An experimental examination of the role of re-engineering in the management of software quality

E. Georgiadou, G. Karakitsos, C. Sadler, D. Stasinopoulos
Faculty of Science, Computing & Engineering, University of North London, 2-16 Eden Grove, London N7 8DB, UK

Abstract

This paper reports on the design and the results of a randomised, controlled experiment estimating the effect of predetermined changes in module complexity on the maintainability of different program versions seeded with equivalent logic errors. The experiment measures maintainability which is a defining sub-attribute of quality. The hypothesis *"low module complexity results in high maintainability"* is tested experimentally by monitoring and recording the time taken to identify and correct the seeded errors. Prior to the experiment programs are statically analysed to produce measurements of internal sub-attributes of the fundamental attribute of complexity .A first program version is modularised according to established rules giving a new version with a larger number of modules and with smaller individual module complexity. The results of this work can be used to design tools capable of providing an indicator, or factor, for re-engineering whereby a given program can be restructured in such a way that quality improvement can be quantified or at least estimated. As maintainability is a defining attribute of quality the insights gained can be further applied in understanding the underlying processes involved and ultimately lead to quality improvements.

1. Introduction

There is frequent talk ranging from simple observation to formal study of what came to be known as the software crisis. Late delivery, inaccuracy, low reliability, high cost of maintenance, bad safety record are some of the manifestations of this crisis, Beizer [1], Fenton [2], Sommerville [3]. All these problems can be summed up by the all encompassing notion of quality - bad quality in this case.

When these problems arise there are two options namely that of maintaining (by corrections, adaptations etc.) the existing software and that of developing completely new solutions. Either of these routes have proved to be extremely costly. In recent years a third avenue has been gaining increasing interest that of software reuse a much envied capability of spare parts usage by engineers and manufacturers of other products.

However, unless the software were built for re-use in the first place it is usually necessary to re-structure it. Although code has always been reused, it is only recently that the necessity and benefits of a rigorous approach to manage the process of re- engineering have been identified and explored , Basili et al[4], Biggerstaff [5], Yourdon [6], Shaw [7], Basili [25].

This work is an attempt to understand the role of re-engineering in the management of product quality. It makes use of automated re-structuring tools Karakitsos et al [8] and endeavours to define and estimate a *re-engineering factor* which in turn will demonstrate whether it is beneficial to re-engineer a given piece of software or not , Sommerville [9,10]. This work aims to identify the characteristics of software that will determine whether it is possible and beneficial to carry out re-engineering of code.

Section 2 examines complexity in terms of some of its measurable sub-attributes. It identifies criteria for modularisation and gives a brief description of the in-house analysis tools that provide the indicators. Section 3 presents the central hypothesis of this work, as well as the design, the preparation and the conduct of the experiment. Section 4 presents a summary of the experimental results and section 5 gives the conclusions and the future directions.

2. Internal attributes and their measurements

2.1 Complexity

In order to restructure software it is necessary to define it in terms of its attributes. Researchers and practitioners have identified numerous attributes and sub-attributes of software and produced taxonomies in their effort to understand the functionality and behaviour of software, Fenton [2], McCall [9], Boehm [10]. Fenton [14] classifies attributes into external and internal and this is the classification adopted by the authors.

Fenton says that "... all external attributes are complex and thus not directly measurable". *Quality* is the external attribute that encompasses the totality of external software attributes. *Maintainability* and *reliability* are two sub-attributes of quality. They are also complex and thus not directly measurable. Although external attributes cannot be defined in terms of the code it has been observed that they are related to some internal attributes, Troy [11], DeMarco [12]. Thus studying and measuring the internal attributes can provide an ***assessment*** of the external ones.

The central hypothesis of this work is that the maintainability of a program is affected by the complexity of its modules. *Complexity* is a term that encapsulates several internal attributes. Therefore complexity is a synthetic concept and as such it cannot be measured directly but measurable sub-attributes can provide an indicator of complexity.

