QUALITY METRICS

IN

SOFTWARE ENGINEERING

QUALITY METRICS IN SOFTWARE ENGINEERING

FAWAZ A. MASOUD

Ph.D 1987

Quality Metrics

In

Software Engineering

Thesis Submitted in Accordance with the Requirements of the University of Liverpool, United Kingdom, for the degree of Doctor in Philosophy.

by
FAWAZ A. MASOUD

June 1987

ABSTRACT

In the first part of this study software metrics are classified into three categories: primitive, abstract and structured. A comparative and analytical study of metrics from these categories was performed to provide software developers, users and management with a correct and consistent evaluation of a representative sample of the software metrics available in the literature. This analysis and comparison was performed in an attempt to: assist the software developers, users and management in selecting suitable quality metric(s) for their specific software quality requirements and to examine various definitions used to calculate these metrics.

In the second part of this study an approach towards attaining software quality is developed. This approach is intended to help all the people concerned with the evaluation of software quality in the earlier stages of software systems development. The approach developed is intended to be uniform, consistent, unambiguous and comprehensive and one which makes the concept of software quality more meaningful and visible. It will help the developers both to understand the concepts of software quality and to apply and control it according to the expectations of users, management, customers etc.. The clear definitions provided for the software quality terms should help to prevent misinterpretation, and the definitions will also serve as a touchstone against which new ideas can be tested.

DEDICATION

This theses is dedicated to my parents, my wife, my children for their support and patience and to the memory of my late brother.

ACKNOWLEDGEMENTS

I would like to thank Professor Michael Anthony Hennell for his guidance and supervision throughout my student life at the University of Liverpool.

I am also thankful to the Yarmouk University, Irbid, Jordan, for financing this research.

I would like also to thank the research group of LDRA and the coworker of the S. C. M. department at the University of Liverpool for giving their time to fill in the circulation of the software quality attributes relationship matrix.

I would like to express my gratitude to my parents for their influence and support.

Finally, I would like to thank my wife San~a who encouraged and gave moral support to me.

CONTENTS

CHAPTER 1 INTRODUCTION

1.1 INTRODUCTION AND PROBLEM DEFINITION 1-1
1.2 OBJECTIVES 1-12
1.3 SCOPE 1-13
1.4 LIMITATION 1-13
1.5 AUTHOR'S CONTRIBUTION 1-14

CHAPTER 2 LITERATURE BACKGROUND

2.1 INTRODUCTION 2-1
2.1.1 Primitive Software Metrics 2-1
2.1.1.1 Halstead's Metric 2-1
2.1.1.2 Function Points Metric 2-6
2.1.2 Abstract Metrics 2-12
2.1.2.1 McCabe's Metric 2-12
2.1.2.2 Knot's Metric 2-17
2.1.2.3 Discriminant Cohesion Metric 2-23
2.1.2.4 Scope Metric 2-30
2.1.2.5 Oviedo's Metric 2-34
2.1.3 Structured Metrics 2-39
2.1.3.1 Haney's Stability Metric 2-39
2.1.3.2 Myer's Metric 2-43
2.1.3.3 S. Henry And D. Kafura's Metrics 2-50
2.1.3.4 Yau And Colofello's Logical Stability Metrics 2-53

CHAPTER 3 EVALUATION OF THE SOFTWARE METRICS

3.1 INTRODUCTION 3-1
3.2 CRITERIA OF GOODNESS 3-8
3.2.1 General Criteria Of Goodness 3-8
3.2.2 Specific Criteria Of Goodness 3-12
3.3 COMPARISON BETWEEN THE PRESENT METRICS 3-13
3.3.1 Primitive Software Metrics 3-14
3.3.1.1 Applicability 3-14
3.3.1.2 Validity 3-16
3.3.1.3 Sensitivity 3-17
3.3.1.4 Procedurising 3-18
3.3.1.5 Language Independency 3-19
3.3.1.6 Simplicity 3-20
3.3.1.7 Positivity 3-21
3.3.1.8 Modularity 3-21
3.3.1.9 Linearization 3-22
3.3.1.10 Unstructuredness 3-24
3.3.1.11 Structuring Transformations 3-27
3.3.2 ABSTRACT METRICS 3-27
3.3.2.1 Applicability 3-28
3.3.2.2 Validity 3-29
3.3.2.3 Sensitivity 3-30
3.3.2.4 Procedurising 3-34
3.3.2.5 Language Independency 3-34

