for the identification of objectives.

3.2 Backward Slicing

We suggest Backward Slicing as analysis mechanism for achieving our objective of cohesion improvement in software systems. Backward slicing can assist a developer by helping to locate the part of program which needs to re-structure. This technique is simple version of original program with some parts left isolated. An important property of backwards slice is that it preserves the effect of the original program on the variable chosen at the selected point of interest within the program. This is the reason we adopt backward tracing process of dependencies. On the other hand forward slice traces the program parts which get affected by modification. Effects of forward slice are useful after program has been subjected to certain changes so forward slice rather helps regression testing and refactoring.

```
proceedure Sale_Pay_Profit

      (days: int; cost: float:
       var sale: int_array; var process: boolean;
       var pay: float; var profit: float;
       var i, j: int; total_sale, total_pay: float)

       begin
       sale: read_Input((days, sale);
       if process= true then
       total_pay:= compute_Pay(days,sale);
       total_sale:-compute_Sale(days, sale);
       end
```

Fig.2. Slicing the module

We will continue our analysis over program in Fig. 1. Backward slicing the on *Sale_Pay_Profit* on criteria of instance variable, we split the program into four modules.

Compute_pay(days,sale), compute_sale(days,sale). compute_avg_Pay(total_pay,days)" ,compute_profit(total_sale) as shown in **(Fig. 2).** Decomposition of program indicates overall improvement in comprehension of code. This module has dependent instance variables which exhibit cohesive bond with each other. Back ward slicing works as core functionality of our restructuring methodology. However yet transformation of code is required to reach ultimate goal of strong cohesive bond within the modules.

3.2 Transformation for restructuring

In this paper, we develop the transformation to restructure the software systems. We assume that program code can be refactored on prime criterion of enhanced cohesion and dependencies can be analyzed using backward slicing.

```
proceedure Sale_Pay_Profit

        (days: int; cost: float:
         var sale: int_array; var process: boolean;
         var pay: float; var profit: float;
         var i, j: int; total_sale, total_pay: float)

         begin
         sale: read_Input((days, sale);
         if process= true then
         total_pay:= compute_Pay(days,sale);
         total_sale:=compute_Sale(days, sale);
         pay:=compute_avg_pay(total_pay, days);
         profit:=compute_profit(total_sale,cost);
         end
```

```
function compute_sale

 (days:int; var sale: int_array; var j: int
    total_sale:int)

    begin
        total_sale=0;
        for j:=1 to days do
          total_sale:=total_sale+sale[i];
          return(total_sale);
   end
```

```
function compute_pay

 (days:int; var sale: int_array; var j:int)

    begin
        total_pay=0;
        for j:=1 to days do
          total_pay:=total_pay+0.1*sale[i];
        if sale[j]>1000 then
          total_pay:total+50;
          return(total_pay);
   end
```

```
function compute_profit

 (days:int; var sale: int_array; var j: int;
    total_sale:int; total_cost:int;
      var cost: int_array; total_profit: int;)

    begin
        total_sale=0;
        for j:=1 to days do
          total_sale:=total_sale+sale[i];
           total_cost:=total_cost+cost[i];

        total_profit:= total_sale-total_cost;
        return (total_profit);
   end
```

```
function compute_Avg_Pay

 (total_pay:int; days:int; var avg_pay=int;)

    begin
        total_pay=0;
        for j:=1 to days do
          total_pay:= total_pay+ compute_pay(j);

        avg_pay:= total_pay/days;
        return (avg_pay);
   end
```

Fig 3. Decomposed functions

Restructuring process in our approach is performed on following standards.

- Identification of modules with complex dependencies and lack of cohesion.
- Tracing dependencies using backward slicing mechanism.
- Splitting the program code into more refined structure

(Fig. 3) represents the complete restructured code on our suggested approach. Transformation resulted into formation of 5 modules i.e., *Compute_Sale_Pay_Profit, Compute_Sale, Compute_Profit, Compute_pay, Compute_Avg_pay*. Every module is separate and integrated in terms of function and structure. Program restructuring is an important option in software evolution in order to improve structures that have deteriorated and to keep software maintenance costs under control. It is also used in software development to turn a poorly designed program into a well-designed one. The early days of restructuring efforts focused on making a program's control flow easier to follow. This category is quite mature (Karhu, *et al.*, 2009). With regard to functional structure, however, one challenge of restructuring is how to meaningfully group related code segments together inside a large or poorly structured function to form small or cohesive functions, because it is not uncommon that unrelated fragments and functionally cohesive code segments are interleaved in practice.

