

# Middlesex University Research Repository

An open access repository of

Middlesex University research

<http://eprints.mdx.ac.uk>

Georgiadou, Elli, Karakitsos, George, Stasinopoulos, Dimitris, Sadler, Chris and Jones, Ray (1994) An experimental examination of program maintainability as a function of structuredness. Software Quality Management II - Volume 2 Building Quality into Software. In: Software Quality Management II: 2nd International conference, August 1994, Edinburgh. ISBN 9781853123535. ISSN 1743-3517 [Conference or Workshop Item] (doi:10.2495/SQM940522)

Published version (with publisher's formatting)

This version is available at: <https://eprints.mdx.ac.uk/26375/>

## Copyright:

Middlesex University Research Repository makes the University's research available electronically.

Copyright and moral rights to this work are retained by the author and/or other copyright owners unless otherwise stated. The work is supplied on the understanding that any use for commercial gain is strictly forbidden. A copy may be downloaded for personal, non-commercial, research or study without prior permission and without charge.

Works, including theses and research projects, may not be reproduced in any format or medium, or extensive quotations taken from them, or their content changed in any way, without first obtaining permission in writing from the copyright holder(s). They may not be sold or exploited commercially in any format or medium without the prior written permission of the copyright holder(s).

Full bibliographic details must be given when referring to, or quoting from full items including the author's name, the title of the work, publication details where relevant (place, publisher, date), pagination, and for theses or dissertations the awarding institution, the degree type awarded, and the date of the award.

If you believe that any material held in the repository infringes copyright law, please contact the Repository Team at Middlesex University via the following email address:

[eprints@mdx.ac.uk](mailto:eprints@mdx.ac.uk)

The item will be removed from the repository while any claim is being investigated.

See also repository copyright: re-use policy: <http://eprints.mdx.ac.uk/policies.html#copy>



# An experimental examination of program maintainability as a function of structuredness

E. Georgiadou, G. Karakitsos, D. Stasinopoulos,  
C. Sadler & R. Jones

*School of Computing, University of North London,  
2-16 Eden Grove, London, N7 8DB, UK*

## Abstract

The general ethos of producing structured programs has been, at least in theory, adopted throughout the software engineering community. By studying and measuring the structure of existing software we can estimate the benefits to be gained from changes in the structure in terms of the external attributes (perceived behaviour) of the re-structured software. [13, 2, 3, 10, 6, 7]. In this paper we report the results of two controlled experiments measuring the improvement on the maintainability of *differently structured* code. These experiments build on the experience and insights gained through an earlier experiment [5]. We discuss a strategy for re-structuring based on an improved re-engineering factor [9] and present the static measures of morphology (depth and width of module calls), coupling and cohesion and module complexity of a range of programs. By plotting these measures and adopting target values (e.g. width of call < 5) we estimate the expected improvement in the maintainability after re-engineering. We subsequently carry out the re-engineering, measure the re-structured code statically and measure the actual maintainability experimentally. The results reveal that unstructured programmes take longer to 'reveal their secrets'. An integral part of this work are the design and execution of controlled experiments as well as the use of automated tools for the static analysis of code and the recording of the experimental data.



## 1. Introduction

In the last twenty years the software engineering community has adopted structured methods for the development of software. However, it is not always obvious or certain that the methods ensure the production of structured programs. Furthermore the degree of structuredness cannot always be measured.

Several attempts have been made to define the attributes of a structured program in terms of the constructs it uses (sequence, iteration, selection). Fenton [6] provided a rigorous definition in terms of S-Structures using only sequence and nesting. When attempting to re-structure code one needs to ascertain the degree of structuredness of the given code as the re-structuring process is both time-consuming and expensive.

In an attempt to gain insights into the structure of software we examined the global attributes of morphology and coupling and we subsequently looked at the cohesion and complexity of the individual modules. We produced the static measures for a variety of programs [12] and carried out re-structuring of the original code according to established rules [7, 1, 9].

In each experiment, both versions of the program (original and re-structured) were seeded with errors and corrective maintenance was carried out under controlled conditions. Automated tools for the analysis and the monitoring of the maintenance activity were used.[14]. The automatically stored data was analysed [8] and conclusions were drawn.