3.3.2.6 Simplicity 3-34
3.3.2.7 Positivity 3-36
3.3.2.8 Modularity 3-36
3.3.2.9 Linearization 3-38
3.3.2.10 Unstructuredness 3-38
3.3.2.11 Structuring Transformation 3-40
3.3.3 STRUCTURED METRICS 3-42
3.3.3.1 Applicability 3-43
3.3.3.2 Validity 3-45
3.3.3.3 Sensitivity 3-45
3.3.3.4 Procedurising 3-46
3.3.3.5 Language Independency 3-47
3.3.3.6 Simplicity 3-47
3.3.3.7 Positivity 3-48
3.3.3.8 Modularity 3-49
3.3.3.9 Linearization 3-49
3.3.3.10 Unstructuredness 3-50
3.3.3.11 Structured Transformation 3-50
3.4 COMMENTS AND CONCLUSIONS 3-50

CHAPTER 4 SOFTWARE QUALITY

4.1 INTRODUCTION 4-1
4.1.1 Definitions And Criticism 4-2
4.2 DESIRABLE APPROACH 4-10
4.2.1 Definition Of The Used Terms 4-11
4.2.1.1 Software Quality 4-12
4.2.1.2 Software Quality Characteristic 4-15
4.2.1.3 Software Quality Property 4-16
4.2.1.4 Software Quality Attribute 4-16
4.2.1.5 Software Qualities 4-17
4.2.1.6 Software Quality Factor 4-18
4.2.1.7 Software Quality Criteria 4-18
4.2.1.8 Software Quality Metric 4-19
4.2.1.9 Software Quality Plan 4-21
4.3 IDENTIFICATION AND DEFINITION OF VARIOUS QUALITY ATTRIBUTES. 4-21
4.4 INTERNAL VIEWS OF SOFTWARE QUALITY 4-24
4.5 THE DETAILS OF THE SOFTWARE QUALITY PLAN: 4-26
4.5.1 Introduction: 4-26
4.5.2 Purposes Of The Quality Plan: 4-28
4.5.3 The Clarification Of The Terms Used In The Quality Plan 4-29
4.5.4 Tools And Support 4-30
4.5.4.1 The Defined Attributes 4-30
4.5.4.2 Synonyms List 4-31
4.5.4.3 Relationship Matrix 4-32
4.5.4.4 Classification List Of The Quality Attributes 4-32
4.5.4.5 Further Classification Of The Quality Attributes 4-39
4.5.5 The Life Cycle Of The Quality Plan 4-43
4.5.5.1 1* Quality Requirements Phase: 4-44
4.5.5.2 2* Quality Factors Phase: 4-47
4.5.5.3 3* Quality Model Phase: 4-49
4.5.5.4 V* Verification And Validation Phase: 4-62
4.5.6 The Diagram Of The Quality Plan Life Cycle . 4-66

4.6 CONCLUSIONS: 4-73

CHAPTER 5 SUMMARY AND CONCLUSION

5.1 INTRODUCTION 5-1
5.2 CONCLUSION 5-4
5.3 SUGGESTIONS AND FUTURE WORK 5-7

APPENDIX A GLOSSARY OF DEFINITIONS OF USED TERMS

A.1 MANAGEMENT A-1
A.2 SOFTWARE QUALITY ASSURANCE A-1
A.3 SOFTWARE QUALITY ATTRIBUTE A-2
A.4 SOFTWARE QUALITY CHARACTERISTIC A-2
A.5 SOFTWARE QUALITY CRITERIA A-2
A.6 SOFTWARE QUALITY FACTOR A-2
A.7 SOFTWARE QUALITY METRIC A-2
A.8 SOFTWARE QUALITY PLAN A-2
A.9 SOFTWARE QUALITY PROPERTY A-2
A.10 SOFTWARE QUALITIES A-3
A.11 SOFTWARE QUALITY A-3
A.12 SOFTWARE SYSTEM A-3
A.13 USER A-3

APPENDIX B DEFINITIONS OF VARIOUS QUALITY ATTRIBUTES

B.1 ACCESSABILITY: B-1
B.2 ACCURACY: B-1
B.3 ADAPTABILITY: B-2
B.4 AUDITABILITY: B-2
B.5 AUGMENTABILITY: B-2
B.6 COERCION: B-2
B.7 COHESION: B-2
B.8 COMMUNICATIVNESS: B-3
B.9 COMPATIBILITY: B-3
B.10 COMPLETENESS: B-4
B.11 COMPLEXITY: B-4
B.12 COMPREHENSIVENESS: B-4
B.13 CONCISENESS: B-4
B.14 CONFORMANCE: B-5
B.15 CONNECTIVITY: B-5
B.16 CONSISTENCY: B-5
B.17 CORRETNESS: B-5
B.18 COUPLING: B-5
B.19 EFFECTIVENESS: B-5
B.20 EFFICIENCY: B-6
B.21 ELASTICITY: B-6
B.22 ERROR-TOLERANCE: B-6
B.23 EXPANDABILITY: B-6
B.24 FEASIBILITY: B-7
B.25 FLEXIBILITY: B-7
B.26 GENERALITY: B-7
B.27 INDEPENDENCE: B-7
B.28 INTEGRITY: B-8