2.2 The program as a whole

Each program is made up of modules and possesses size which can be measured as length or total number of statements. Modules are inter-connected by virtue of module calls and parameter passing. The depth and width of the hierarchy of calls as well as the density of calls provide the shape or *morphology* of the whole program.

A program can be represented as a network and/or a hierarchy of calls. Both representations provide a picture of the overall structure often referred to as the program morphology and reveal areas that may generate problems such as high density of calls and 'large' depth of call.

Global coupling is a measure of the degree of interdependence between modules. It is desirable for modules to have low coupling, Fenton [2], Sommerville [3]. Further investigations will deal with inter-modular attributes and program-wide issues.

2.3 Individual modules

2.3.1 The attributes of individual modules Individual program modules possess attributes such as *granularity* i.e. the number of statements or size, *McCabe* complexity, *cohesion* and *coupling*.

The McCabe complexity is a well quoted measure which, despite its limitations Troy [11], DeMarco [12], has served as a decisive indicator of innate code complexity since its original inception ,McCabe [13]. *Cohesion* is the module's functional strength i.e. the extent to which the individual module components are needed to perform the same task. It is preferable for modules to have functional cohesion and for the same reasons undesirable to have coincidental cohesion, Gilb [14].

Although there is no universally accepted definition of *module complexity* several measures involving the fan-in, fan-out counts were considered including the Henry & Kafura [15] measure as well as the refinement by Shepperd [16].

In this paper the information flow into and out of the model can be represented by the total number of parameters of the parameters in the parameter list together with global references.

2.3.2 Modularisation There are three types of module lending themselves as candidates for modularisation. Firstly there are the modules with high modularity (> 50 statements), secondly modules with a high McCabe metric (>10) and thirdly modules that contain repetitions of common code.

Figure 1: Factor Analysis

```
ROTATED FACTOR LOADINGS (PATTERN)
---------------------------------

                        FACTOR     FACTOR
                           1          2

granularity    1        0.559      0.299
halstead (E)   2        1.015     -0.039
McCabe         3        0.437     -0.034
local_ids      4        0.749      0.284
global_vars    5        0.042      0.784
num_of_calls   6        0.892     -0.122
info_flow      7       -0.005      0.108

              VP        2.890      0.814

THE VP FOR EACH FACTOR IS THE SUM OF THE SQUARES OF THE
ELEMENTS OF THE COLUMN OF THE FACTOR PATTERN MATRIX
CORRESPONDING TO THAT FACTOR.  WHEN THE ROTATION IS
ORTHOGONAL, THE VP IS THE VARIANCE EXPLAINED BY THE FACTOR.
```

Several programs were statically analysed in order to identify the areas that needed attention. In addition, Factor Analysis was carried out in order to establish whether some attributes were measuring the same or similar property. In fact, earlier results obtained by Khoshgoftaar et al [24] were confirmed by this analysis which also revealed that the Halstead, McCabe and Granularity measures cluster together (factor 1). In general they measure the module 'size'. Another cluster (factor 2) shows that the information flow and the number of local identifiers measure the size of the variable environment. Figure 1 shows the Factor Analysis of a program with 99 modules.

```
                     ROTATED FACTOR LOADINGS

             ....+.......+.......+.......+.......+....
             -                                       -
         1   +                                       +
             -                                       -
             -                    5                  -
    F        -                                       -
    A        -                                       -
    C        -                                       -
    T        -                            1  4       -
    O        -                   7                   -
    R    0   +                          3            H
             -                                 6     -
    2        -                                       -
             -                                       -
             -                                       -
             -                                       -
             -                                       -
             -                                       -
        -1   +                                       +
             -                                       -
             ....+.......+.......+.......+.......+....
                       -.50             .50
               -1.0             0.0             1.0

                             FACTOR 1

              VARIABLES ARE DENOTED BY 1,..., 9, A,..., Z
              OVERLAPS ARE DENOTED BY AN ASTERISK.
```

Obviously as the number of modules increases the overall structure of the program changes and the structuredness increases. It can be seen that blind modularisation may lead to increased overall complexity. Beizer [1] proved that for one call of the removed (common) code the overall complexity increases.