Section 2 of this paper provides the hypothesis under test, and the static analysis of code in terms of global and module-based attributes. It discusses examples of the measures produced. Section 3 deals with the design and conduct of the experiments and section 4 reports the results and their interpretation. Finally Section 5 provides the conclusions and future directions of this research work.



## 2. Re-structuring Software

### 2.1 Strategy for re-structuring

The re-structuring strategy is based on analysing the code in order to identify problem areas such as high concentration of calls and high McCabe complexity measure. The code is then modularised according to established rules [1, 9]. It is expected that structured programs are easier to maintain and it is therefore the expected mean time to find and correct an error in a structured program should be smaller than the time taken for an equivalent unstructured version.

The rationale is based on the need to measure simple (internal) software attributes in order to estimate and measure the more complex ones (external attributes) such as maintainability. Measuring the maintainability of a program provides an understanding of its quality. It is desirable to produce software of high maintainability [16, 2, 1, 17].

### 2.2 Global View : The overall structure

When judging the overall shape of a tangible object we look at its proportions and the way its constituent parts are connected. Certain shapes are pleasing to the eye either because of their symmetry or smoothness of contours. In addition, the shape of an object may be dictated by its purpose (use). Unfortunately it is not possible to judge programs in the same way. However, it is possible to model the shape of a program in terms of the inter-relationships of its modules. The shape of a program known as its *morphology* [18] is given by its call-hierarchy, the way the individual modules are inter-connected. Researchers and practitioners on this aspect have identified four attributes of morphology namely *size, depth, width and connectivity density*. All these measures can be obtained from the call-graph where each module is represented by a node and each call by an arc.

The size of the call-graph can be the number of modules and the number of calls or even the sum of the two. The depth is the longest path from the root node to a leaf node and the width is the maximum number of nodes at any one level. Finally the connectivity density is the ratio of arcs to nodes which increases as more connections (calls) between modules are added.







## 756 Software Quality Management

We present one example program which was analysed to produce measures of inter-modular and intra-modular attributes. Figure 1 shows the Call Hierarchy Matrix which together with the levelling information (Table 1) provide the architecture of the whole program. Furthermore, figure 2 shows the density of module calls also in matrix form. It can be seen that the program has 71 modules.

The total number of calls was found to be equal to 493 giving an average connectivity density of 7. In this particular example about one third of the calls are made from module 6 to module 32. This high concentration of calls rather distorts the overall result. In fact it is a call to a library routine which for the sake of this calculation we can ignore obtaining a more realistic average connectivity density of 4. This means that on average each module is called four times. It is important to examine the reasons why one of the modules is called so many times and ensure it is tested exhaustively. A well balanced graph tends to be an indication of a "balanced" design.

**Table 1 : Levelling of Program Modules**

level	width
0	14
1	21
2	15
3	13
4	5
5	3

max width = 21 (level 1)

max number of calls = 231 on module 32

### 2.3 Coupling of modules

Coupling is a measure of module interdependence and it involves two modules. The global coupling is the overall connectivity of the system represented by a directed graph showing all the connections (calls between modules) including multiple connections between two modules. Low coupling is desirable.

The global coupling can be obtained from the median value of the set of pairwise coupling values  $c(m_1, m_2)$  which are calculated using formula 1 and the coupling classification shown in table 2 [6]. where  $m_1$  and  $m_2$  are the connected modules,  $i$  the worst type of coupling between them and  $n$  the total number of interconnections between  $m_1$  and  $m_2$ .



$$c(m1, m2) = i + [n / (n+1)] \quad [1]$$

**Table 2: Coupling Classification**

Coupling identification	Description	Quality level
5	Content Coupling	bad
4	Common Coupling	
3	Control Coupling	
2	Stamp Coupling	
1	Data Coupling	∨
0	No Coupling	good

In this example the global coupling is equal to 2.95. The local coupling between modules 6 and 32 is  $c(6, 32) = 1 + (231/232) = 1.987$  (approx. = 2).

#### 2.4 Intra-modular view

One fundamental attribute of program modules is that of cohesion. Another word describing module cohesion is module strength which is the extent to which the module components are needed to perform the same task. High cohesion is desirable and a rough idea about its value can be obtained if the purpose of a module can be stated in a single sentence. If this is impossible then the module is likely to perform many tasks which means that it possesses sequential or communicational cohesion. It is believed that programs made up of modules with functional cohesion are highly maintainable.