B.29 INTEROPERABILITY: B-8
B.30 INTRAOPERABILITY: B-8
B.31 LEGIBILITY: B-8
B.32 MAINTAINABILITY: B-8
B.33 MEASURABLITY: B-9
B.34 MODIFIABILITY: B-9
B.35 MODULARITY: B-9
B.36 OPERABILITY: B-9
B.37 PORTABILITY: B-9
B.38 PREDICTABILITY: B-10
B.39 READABILITY: B-10
B.40 REDUNDANCY: B-10
B.41 RELIABILITY: B-10
B.42 RESILIENCY: B-11
B.43 REUSABILITY: B-11
B.44 ROBUSTNESS: B-11
B.45 SECURITY: B-12
B.46 SELF-DESCRIPTIVE: B-12
B.47 SENSITIVITY: B-12
B.48 SIMPLICITY: B-12
B.49 STABILITY: B-13
B.50 STRUCTUREDNESS: B-13
B.51 SURVIVABILITY: B-13
B.52 TESTABILITY: B-14
B.53 TRACEABILITY: B-14
B.54 TRANSPORTABILITY: B-14
B.55 TRUSTWORTHY: B-14
B.56 UNAMBIGUITY: B-15
B.57 UNDERSTANDABILITY: B-15
B.58 UNIFORMITY: B-15
B.59 USABILITY: B-15
B.60 VERACITY: B-16

APPENDIX C TABLES

APPENDIX D THE SYNONYMS OF SOFTWARE QUALITY ATTRIBUTES

D.1 INTRODUCTION D-1

APPENDIX E RELATIONSHIP MATRIX

APPENDIX F THE ATTRIBUTES OF THE ANALYSIS PHASE

APPENDIX G THE LIFE CYCLE SUMMARY

APPENDIX H SOFTWARE QUALITIES

APPENDIX I REFERENCES

CHAPTER ONE

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION AND PROBLEM DEFINITION

The research in the area of software quality metrics generally and software quality particularly is still in a state of flux. This is because the area is in its infancy. There is a great demand for ideal software quality metrics and a real need for distinct and precise definitions of software quality and related terms. Software metrics can be considered as the means of measuring software qualities. These measurements are required for quantitative comparison, cost estimation and quality evaluations. In practice, a software metric is the representation of the level of confidence gained through numerical control of software production. This representation can show the extent to which a software system possesses a specified attribute that affects the quality. However, the term software quality, for which various metrics have been developed and applied, is one of those terms in Software Engineering which has not yet been defined clearly, precisely and properly despite numerous attempts. Moreover, other terms related to

software quality such as criterion, factor, characteristic, attribute, etc., are also not defined precisely and clearly by Software Engineers. These terms are used by various authors with different meanings. Authors also use different terms for the same thing and the same term for different things. For example, Hocker et al [PP.2, 90] use the terms attribute and criterion for the same thing.

The term quality characteristic is used by McCall et al [46] as a quality factor. Kitchenham et al [PP. 8, 84] use the terms quality factor, quality attribute and quality characteristic with the same meanings. Further, the term quality is used by Jones [27] to denote the absence of defects from the software without defining the term quality. Yourdan [7] has used the term quality without having any definition. He considered a high quality software system design, as one which consists of modules having a high degree of functional cohesion.

The IEEE standard glossary [13] has defined the term software quality. In their definitions, they have used certain terms or words without giving any definition. For example, the terms "characteristics" and "attributes" have been used without giving a definition in the glossary. This confuses the users, management and developers.

Boehm et al [29] have not given any specific definition of terms software quality and software quality characteristic when they were discussing and developing their hierarchy of software quality characteristics.

A study similar to Boehm's approach was carried out by McCall et al [46] to define software quality aspects. They have defined quality as: "a general term applicable to any trait or characteristic, whether individual or generic; a distinguishing attribute which indicates a degree of excellence or identifies the basic nature of something". In the above definition, the authors have not given any definition for the terms characteristic and attribute which are used in their definition.

A recent study has been carried out by Bowen, Wigle, and Tsai [79], to explain software quality. It is observed that they have also not explicitly defined the term software quality.

Garvin D.A. [92] has synthesised five various distinct views of product quality. He derived these views from philosophy, economics, marketing, and operation management. These views are entirely general and not related to any particular product. However, it needs more care when applying them to a software product.