It is possible to make some judgement as to whether a particular version of a program is less complex than another version. Four attributes were considered as suitable for providing such an indicator. These were the McCabe complexity, the number of local identifiers, the granularity and a measure of information flow .

The averages of these measures are presented in a star plot , Neil [17] for each version. A very compact star plot indicates a more structured and therefore less complex program. In general, modules with high granularity will contain a large number of identifiers and will have high a McCabe measure resulting in a star plot of larger area. It is possible but rare for the star plot of the re-structured version to occupy a larger area. This happens when the first version is one straight set of program statements (i.e. one module only) therefore passing no parameters.

Provided that the underlying complexity remains the same (i.e. the program versions are solving the same problem) the star plots shown in section 2.4.3 reveal that the restructured versions of the programs under study have lower complexity than the original versions.

2.4 Tools for static analysis

2.4.1 Multi-language Translator : (Static Analyser Module) In-house tools were developed as part of a larger re-engineering research activity at the University of North London which proved very useful in providing measurements of code. The particular module used was MLTSAM, Karakitsos et al [8].

MLTSAM is a modular tool for static analysis. Each module is being built on top of an EBNF parser. It aims to provide resource and re-engineering facilities, together with a powerful language for abstracting and transforming the automatically produced concrete syntax tree representation.

The first version of each program under study was analysed and measured revealing the areas needing re-structuring. The code was then re-engineered by modularising, reducing local (module) complexity and granularity. In each case further modularisation was carried out on the module with the largest granularity (usually with the largest McCabe as well), Rombach [26,27]. Among many programs that were analysed one was a fairly large program with 99 modules. One module had a McCabe measure of 259!. In fact this is a special module containing an extremely large CASE statement carrying out the parsing of code, generated by YACC, describing the BNF rules and actions of the C language analyser.

2.4.2 The measures The obtained program measures are listed in a comparative table (Table 1). For each program four measurements were selected to generate a star plot. The two versions in case produced a drastically different plot giving a strong indicator of variations in complexity following modularisation.

Table 1 : Static Measures

	p1v1	p1v2	p2v1	p2v2
No of statements	65	52	91	85
No of modules	2	8	1	10
Total no of calls	1	11	0	18
Maximum depth of call	2	3	1	3
Maximum Width	1	6	1	6
Halstead Effort (x 1000)	30	5	50	21
Total McCabe Comp.	16	18	26	28
Maximum McCabe	13	3	26	10
Average McCabe	8	2	26	3
Average granularity	33	7	91	9
Av. Information flow	14	7	1	2
Av. num. of local variables	6	2	17	2

2.4.3 The star plots The star plots shown in figures 2 and 3 indicate that at least for the programs under study the restructured versions (shaded area) possess smaller average module complexity. Further work is needed on the weighting of the attributes so that appropriate scaling can be adopted.