In the example presented here the cohesion ratio of the number of modules with functional cohesion to the total number of modules is equal to 0.7 which means that this program is expected to be highly maintainable.

#### 2.5 Re-engineering factor

Before embarking onto the difficult and resource hungry activity of re-structuring we can obtain an estimate of each programs re-engineering potential by calculating its re-engineering factor which provides an indication of the amount of re-structuring required for a specific program if a certain target is to be achieved [9]. Following the static analysis we have measures of the code's attributes. The re-engineering factor can be calculated using formula 2 where the  $t_i$ 's are the target values and the  $a_i$ 's are the actual values and





## 758 Software Quality Management

t1 = Target Local Variables      a1 = Actual Local Variables  
 t2 = Target Granularity          a2 = Actual Granularity  
 t3 = Target McCabe                a3 = Actual McCabe  
 t4 = Target Information Flow      a4 = Actual Information Flow

$$r = \frac{\{[a1+a3] \cdot [a2+a4] - [t1+t3] \cdot [t2+t4]\}}{[a1+a3] \cdot [a2+a4]} \quad [2]$$

When the value of r is high (nearly 1) then the program is a strong candidate for re-structuring. Adopting the target values of Granularity  $\leq 50$ , McCabe  $\leq 5$ , Information Flow  $\leq 5$  and Number of identifiers  $< 5$  we calculated the re-engineering factors of 6 programs (Table 3).

It can be seen that Programs B and D are already well structured which in fact will suffer deterioration if they undergo re-structuring according to the target. Program C is a good candidate for re-structuring whilst the improvement expected for Program A is minimal. Similarly Programs E and F can be re-structured to a 27% and 54% level.

**Table 3**

	McCabe	Granularity	Information Flow	Local Variables	Re-engineering Factor
Prog A	8	33	14	6	0.08
Prog B	2	7	7	2	-9.80
Prog C	26	91	1	17	<b>0.81</b>
Prog D	3	9	2	2	-10
Prog E	8	31	21	8	<b>0.27</b>
Prog F	10	70	8	7	<b>0.54</b>
<b>Target Measures</b>	<b>5</b>	<b>50</b>	<b>5</b>	<b>6</b>	<b>0</b>

### 2.6 Additional Static Measures

A reasonably large program written in C was analysed using MLT [12] to produce static measures some of which are listed below [3, 10]. In addition, Table 4 shows the number of local variables for each module, with the granularity of each module shown in brackets.

Total number of statements = 2228      Number of modules = 71  
 Maximum Granularity 1079 (on yyparse)      Average granularity = 31

*The experimental material* for each experiment consisted of two versions of a program (written in C) carrying out the same tasks, using the same input and producing the same output which differs from the required output due to seeded logic errors. Other materials were hard copies of the program, its actual output, its required output and simple user instructions on the interface.

The subjects were required to carry out a series of *tasks* starting with the loading of the allocated version of the program and continuing with corrective maintenance in order to produce the required output.

### **3 The experiments**

#### **3.1 The parameters**

All the experiments in this series have been designed using the 4ET guidelines [4, 15]. The hypothesis under test was : "*Maintainability is a function of the program structuredness*" and more specifically "*Programs with a high degree of structuredness take shorter time to maintain*".

The *experimental subjects* were beginners in programming having studied principles of programming and the syntax of C for 8 weeks. They were familiar with the fundamentals of the UNIX operating system.

#### **3.2 The environment and recording the experimental data**

The maintenance activity was carried out using the interface environment INTER [14] running under UNIX. This is a non-obtrusive Open Windows environment which logs the activity of the subject and the time of their actions. In particular the interface carries out a comparison between the required output and the actual output every time the program is executed and it logs exactly which errors have been rectified.

#### **3.3 The experimental procedures**

Corrective maintenance is carried out on code which is behaving in a different manner to the expected one. The maintainers are provided with a listing of the program, a printout of the required output and a printout of the actual output. The differences in output are caused by carefully seeded errors which need to be located and rectified.