Kitchenham et al [PP. 394, 70] have discussed the term quality in the light of the Garvin approach which is of a general nature.

Due to the discrepancies and ambiguities in the definitions, it was felt to be unwise and difficult to use them during this study. Therefore, efforts were made to develop clear definitions of these terms to assess the software quality(ies).

Later during the study of software metrics, it was observed that, up to now, efforts in developing software quality metrics have been concentrated on very few quality attributes such as complexity, stability, etc.. On the other hand, for certain important quality attributes such as usability, readability, etc., real metrics are still not available. Moreover, some desired attributes of software quality can only be satisfied at the expense of other attributes. For instance, reliability may be influenced by cost, size complexity, etc.. Further, the quality of a software system is environment dependent, thus, it is inadequate to establish a single figure for software quality. Instead, meaningful attributes which contribute to software quality must be identified.

The evaluation of the available software metrics is mainly needed to select a suitable candidate metric (s) which can be used as a measurement, estimation and forecast of the quality of a software system. To evaluate the available software metrics, a set of criteria of goodness is essential. A perfect set of criteria of goodness which can be applied to all categories of software metrics is still not available.

Moreover, the pressing demands for immediatly usable metrics may cause a quick decision to accept and implement these metrics without detailed examination. Such a quick acceptance may lead to a danger in the area of software engineering, through the selection of invalid metrics. Therefore, the people concerned should realise the potential benefits of having a set of criteria of goodness which a metric has to satisfy and clear definitions of software qualities.

Researchers have made considerable efforts to establish useful metrics. Halstead [2] has developed software metrics which have received much attention in literature of software science. These metrics are based on counting the lexical tokens in the program.

Many attempts have been made to create quantifiable control flow complexity metrics such as McCabe [5] , Woodward [8] etc.. These metrics are based on graph theory. Further many other researchers such as Henry et al [6], Haney [47] etc., have developed a number of software stuctured metrics. These metrics are based on system component connections. There are a large number of software metrics available, but the difficult problem is, how to evaluate and select a suitable and reliable one.

It is difficult for the user, management and developer to decide which metric(s) should be selected to evaluate software quality. The reasons for this are:

i. most of the available software metrics were developed to measure the in-hand code,

ii. these metrics concentrate on very few software quality attributes such as complexity, stability, etc.,

iii. metricians never show the shortcomings of their metrics,

iv. most of these metrics were validated on small programs,

To compare and evaluate these metrics, their validity, applicability, etc., is a tedious job. The following are some possible approaches for helping to solve this problem. Some of these approaches were applied previously; some of these approaches are not possible to apply; some of them are applied in this study.

1. Describe each metric and record general information about each one, such as their use, objectivity, validity and economy in using them, etc..

2. Compare a group of different metrics by showing their purpose, advantages and disadvantages and show their ability to evaluate certain aspects of software quality.

3. Compare each metric against an ideal one. A metric which is commonly used and applicable at different phases of the software system life cycle.

4. Select a set of metrics of similar behaviour and evaluate against certain properties, then apply the suitable one(s) to certain attributes of software quality.

5. Classify software metrics into different categories according to their purpose, behaviour, etc., and select suitable metric(s) to create a tool to measure the software quality.

6. Define software quality and its related terms and identify more attributes which may have impact over the software quality. Apply the suitable metric(s) to measure the possible quantifiable quality attributes.

7. This approach is derived after modification of approaches five and six. It is as follows:
   a. classify software metrics into different categories according to their purpose, behaviour, etc., and select the popular metrics from each category as a sample study and evaluate them against a set of criteria of goodness,
   b. define software quality and its related terms and identify more attributes which may have impact on software quality. Apply the metric(s) which are selected according to the approach given in section a) of this approach to measure the possible quantifiable quality attributes.
   c. define software quality and its related terms from an internal viewpoint. The internal view relates to the

structure or construction of the software.

The first approach in this list was adopted partially by Hocker et al [90] in which they describe 50 software metrics. However, their study was too abstract to be considered for metrics evaluation and selection. Moreover, it was not possible to describe all the available software metrics due to their number. Curtis [PP. 2, 1] stated that " There are more complexity metrics than practicing computer scientists".

The second approach was adopted partially by Baker et al [23]. They have studied and described three software metrics. The purpuse of their study was to select a suitable metric(s) to assess software complexity. The shortcoming of their approach is the lack of cover for so many necessary aspects of these metrics such as applicabilty, validity, sensitivity, etc..

The second approach also was adopted by Kitchenham [87] on a limited basis. She described and discussed Halstead's and McCabe's metrics for the purpose of assessing the ability of these metrics to provide an objective indicator to selected subsystems of the ICL operating system VME/B. This approach was restricted to a specific aspect of software quality and therefore cannot assess all aspects of software quality.