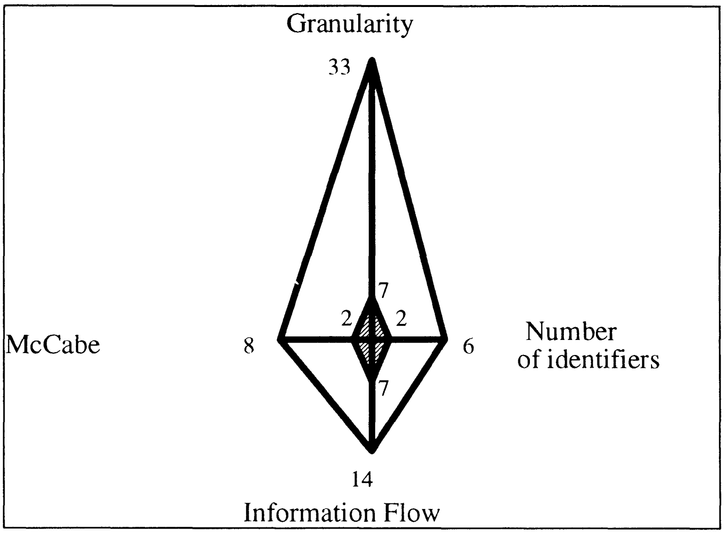

Figure 2: The star plots for the two versions of Program 1

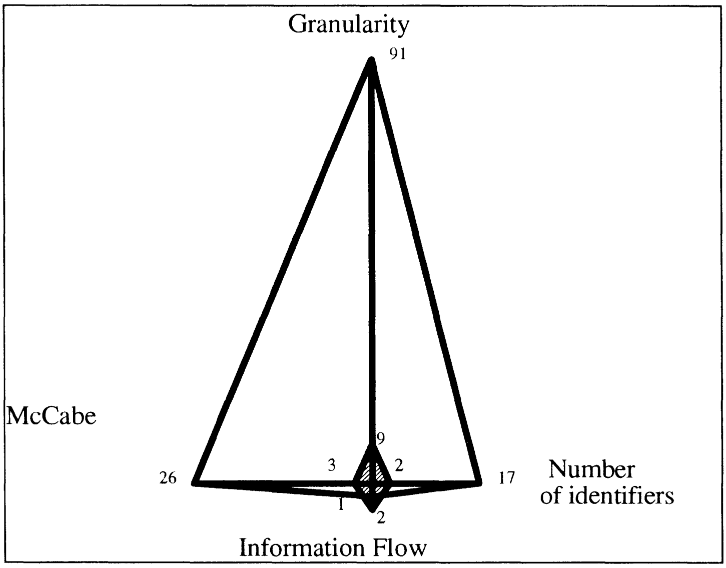

Figure 3: The star plots for the two versions of Program 2

3 The experiment

3.1 Background

Formal experimentation is not very widespread in Software Engineering, Basili [18], Perlis et al [19], Mohamed at al[23]. The few examples have not thrown much light on re-engineering. From the re-engineering point of view it is desirable to ascertain to what extent modifications in internal attributes can affect quality attributes.

It was decided to carry out a series of experiments in order to test the belief that alterations in the internal attributes will affect the external attributes. The experiment reported here is the first one in the series.

3.2 The hypothesis

Low complexity will be ascertained with reference to measures described in section 2 and maintainability will be assessed by recording the time taken to identify and correct errors previously seeded into programs. An error manifests itself in deviations from expected behaviour. Four types of non-syntactic error were used namely :

(a) assignment
(b) iteration (FOR loop)
(c) array
(d) Boolean expression

These were seeded inside the two versions of the code. The hypothesis supposes that an error which is seeded inside linear, unstructured code (i.e. more complex at the module level) cannot be identified and/corrected as quickly as in the case of a similar error seeded in well-structured , less complex code.

3.3 The experimental design

The experimental subjects at this stage are students on the University's Post-graduate Diploma in Computing. They are all novices in programming in "C" although they passed a programming unit in Modula-2.

In order to ensure validity, interpretability and accuracy of the results certain precautions had to be taken so that other factors such different experience of the students in other programming languages would not invalidate the results. To achieve that the statistical methods of blocking, replication and randomisation were used , Das & Giri [20], Basili [18].

Students were allocated randomly to four different groups (A, B, C, D). Each group consisted of nine subjects (replicates). These groups together with the two different programs were treated as blocks for the experiment. The treatment factor is the different versions of the two programs differentiated by module complexity according to the principles outlined in section 2.3.2. The design made use of the guidelines provided by the DESMET Experimental Framework and in particular the EXPDA module, Mohamed [21]. The design is shown below :

TABLE 2: Cross-over design

p1v1 - program 1 version 1-	(high complexity)
p1v2 - program 1 version 2	(low complexity)
p2v1 - program 2 version 1	(high complexity)
p2v2 - program 2 version 2	(low complexity)

	p1	p2
A	v1	v1
B	v1	v2
C	v2	v1
D	v2	v2

3.4 Preparatory steps

A questionnaire was circulated to all the subjects one week in advance of the experiment in order to ensure that the population was homogeneous as far as previous relevant experience was concerned. This served to establish that subjects were novices in programming in "C", with an average of 12 months in programming in another language (Modula 2), their average age was 33 years with standard deviation equal to four years.