## 760 Software Quality Management

**Table 4 : Local Variables per module.**

1 ... ( st: 1 ) (env.-ids: 0) yywrap	37 ... ( st: 35 ) (env.-ids: 5) mrkdis
2 ... ( st: 477 ) (env.-ids: 2) yylex	38 ... ( st: 25 ) (env.-ids: 7) pick
3 ... ( st: 122 ) (env.-ids: 12) yylook	39 ... ( st: 25 ) (env.-ids: 7) pickall
4 ... ( st: 1 ) (env.-ids: 0) yyinput	40 ... ( st: 4 ) (env.-ids: 0) getitlist
5 ... ( st: 85 ) (env.-ids: 19) main	41 ... ( st: 7 ) (env.-ids: 1) getsisel
6 ... ( st: 1079 ) (env.-ids: 2) yyparse	42 ... ( st: 5 ) (env.-ids: 2) strfromy
7 ... ( st: 18 ) (env.-ids: 5) alldis	43 ... ( st: 4 ) (env.-ids: 0) getstrlist
8 ... ( st: 6 ) (env.-ids: 0) prlev	44 ... ( st: 4 ) (env.-ids: 0) prlst
9 ... ( st: 5 ) (env.-ids: 0) whereinx	45 ... ( st: 21 ) (env.-ids: 7) pickifin
10 ... ( st: 14 ) (env.-ids: 12) tablecalls	46 ... ( st: 3 ) (env.-ids: 1) findsubs
11 ... ( st: 14 ) (env.-ids: 2) prttable	47 ... ( st: 6 ) (env.-ids: 0) pralist
12 ... ( st: 3 ) (env.-ids: 0) yyerror	48 ... ( st: 3 ) (env.-ids: 0) xcalled
13 ... ( st: 7 ) (env.-ids: 0) yyback	49 ... ( st: 1 ) (env.-ids: 0) called
14 ... ( st: 1 ) (env.-ids: 0) yyoutput	50 ... ( st: 6 ) (env.-ids: 0) topfolds
15 ... ( st: 6 ) (env.-ids: 0) yyunput	51 ... ( st: 6 ) (env.-ids: 0) lowfolds
16 ... ( st: 4 ) (env.-ids: 1) consgen	52 ... ( st: 4 ) (env.-ids: 0) prlst2
17 ... ( st: 3 ) (env.-ids: 0) length	53 ... ( st: 4 ) (env.-ids: 0) prlstcn
18 ... ( st: 5 ) (env.-ids: 0) isstrin	54 ... ( st: 5 ) (env.-ids: 1) manifolds
19 ... ( st: 11 ) (env.-ids: 2) snocgen	55 ... ( st: 21 ) (env.-ids: 1) dd_ident_one
20 ... ( st: 3 ) (env.-ids: 0) appgen	56 ... ( st: 4 ) (env.-ids: 0) dd_ident
21 ... ( st: 5 ) (env.-ids: 0) delstr	57 ... ( st: 6 ) (env.-ids: 0) sel_dd
22 ... ( st: 5 ) (env.-ids: 0) delastr	58 ... ( st: 5 ) (env.-ids: 0) lcommon
23 ... ( st: 3 ) (env.-ids: 0) noreplst	59 ... ( st: 5 ) (env.-ids: 1) subs
24 ... ( st: 3 ) (env.-ids: 0) norepalist	60 ... ( st: 5 ) (env.-ids: 1) subs2
25 ... ( st: 3 ) (env.-ids: 0) llast	61 ... ( st: 1 ) (env.-ids: 0) whereim
26 ... ( st: 5 ) (env.-ids: 2) locit	62 ... ( st: 3 ) (env.-ids: 0) listcalls
27 ... ( st: 2 ) (env.-ids: 1) lobj	63 ... ( st: 9 ) (env.-ids: 0) isclose
28 ... ( st: 3 ) (env.-ids: 1) lsel	64 ... ( st: 8 ) (env.-ids: 0) isrecur
29 ... ( st: 3 ) (env.-ids: 1) ltpnts	65 ... ( st: 8 ) (env.-ids: 0) findcycle
30 ... ( st: 3 ) (env.-ids: 1) lrmk	66 ... ( st: 3 ) (env.-ids: 0) nodeastr
31 ... ( st: 9 ) (env.-ids: 3) keep	67 ... ( st: 9 ) (env.-ids: 4) rmcycle
32 ... ( st: 14 ) (env.-ids: 6) MKstr	68 ... ( st: 11 ) (env.-ids: i) nomr
33 ... ( st: 7 ) (env.-ids: 3) MKy	69 ... ( st: 3 ) (env.-ids: 0) recons
34 ... ( st: 3 ) (env.-ids: 1) get_y	70 ... ( st: 12 ) (env.-ids: 1) relocate
35 ... ( st: 3 ) (env.-ids: 1) readit	71 ... ( st: 3 ) (env.-ids: 0) dftype
36 ... ( st: 8 ) (env.-ids: 1) mark	