The third approach is infeasible because at present there is no ideal metric which is accepted univesally and applicable to all different phases of the software system life cycle.

The fourth approach was adopted by Sinha et al [38]. Their study was restricted to a certain number of metrics as samples from a single specific category. That category was the one which is based on a graph theory approach. There may be a need to develop new criteria of goodness for each individual category.

The fifth approach was adopted by Ince et al [16]. They have studied, classified and described different software metrics for the purpose of evaluating and selecting suitable metric(s) to be used in a tool to select an optimally designed quality. Their study was restricted to a certain number of metrics. They did not show any detail of the studied metrics. This approach depends on the way the studied metrics are selected.

The sixth approach was adopted partially by Boehm et al [29] , McCall et al [46], Bowen et al [79], and Kitchenham et al [84]. The shortcoming of these approaches is that, the terms quality, quality characteristic, quality attribute, quality criterion, etc., have not been defined clearly and precisely. Moreover, there are no sound metrics available to measure the software quality.

The seventh approach is a modification of the fifth and sixth ones, and it is the one which is adopted in this study. In chapter 2 of this study, software metrics are classified into three categories so that an analytical comparative evaluation of the available metrics can be carried out easily. These categories are:

1. Primitive software science metrics which are based on counting lexical tokens of a program.

2. Abstract software metrics which are based on graph theory.

3. Structured software metrics which are based on software system component connections.

In chapter 2, a sample of metrics from each category is described in detail. An example is given for most of those described metrics so that the metrics can be easily understood. Further, a counter example is also given for most of the metrics described so that the user/management may be made aware of some of their limitations. To facilitate the selection of a suitable metric, a set of criteria of goodness is developed in chapter 3 against which software metrics can be judged and evaluated. Such a set of criteria of goodness is generated after performing a comprehensive study of the popular metrics which are discussed in the software literatue such as [2, 3, 4, 5, 6, 7, 8, 12, 21, 37, 39, 40, 41, 42, 43, 45, 47, 63, 73, 74, 87, 102, etc.,]. Such a set of criteria of

goodness is needed. This is because gathering empirical evidence about the usefulness of a quality metric is hardly worthwile if it fails to satisfy even intuitive criteria of goodness. Also comments and conclusions are developed in chapter 3 to show the advantages and the disadvantages of each of the above mentioned categories and the metrics which are taken as samples in the comparison study. This may help the management, user, developer, etc., to choose a suitable metric(s). Moreover, all the above mentioned problems concerning software quality and its related terms are considered in chapter 4 which consists of the following:

i. the development of clear definitions for the terms software quality, quality characteristic, software quality property, software quality attribute, software quality factor, software quality criteria, software quality metric and software quality plan.

ii. identification and definition of various quality attributes,

iii. identification of the internal views of software quality,

iv. development of a software quality plan.

This approach will enable the developers to understand the concepts of software quality and help them to apply and control it according to the expectations of users, management, customers etc.. The clear definition of

these terms may help to prevent misinterpretation. These definitions will also serve as a touchstone against which new ideas can be tested.

The final chapter contains the conclusions and suggested future work.

1.2 OBJECTIVES

This study is designed to achieve the following objectives:

i. to perform an analytical comparative study of the metrics selected.

ii. to determine, which software qualities, the metric(s) can measure.

iii. to develop a description scheme for the selected metrics.

iv. to establish clear definitions of software quality and related terms.

v. to identify and define various quality attributes which may have impact over the software system life cycle.

vi. to establish internal views of software quality.

vii. to develop a detailed software quality plan.

1.3 SCOPE

The following define the scope of this study:

1. evaluate in an analytically comparative manner the metric(s) which are commonly used in Software Engineering literature.

2. study and examine the selected metrics in each category on the basis of their available referenced material, published literature and direct discussion with some of the authors.

3. attempt to develop uniform, consistent, unambiguous and comprehensive terminologies and their definitions for possible future use.

4. recommend feasible approach which can be adopted in future to improve the quality of the software system.

1.4 LIMITATION

It was not possible or prudent to describe and compare all the available software metrics for the following reasons:

1. most of the metrics which are from the same category have similar behaviour and properties. For instance, McCabe's metric [5] and Myer's interval metric [44]. Further, Van Verth [101] uses extentions of Oviedo's metric [56] together with certain modularity metric to measure student programs.

2. to consider a large number of metrics would make the study difficult to handle and non-transparent.

3. attention should only be given to those metrics which are sufficiently described in the literarture.

It was not possible to relate the internal views of software quality with the external views due to restricted time and finance.