It was decided to capture the activity of the experimental subjects by recording automatically whether they were Editing, Compiling or Running the program. This required the subjects to carry out the maintenance from within a specially designed interface . The time taken to edit the programs is recorded automatically using the INTER interface , Karakitsos, Georgiadou & Jones [22]. The number of errors corrected is also recorded automatically. In addition the subjects were to be instructed to annotate their respective program listings with the appropriate corrections so that further information could be captured in case time ran out or their lack of knowledge of the language was prohibitive.

Other data that can be recorded by the interface include data recorded included the number of times each subject compiled and executed the program.

Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

Some of this data will be used for further studies so that subsequent experiments can benefit in selecting the groups more rigorously.

During the week preceding the planned experiment the subjects were introduced to the interface so that lack of acquaintance with the interface would not influence their performance on the day.

3.5 On the day

Each subject was randomly allocated to a group (A, B, C or D). They had 45 minutes to carry out maintenance on one of the program versions followed by another 45 minutes to maintain the relevant second program version as shown in the Table 2. The subjects were issued with: the following:

- interface user instructions enabling them to copy across to their user areas the appropriate programs;
- a particular combination of program listings under maintenance;
- a brief description of the required outputs;
- a copy of the required outputs (produced by a correct version of the program - which the subjects have no access to);
- a copy of the test data.

4 A summary of the experimental results

4.1 The measurements

A total of 34 subjects participated out of the expected 36. Fourteen of them had missed the preparatory session resulting in some problems with using the interface. This led to some subjects being unable to make satisfactory progress. Twelve of them did not manage to identify any errors, the majority of which had missed the preparatory session. Although the design of the experiment was robust it was not enough to ensure that all subjects would turn up or follow the instructions.

It was decided that the zero hits (successes) should be excluded from the analysis, since they do not contribute any information.

The ratio of time taken to number of hits plotted against the program version is provided the graphical representation shown in figure 4 below.

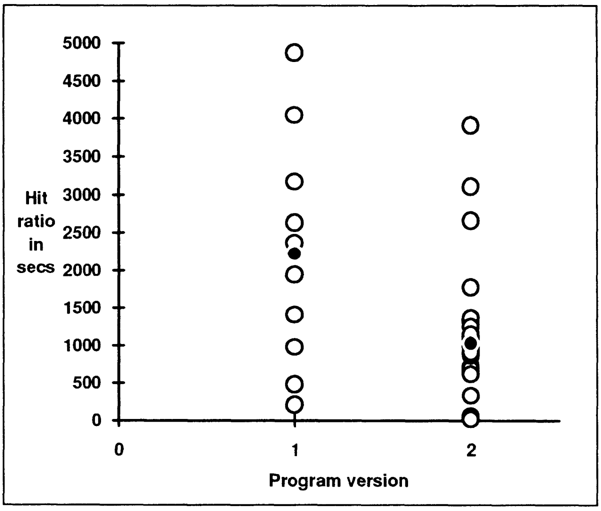

figure 4: Scatter graph of hit ratio against program version

By inspecting the data it was realised that several points overlapped, which was not obvious from figure 4. A second attempt using GLIM produced the following plot (figure 5) which reveals the multiple points.

```
           +---------+---------+---------+---------+---------+
          |                                                   |
          |                    .                              |
          |                                                   |
          |                                                   |
    4000. +                    .                   .          +
          |                                                   |
H         |                                                   |
i         |                    .                   .          |
t         |                                                   |
          |                    2                   .          |
R         |                    9                              |
a   2000. +                    2                              +
t         |                                        .          |
i         |                    .                   2          |
o         |                                        9          |
          |                    .                   6          |
          |                    .                   3          |
          |                    .                   .          |
       0. +                                        2          +
           +---------+---------+---------+---------+---------+
          0.0                 1.0                 2.0
                               program versions
```

figure 5 GLIM plot (hit ratio against program version).