5. EXPERIMENTAL ANALYSIS

The purpose of carrying experimental analysis is to determine the efficacy of proposed approach over open source software systems. This will further consolidate its application and research worth.

5.1 Subject Systems

There are mainly five well-known open source projects utilized in our study to investigate effectiveness of back-ward slicing. These include JDT Core-3.4, Lucene -2.4.0, JEdit-4.3. Ant-1.7.

Table 1 Descriptive Information of Data-Sets

System	Version	# Methods	KLOC
JDT-Core	3.4	18046	277
Lucene	2.4	7945	125
Ant	1.7	13488	207
JEdit	4.3	4567	137

JDTCore[i] is an Eclipse plug-in that implements the Java infrastructure of the Java IDE. Lucene[ii] is a software package from apache distribution, used for full-featured text search engine library. JEdit[iii] is programming text editor for writing java source code. Ant[iv] is command line tool used for java build files. (Table 1) includes descriptive information of subjects systems used for experimenting our approach. It can well be inferred that all the systems are of reasonable and manageable size in terms of source code structure and number of methods.

5.2 Methodology

We opted for automated tool support to conduct our experiment, which may in turn yield precise, accurate and cost effective result. Our experimental setup consists of Integrated Development Platform: Eclipse 3.1.2, Analysis Tool Indus[1] 0.8.3.14, Slicing Tool Kaveri 0.8.3.7, Eclipse Metrics[2] Plugin Java 6. In order to obtain productive results of our approach, we have set comparative analysis of experiment between Baseline metrics and slice-based metrics. Following is the description of metrics.

Table 2 Baseline Metrics

Metric Name	Description
LCOM	Lack of Cohesion of Methods: Describes the extent to which methods in the source are lacking the cohesive bond.
CC(complexity)	McCabe Cyclomatic complexity is Indication of Complex structure formed due to loops and decision making statement within the source code.
Ca	Afferent Coupling exhibits dependence of source components on classes from other modules.
Ce	Efferent Coupling shows the dependence of classes on source code components of other modules.

Table 3 Slice Metrics

Metric Name	Description
Coverage (COV)	Ratio of means slices to modules within the source code
Overlap (OLAP)	Interdependence of slices
Tightness (TGT)	Cohesive bond among the slices of module

(Table 2 and 3) summarizes the definitions of *Baseline* metrics and Slice based metrics. It is worthwhile to note that metrics described in **(Table 2)** mainly outlines structural properties of source code. Table 3 enlists metrics which actually represent cohesive nature and properties of slices obtained from source code.

[1] http://indus.projects.cs.ksu.edu/projects/kaveri.shtml
[2] http://metrics.sourceforge.net/

5.3 Application

Firstly, we conducted experiment over open source software Interpreter.java[3] to assess the application of our proposed technique. Interpreter.java contains 6 classes, 33 methods and 715 total lines of code. Initial review of experimental study led to following major observations.

- 5 classes exhibit normal metric values including Lack of Cohesion on methods and McCabe Cyclomatic Complexity.
- Biggest class of systems contains 700 lines of code and 33 methods
- Restructuring of methods was carried out on priority basis of determining complexities
- Restructuring mainly focused to restructure the functions with high complexity values on the criterion of enhancing cohesion.

Table 1 mainly indicates quality values of subject system. Before restructuring process, software bears Lack of Cohesion value 0.948, which is out of range as calculated by Hellen's Formula.

Table 3. Before Restructuring

Metric	Value
LCOM	0.948
CC	0.56
Ca	0.68
Ce	0.43

Overall complexity of class structure is also not feasible, hence, functions within that class need decomposition. Generally 33 methods within class is not a big problem but slicing statistics may get affected as shown in **(Table 2).** Consequently restructuring process becomes applicable to subject system for improvement in code quality. We identified functions with critical values of subject system and brought them under restructuring process. The decomposition results in enhanced cohesion of overall structure program. Function print was also sliced to reduce complex structure. **(Table 4)** shows properties of restructured system. It is evident from the results that restructuring process improved the Lack of Cohesion and reduced its value up to approximately 6%.Overall cyclomatic complexity of class 30% which is indication of efficient code transformation.4 new functions were created using decomposition process.