1.5 AUTHOR'S CONTRIBUTION

In this thesis the author attempts to identify the problems which the software users, developers, management, engineers are facing when using existing software metrics to evaluate software quality. For this purpose, a set of criteria of goodness is developed in chapter 3 against which the available software metrics can be judged. Further, clear and distinct definitions of software quality and its related terms are developed in this study. Seven different internal software quality viewpoints are also identified and defined. The number of quality attributes is extended to 60. This is achieved by identifying and defining these software quality attributes. The definitions of the above mentionened 60 attributes are given in appendix [B]. Attempts are made to include all the attributes which describe improvement to or achievement of the quality of a software system to any extent and at any phase of the system development life cycle. It was observed that out

of these 60 attributes some of them are synonyms. A list of these synonyms is generated in this study. These synonyms are expressed in terms of their general meanings as defined in [82]. The list of synonyms is given in appendix [D]. In addittion to providing definitions of the quality attributes and list of synonyms, it was also decided to investigate the extent to which experienced software engineers were familiar with, and consistent in their use of, software quality attributes. This was done by asking coworkers in Liverpool University and LDRA to indicate what they understand to be the nature of the relationship between each of the different software quality attributes. The coworkers who took part in the experiment were each given a relationship matrix, and asked to identify the nature of the relationships among the software quality attributes as shown in appendix [E].

A composite response was constructed from each individual response by including all the relationships which were agreed unanimousely. The composite response is shown in appendix [E].

A detailed quality plan which provides a clear strategy for selecting the desired quality attributes and which provides a frame work to examine the possible effects or behaviour of a certain quality attribute is also developed. Further, new ideas and suggestion for future work are given.

CHAPTER TWO

CHAPTER 2

LITERATURE BACKGROUND

2.1 INTRODUCTION

Metrics for software products can be classified into the following categories:

2.1.1 Primitive Software Metrics

The primitive metrics are based on counts of lexical tokens in a program or program interface features. This type of metric can be applied during the implementation phase of the software development life cycle. Halstead's metric and Albrecht's function points metric are two examples in this category and are discussed in the following sections.

2.1.1.1 Halstead's Metric -

Halstead [2] is the first who presented lexical analysis in his theory of software science. He argued that algorithms have measurable characteristics analogous to the physical laws. In a given program he counted the number of unique or distinct operators(=n1), unique or

distinct operands(=n2), total usage of all of operators(=N1), and total usage of all of operands(=N2). Halstead divided the operators into three groups:

1. fundamental operators such as; =, +, *, /, and, .NE., etc.,
2. keyword operators such as; IF () THEN, DO, GOTO, END OF STATEMENT, which is abbreviated by EOS, etc.,
3. specific operators such as; names of functions, subroutines and entry points.

The operands consist of the all variable names and constants.

Halstead's metrics are developed from the above quantities and are;

a. Vocabulary size: $n = n1+n2$,

b. Observed program length: $N= N1+N2$,

c. Calculated program length:

   $N' = n1*\log_2(n1) + n2*\log_2(n2)$,

d. Program volume: $V = N*\log_2(n1+n2)$,

e. Program mental effort: $E = V/L$, (see below for L),

f. Program time equation: $T = E/S$ where,

   $S = 18$ (Stroud number),

Further, Halstead [2] assumed that the potential volume V' is the most compact representation of the algorithm. Since the most compact representation is considered to be the one corresponds to a single procedure call, Halstead calculated V' as:

V' = (N1'+N2')*log (n1'+n2') where;

n1' is the number of unique operators for a procedure call,

n2' is the number of unique operands for a procedure call,

N1' is the total number of operators for a procedure call,

N2' is the total number of operands for a procedure call,

and program level: L = V'/V.

Since V' is not always easy to determine. Halstead [2] developed a formula which he argued gives an estimate L' for the program level which is calculated as:

L' = (2*n2)/(n1*N2) and

program intelligence content:

I = L'*V.

The example shown in figure 2.1 and table 2.1 make Halstead's metric more clear.

figure 2.1

```
      ISUM = 0
      I = 0
   10 ISUM = ISUM + I
      I = I + 1
      IF (I.LT.10) GO TO 10
      PRINT* , ISUM
```

Halstead's metrics for this sample program are calculated from table 2.1

Table 2.1

i	operator	f1,i	i	operands	f2,i
1	EOS	6	1	ISUM	4
2	=	4	2	I	5
3	IF	1	3	10	3
4	PRINT	1	4	1	1
5	+	2	5	0	2
6	*	1			
7	(	1			
8	)	1			
9	,	1			
10	.LT.	1			
11	GO TO	1			
n1=11		N1=20	n2=5		N2=15

$n = n1 + n2 = 16.$

$N = N1 + N2 = 20 + 15 = 35.$

$N' = n1*\log_2(n1) + n2*\log_2(n2)$

$= 11*\log_2(11) + 5*\log_2(5)$

$= 49.83$

$V = N*\log_2(n1 + n2)$

$= 35*\log_2(16) = 140.48$

etc..