4.2 The interpretation

In the case of the first program (p1v1,p1v2) the original hypothesis was born out by the experimental evidence provided by the differences in the identification and correction rates of the two program versions (figures 4& 5).

There was a significant variation in the mean time taken to achieve a hit (find or find & correct) between versions. The mean for p1v1 was 2224 seconds while for p1v2 it was 1027 seconds a statistically significant difference with a p-value equal to 0.008. The unstructured version was more difficult to understand and therefore the maintenance was not very successful. The results from the second program were less encouraging. Very few subjects completed their experimental tasks leading to a small and biased from which conclusive results are neither possible or desirable.

The reasons leading to the differences in the number of people attempting the maintenance on the second program (both versions) included: absenteeism, reluctance to attempt a second program presumably because of its nature. The first application was data retrieval and the second one was abstract manipulation and counting of text characteristics.

The results for the first program (p1v1, p1v2) maintenance on which was attempted for a longer time overall, were the expected ones. Further data are necessary in order to strengthen the obtained results.

4.3 Lessons learned

As a number of follow up experiments have already been planned it was important to carry out a post-mortem so that the problem areas can be dealt with. A second questionnaire was issued to all participants at the end of the experiment in order to benefit from their evaluation and criticisms. It was felt that the time allocated was very short and this belief was reinforced by the fact that all the subjects who reverted to their first program and persevered recorded more hits.

Conducting the experiment highlighted that the design did not anticipate erratic attendance or the reluctance of the subjects to conform to the requirements. There was no automatic switch to ensure that at the end of the 45 minutes everybody embarked on the maintenance of the second program. In fact some subjects kept swapping from program to program presumably as they encountered difficulties with the other one!

The post mortem dealt with the inadequacies of the interface which is currently being updated. Finally it was gathered that experiments that involve human beings carry a large unpredictability of behaviour and the design of the experiment must be robust and have in-built contingency plans.

5. Postscript

5.1 Conclusions

Re-engineering of code is necessary in particular prior to reuse. The facilities offered by the automated static analyser highlight the possible areas for further modularisation such as large granularity, large McCabe or repeated code.

Modularisation of a program results in another version which is usually made up of a larger number of smaller modules. The underlying functionality remains the same. The differences between the two versions can be graphically represented in a star plot which provides an indicator of the overall complexity. The more compact (smaller area) the star plot the smaller the complexity.

The identification of errors is expected to be more difficult for program that are unstructured. The results of the pilot experiment were encouraging but further evidence is necessary.

An estimate of the re-engineering factor can be obtained from a star plot or other representation of the static measures (or a representative subset of the measures as in this experiment). In fact if the star plot of original version of the program is represented as a square in most cases the restructured program will have a star plot of smaller area inside the square. The difference in the two areas provides a measure of the improvement achieved through re-engineering.

Ultimately the knowledge gained can be applied for the improvement in the quality of software. These attributes can be measured in a predictable, controllable and therefore repeatable fashion.

5.2 Future directions

Following the first experiment additional functions and refinements are being added to the INTER interface in order to enforce stricter conditions during the conduct of the experiment and thus enhance the reliability of the data collected. Further investigations are planned for inter-program comparisons in order to establish whether there are universal and/or predictable relationships between the measured internal sub-attributes such as the number of local identifiers and the modules' granularity.

Additional experiments are planned within the University of North London with other groups of students. It is also hoped that the same experiments can be repeated within the environment of industrial partners with professional programmers and real life applications.

It is planned to further develop the tools in order to enhance the Static Analyser and to add a Dynamic Analyser. It is hoped that the integrated tools can be used for source code re- engineering.

As a result of the insights gained into software attributes of significance to reverse engineering, it is hope that it will be possible to formulate taxonomies of internal and external attributes and define rigorously the effects of the internal domain onto the external one. Ultimately it is hoped to re-examine the development methodologies whereby the original software was produced and to propose appropriate strategies which will result in the improvement of quality.