[3]https://sites.google.com/site/smallbasicinterpreters/source-code

Table 4 Metrics Statistics after Restructuring

Metric	Value
LCOM	**0.89**
CC	**0.49**
Ca	**0.56**
Ce	**0.39**

5.4 Result

In this section, we have represented the results using box-plots for different open source software systems. These graphical representation of subject systems is obtained after backward slicing and their source code is analyzed using research methodology described in section 5.2.

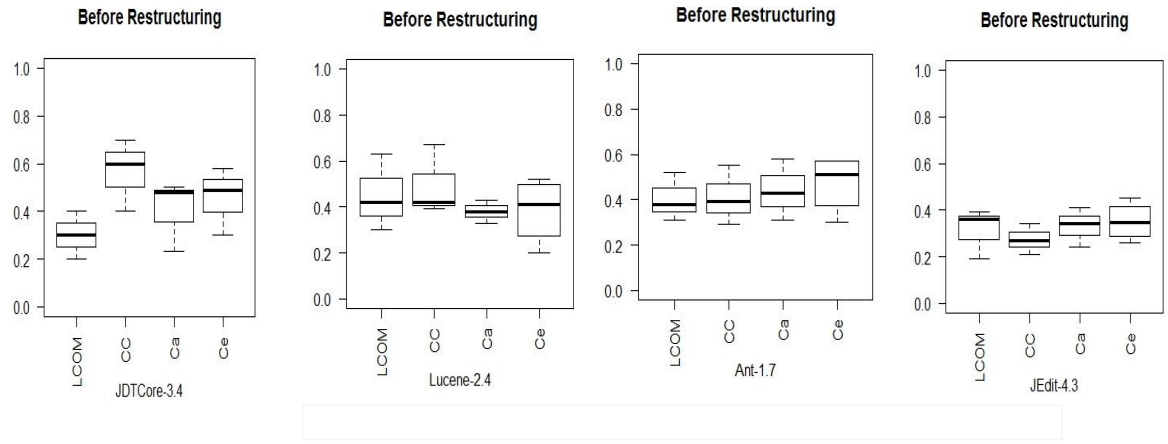

Fig. 4. Subject Systems metric values Before Restructuring

(Fig. 4) represents box-plot distribution of process metrics for subject systems used in this study. These metric values characterize the software systems by certain properties of complexity and design strength. It can be well be observed that LCOM is almost persistent in all the subject systems whereas CC is high as well except JEdit-4.3 It indicates that complexity and design of software systems are not adequate, so, they are subject to restructuring.

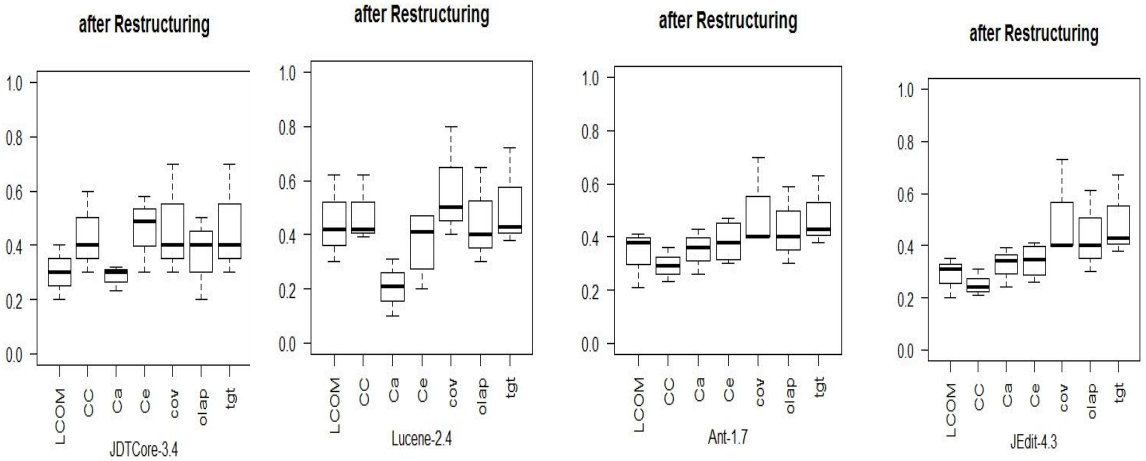

Fig. 5. Subject Systems metrics values After Restructuring

(Fig. 5) represents the process metric values and slice metric values after applying back-ward slicing. All the systems have generally shown significant improvement with the application refactoring effort: LCOM shows decreased mean value in Lucene, Ant and JDTCore, Ca, and Ce seem to be decreasing after restructuring in Ant and JEdit too. Interestingly, metrics related slice cohesion are in improving and increasing trend in JDTCore-3.4 and Lucen-2.4, where median values of COV remain 0.4 to 0.7, TGT values exhibit significant change due to restructuring. In summary, cohesion metrics values have shown substantial improvement as result of restructuring on the basis of backward slicing.