The scope of Halstead's metric can be extendeded to other phases of the system development life cycle. For instance, they can be applied on the design phase provided that the design phase notations are represented as tokens. Using these tokens, metrics can be derived similar to Halstead's metrics. To the author's knowledge Halstead's metrics have not yet been applied to any other phase, except the implementation phase, of the software system development life cycle.

However, it is not difficult to produce examples where Halstead's metrics fail to be good indicators about coding. For instance, in ALGOL-68 it is difficult to differentiate between operands and operators. Figure 2.2 illustrates this dificullty.

Figure 2.2

```
procedure sub = (ref real z,real x,y) void:
                 z := x + y;
                 sub(c,a,b);
```

(The two ALGOL-68 statements are equivalent to the FORTRAN assignment statement c = a + b).

According to Hammer et al [22] the empirical results of Halstead's metrics are inaccurate. Typical research based on the software science metrics is described in [2, 3, 4, 53, 54, 55, 58, etc,]. More comments on Halstead's metrics are presented in chapter 3.

2.1.1.2 Function Points Metric -

Albrecht A.J. [76] has developed a metric called the function points metric. His metric is in the same class as Halstead's metric [2], but instead of counting operands and operators as in the Halstead case, Albrecht counts the number of external functions in the program. Albrecht's function points metric is developed to estimate the complexity of a function which a program performs in terms of input and output data. The general approach is to list , and count the number of external user inputs, inquiries, outputs, master files, and interfaces to be delivered by the development project. Albrecht A.J [76] has pointed out that " these factors are the outward manifestations of any application. They cover all the functions in an application". These counts are weighted by numbers so as to reflect the function value to user/customer. The weights used were determined by Albrecht through debate and trial. The following weights are recommended:

a. number of inputs * 4,

b. number of outputs * 5,

c. number of inquiries * 4,

d. number of master files * 10,

e. number of interfaces * 7.

The weighted sum of the inputs, outputs, inquiries, master files and interfaces is called the function points count. This weighted sum is adjusted for other factors. It can be adjusted within a range of -/+ 25 percent,

depending up on the assessment of a person who is assessing the complexity of the project. The weight given to project influence factors which are given in table 2.2 were recommended by Albrecht et al [75]:

Table 2.2

Weights given to project influence factors	weight
not present, or no influence if present	0
insignificant influence	1
moderate influence	2
average influence	3
significant influence	4
strong influence	5

The project influencing factors are to be considered when the complexity of any system has to be evaluated or assessed. The function points metric is computed as:

FP = FC * W * PIF where,

W is a weighting factor,

PIF is the project influencing factors,

FC is the function points count,

The example which is shown in figure 2.3 may help in the understanding of Albrecht's function points metric. In this example, assume that there is a project to develop a payroll system. This system may have the following external functions:

a. 6 external inputs of moderate complexity level,

b. 6 external outputs of complex level,

c. 2 master files of average complexity level,

d. 3 inquiries of simple level,

e. 2 system interfaces of highly complex level.

Assume that the project influencing factors which are

given in table 2.3 are to be considered in developing the above system:

Table 2.3

PROJECT INFLUENCING FACTORS	WEIGHT
communication facilities	0
on-line processing	3
complex processing logic	5
conversation difficulty	2
system flexibility	4
TOTAL	14

The above weighting numbers are taken from table 2.2. To simplify the example, all the function providing elements of a specific form are assigned the same level of complexity, e.g. each input, is considered to be of moderate complexity.

Figure 2.4

COMPLEXITY WEIGHT SCALE/LEVELS					
FUNCTION ELEMENT	SIMPLE	MODERATE	AVERAGE	COMPLEX	HIGH COMPLEX
input	2	3	4	5	6
output	3	4	5	6	7
master file	5	7	10	13	15
inquiry	2	3	4	5	6
interface	4	5	7	9	10

COUNTS	FUNCTION	WEIGHT	FUNCTION POINT
6	external inputs(moderate complexity)	* 3 =	18
6	external outputs(complex)	* 6 =	36
2	master file (average complexity)	* 10 =	20
3	inquiry(simple)	* 2 =	6
2	system interface(highly complex)	* 10 =	20
Total unadjusted function points count = 100			

Metric computation from figure 2.4 is given below:

FP = FC * W * PIF where,

W is a weighting factor,

PIF is the project influencing factors = 14,

FC is the function points count,

W = .01 + constant,

FP = 100 * [.8 + (14 * .01)] = 94,

(the constants .8 and .01 are developed according to the trial and debate).