Acknowledgements

The authors would like to thank all the students who participated in the experiment and all the colleagues who helped with suggestions and ideas. In particular, we greatly appreciate the contribution of Ray Jones and Al Murugaiyan during the actual experiment.

References

[1] Beizer, B., 'Software testing Techniques', Van Nostrand Reinhold, 1990.

[2] Fenton Norman,Software Metrics - A Rigorous Approach, Chapman & Hall 1991

[3] Sommerville I. "Software Engineering"(1992)Addison-Wesley

[4] Basili V.R. and Shaw M. "Scope of Software Reuse", White paper, working group on 'Scope of Software Reuse', Tenth Minnowbrook Workshop on Software Reuse, New York, July 1987

[5] Biggerstaff T. "Reusability Framework, Assessment, and Directions", IEEE Software Magazine, march 1987, pp. 41-49

[6] Yourdon E., Constantine L.L. 'Structured Design', Prentice Hall, 1979.

[7] Shaw M. "Purposes and Varieties of Software Reuse" Proceedings of the Tenth Minnowbrook Workshop on Software Reuse, New York, July 1987

[8] Karakitsos G., Danicic S. "M.L.T. A Multi-Language Translator", University of North London, Research Seminar Series,, 1990

[9] McCall J.A., et al., "Factors in Software Quality", RADC-TR- 77-369, General Electric Co., Sunnyvale, CA, July 1977

[10] Boehm PG et al "Characteristics of Software Quality", North Holland, 1978

[11] Troy D.A. and Sweben S.H. "Measuring the Quality of Structured Designs", The Journal of Systems and Software , 2, 113-120, 1981

[12] DeMarco T. "Controlling Software Projects - Management, Measurement & Estimation", Yourdon Inc., 1982

[13] McCabe T.J. "A complexity Measure", IEEE Trans. on Software Engineering, SE-2 no.4, pp. 308-320, December 1976

[14] Gilb T, "Principles of Software Engineering", Addison Wesley, 1987

[15] Henry S. and Kafura D., "Software structure metrics based on information flow", IEEE Transactions in Software Engineering , SE- 7(5), 1981, 510 -518

[16] Shepperd M.J., "Design Metrics: An empirical analysis", Software Engineering Journal 5(1), 1990, 3-10

[17] Neil M. "Multivariate Assessment of Software Products", Journal of Software Testing, Verification and Reliability, 1(4), Jan-Mar 1992

[18] Basili V.R. and Weiss D.M. "A Methodology for Collecting Valid Software Engineering Data", IEEE Transactions on Software Engineering, vol. SE-10, no.3, November 1984, pp. 728-738

[19] Perlis A., Sayward F., Shaw M. (ed), "Software Metrics" the MIT Press, 1981.

[20] Das M.N. & Giri N.C. "Design and Analysis of Experiments", Halstead Press- John Wiley, 1986

[21] Mohamed W.E. "DESMET: EXPDA Experimental Design Analysis Procedures", University of North London Internal Report, 1992

[22] Karakitsos G., Georgiadou E., Jones R. "INTER - An interface for recording experimental data", University of North London - research Seminar Series, 1992

[23] Mohamed W.E. & Sadler C.J. "Methodology Evaluation: A critical Survey", Eurometrics Proceedings, Brussels, April 1992

[24] Khoshgoftaar T.M., Munson J.C., Ravichandran S. "Comparative aspects of software complexity metrics and program modules - a multidimensional scaling approach", Software quality Journal 1, 159-173 , 1992

[25] Basili V.R. "Viewing Maintenance as Reuse Oriented Software Development", IEEE Software Magazine, January 1990, pp.19-25

[26] Rombach H.D. "Software Design Metrics for Maintenance" Ninth Annual SE Workshop, NASA Goddard Space Flight Center Greenbelt, MD, November 1984

[27] Rombach H.D. "Measurement-based Improvement of Software Development", Tutorial, Eurometrics, Paris, April 1991