6. CONCLUSIONS

Software restructuring should be easy, efficient and semantic preserving. We provide software restructuring frame work where enhancing cohesion is prime objective using backward analysis of code statically. Restructuring process and operations gives dynamic mechanism for optimizing improvements in cohesion. A software system that has gone through this restructuring process has higher quality and is more maintainable. The return on investment in this refactoring process can be measured in lower error rates, fewer test cases per module, and increased overall under stability and maintainability.

These results show that restructuring enhanced cohesion and decreased Cyclomatic complexity. It is also evident from the results that original functional core of program is preserved after restructuring the system. Despite the fact that Software engineers need calculated and economical mechanisms to deal with code complexity. There is no commercial acceptability to new approaches. Our future research will include detailed empirical study to further validate implementation of our methodology on commercial systems.

REFERENCES:

Arnold R S. (1986) An introduction to software restructuring. IEEE Computer Society Press, Washington, DC;

Binkley D. (1998). The application of program slicing to regression testing. Information and software technology. 1; 40(11):583-94.

Binkley D., M. Harman J. Krinke (2007) Empirical study of optimization techniques for massive slicing. ACM Transactions on Programming Languages and Systems (TOPLAS). 1; 30(1): 3-6.

Binkley D., N. Gold M. Harman (2007) An empirical study of static program slice size. ACM Transactions on Software Engineering and Methodology (TOSEM). 1; 16(2):8-12.

Gallagher K. B., J. R. Lyle (1991) Using program slicing in software maintenance. IEEE transactions on software engineering. (8):751-61.

Ishio T, S. Kusumoto K. Inoue (2003) Program slicing tool for effective software evolution using aspect-oriented technique. In Software Evolution, Proceedings. Sixth International Workshop on Principles of 3-12. IEEE.

Ishio T., S. Kusumoto K.. Inoue (2003) Program slicing tool for effective software evolution using aspect-oriented technique. In Software Evolution,. Proceedings. Sixth International Workshop on Principles of 3-12. IEEE.

Karhu K., T. Repo O. Taipale K. Smolander (2009) Empirical observations on software testing automation. InSoftware Testing Verification and Validation,. ICST'09. International Conference on 201-209. IEEE.

Kusumoto S., A. Nishimatsu, K. Nishie K. Inoue (2002) Experimental evaluation of program slicing for fault localization. Empirical Software Engineering. 1; 7(1):49-76.

Kataoka Y., T. Imai H. Andou T. Fukaya (2002) A quantitative evaluation of maintainability enhancement by refactoring. InSoftware Maintenance,. Proceedings. International Conference IEEE. 576-585.

Lakhotia A., J. C. (1998) Deprez. Restructuring programs by tucking statements into functions. Information and Software Technology. 40(11):677-89.

Mohsin S., Z. Kaleem (2010) Program slicing based software metrics towards code restructuring. In Computer Research and Development, Second International Conference 738-741. IEEE.

Saleem M., R. Hussain, V. Ismail S. Mohsin (2009) Cost effective software engineering using program slicing techniques. In Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, 768-772. ACM.

Sward R E., A. T. Chamillard D A. Cook (2004). Using Software Metrics and program slicing for refactoring. Air Force Academy Colorado Springs Co;

Shaikh M., C. G. Lee (2016) Aspect Oriented Re-engineering of Legacy Software Using Cross-Cutting Concern Characterization and Significant Code Smells Detection. International Journal of Software Engineering and Knowledge Engineering. (03):513-36.

Weiser M. (1981) Program slicing. In Proceedings of the 5th international conference on Software engineering Mar 9 439-449. IEEE Press.

Xu B., J. Qian X. Zhang, Z. Wu L. Chen (2005) A brief survey of program slicing. ACM SIGSOFT Software Engineering Notes. 1; 30(2):1-36.