The whole process is monitored by a program which provides the interface through which the maintainers load, compile and run the program. Each time the program is executed the output is compared to the correct (expected) output without the programmer knowing it. When the programmer is happy or indeed when the time runs out or the programmer gives up (!) they log out.



The interface records all the activities. Two experiments were designed, planned and conducted in two consecutive weeks. The reason for doing this was to ensure familiarity with the environment and as a contingency in case of unforeseen circumstances (power cut, system failure, fire).

## 4. Analysis of the data

### 4.1 Logistic regression

The data were analysed in two different ways. Firstly the number of errors found was treated as a response variable in a logistic regression with binomial errors. The appropriate model was fitted using GLIM4 [ 8 ] to investigate whether the probability of detecting and correcting an error was higher in the case of the structured version of the program than in the case of the unstructured version. The resulting probabilities from the fitted model are shown in Table 5.

**Table 5: Probability of error detection**

	unstructured	structured
1st experiment	0.3182	0.4091
2nd experiment	0.2500	0.7083

The probability for detecting an error for the structured program is higher in both experiments. This confirms the hypothesis. In the case of the first experiment the difference in probability is statistically significant at a 0.05 significant level whilst the same applies to the second experiment at a lower significant level.

### 4.2 Time taken to correct errors

The second analysis is based on the time taken to find and correct an error in the two program versions and the two experiments. Figures 3 and 4 show the total time taken against the number of errors found. It can be seen that the structured versions have, in general, lower values than the unstructured ones. This is more obvious in the second experiment. Table 6 below confirms this observation. It shows the average time taken to find and correct an error in the two program versions and the two experiments (no errors found are excluded). The exclusion of the non hits is necessary but likely to distort the results especially in experiment one where the success rate was lower. Nevertheless the table shows a significant difference in the second experiment.

**Table 6: Time taken to correct errors (in seconds)**

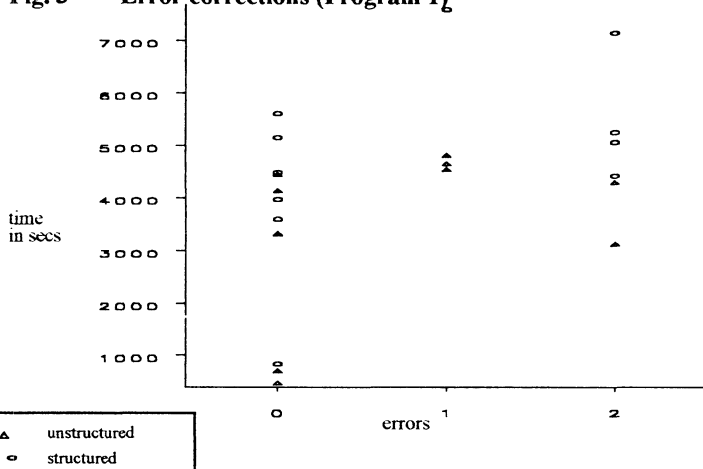
	unstructured	structured
1st experiment	4121	4089
2nd experiment	5708	2262



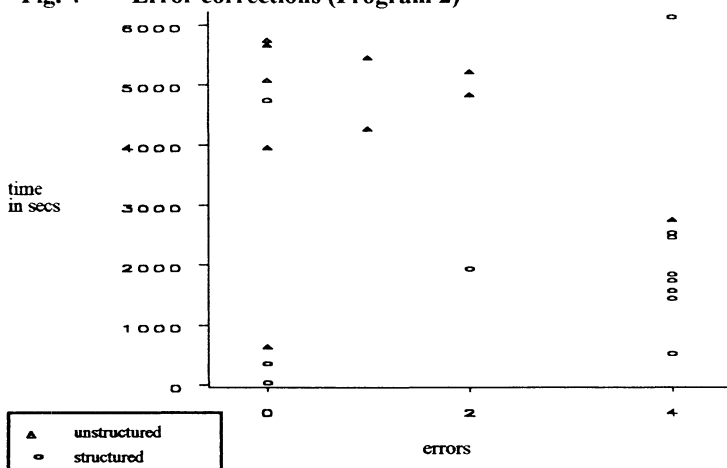
## Figure3 & 4 : Time in seconds versus Number of errors

In the first experiment the programs contained 2 errors therefore the possibilities are 0, 1 or 2 errors. Similarly in experiment 2 the number of errors was equal to 4 giving rise to 5 possibilities namely 0, 1, 2, 3 or 4.

**Fig. 3 Error corrections (Program 1)**



**Fig. 4 Error corrections (Program 2)**





## **5. Postscript**

### **5.1 Conclusions**

The experimental results suggest that the original hypothesis is true. Programs with a high degree of structuredness take shorter time to maintain. Existing code can be analysed and re-structured provided it has a high re-engineering factor. Static analysis tools indicate the problem areas which can then be isolated and improved.

### **5.2 The future**

Further experiments have been planned to carry out adaptive maintenance. If any industrial sponsors become available we aim to run the same experiments in industry where the programs can be large. In addition the subjects can be experienced programmers. Finally we intend to test the same hypotheses using other languages (Pascal and C++).

### **Acknowledgements**

The authors would like to thank all the students who participated in the experiments and all the students and colleagues who contributed their help and suggestions. Special thanks go to Betty Yeboah-Afari for assisting with the logistics of the experiments.



## References

1. Beizer, B. "Software Testing Techniques" Van Nostrand Reinhold, 1990
2. Choi, S.C. and Scacchi W. "Extracting and Restructuring the Design of Large Systems", 11 Software, January 1990
3. Copigneaux, F. "Trends Analysis in Software metrication", EUROMETRICS '91, Mar.1991. Paris, France
4. DESMET - EXPDA Working Documents - NCC, 1993
5. Eastel, C. & Daves G. "Software Engineering Analysis and Design", McGraw-Hill, 1989
6. Fenton N. "Software Metrics - A rigorous approach", Chapman & Hall, 1991
7. Fenton, N. & Hill G. " Systems Construction and Analysis", MacGrw-Hill, 1993
8. Francis, B. , Green, M. , Payne, C. "GLIM4 ", Oxford University Press, 1993
9. Georgiadou E., Karakitsos G. , Sadler C., Stasinopoulos D. "An experimental examination of the role of re-engineering in the management of software quality", SQM'93 Conference Proceedings, Computational Mechanics & at Publications, 1993
10. Gill, G.K and Kemerer, C.F. "Cyclomatic Complexity Density and Software Maintenance Productivity", 11 Trans. on SW Engineering, Vol.17 n0.12, pp 1284-1288
11. Institute of Electrical and Electronics Engineers, Inc. "Glossary of Software Engineering Terminology", ANSI/11, New York, 1983
12. Karakitsos G. & Danicic S. "A i-Language Translator", University of North London, Research Seminar Series, 1990
13. Lewis, T.W. "CASE: Computer-Aided Software Engineering", Van Nostrand Reinhold, 1991
14. Karakitsos G., Georgiadou, E., Jones, R. "INTER - An interface for recording experimental data". University of North London - Research Seminar Series, 1992
15. Mohamed, W.E. & Sadler, C.I. "Design and Analysis of Experimental Design Procedures", University of North London - Internal Report, 1992
16. Pressman R.S. "Software Engineering, A Practitioner's Approach" McGraw-Hill, 1992
17. Troy D.A. "Measuring Quality of Structured Designs", The journal of Software and Systems, 2, 1982
18. Yourdon E. & Constantine L.L. "Structured Design", Prentice Hall, 1979