Some of the objectives of Albrecht's function points metric are the following:

1. to define a productivity measure for applications development and maintenance functions that avoids the problems inherent in productivity measures based on lines of code [72],

2. to help the management to focus their attention on different levels of applications development productivity being achieved within a data processing facility.

The scope of Albrecht's function points metric is as follows:

i. it can be applied during the early phases of the software system development life cycle. This is because the function points counts can be obtained relatively easily in earlier stages of the system development through the discussion with the user/customer. It can be applied also after the implementation phase either to help in evaluating the cost of maintenance or to give an indication to the user/management about the complexity of the software system.

ii. to help the managers to analyse applications development and maintenance work,

iii. to highlight productivity improvement opportunities,

iv. to measure the equivalent functions of end user applications regardless of language, technology, technique or development environment.

The following are some important properties of Albrecht's function points metrics:

1. it is technology independent. The metric gives the same value for equivalent application functions which are based on external design documents regardless of programming languages or methodologies,

2. it is not affected by the programming style,

3. it is user oriented,

4. it is easy to implement.

However, Albrecht's function points metric needs more research to be performed, in order to improve it against the following:

i. the wastage of time and work due to indecisive user/management,

ii. effect of inexperienced programmer, analysts, etc., there should be some adjustment factor to normalise such effects,

iii. there should be some sound criteria for assessing the weights of the function points metric counts instead of depending on the individual assessment and trial and debate strategy.

2.1.2 Abstract Metrics

The abstract metrics are based on graph theory. In this category for example the researchers measure the properties of a program control flow diagram. The researchers have developed metrics which are derived from a flow graph representation of a program and use these to show the difficulty of carrying out tasks such as coding, debugging, testing and modifying software. This type of metric can be applied to the implementation phase and to the design phase of the software development life cycle. The following are some of the important metrics taken as a sample from this category:

2.1.2.1 McCabe's Metric -

Thomas McCabe [5] has developed a complexity metric which is based on the control flow graph representation of a program. The control flow graph is defined as a directed graph in which each basic block of the program is represented by a node, and the possible flow of control between these blocks is represented by an edge. McCabe's metric is denoted by V(G), and is computed as:

$V(G) = E-V+2P$ where;

E is the number of edges in the flow graph representation of the program,

V is the number of nodes in the flow graph representation of the program,

P is the number of connected components in the flow graph.

McCabe has recommended that: the upper bound of complexity, V(G) in any particular module should have a maximum value of "10". Programmers should consider the upper limit of complexity while developing software modules. If the complexity of a module is greater than "10", then the module should be decomposed or recoded. This enables a software engineer to control the size of a program by setting an upper limit to the measure instead of using just physical size. If a program is decomposed into m connected components, then the value of McCabe's metric for that program will be the sum of the cyclomatic complexities of the components calculated as:

$$V(G) = \sum_{i=1}^{m} V(Gi) \text{ where,}$$

V(Gi) is the complexity of the individual modules

McCabe showed that the cyclomatic complexity of any structured program is equal to the number of predicates in the program plus one. Further, McCabe represented the control flow graph of a program as a directed graph. This directed graph can be reduced by interactively replacing each subgraph corresponding to a structured

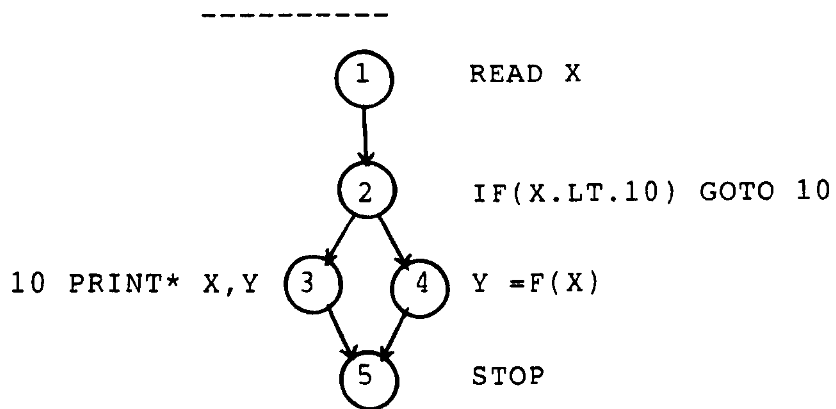
program feature with a single entry/exit node until no further replacement is possible. Applying McCabe's metric to the reduced graph G' gives a measure which is termed the essential cyclomatic complexity. McCabe's essential cyclomatic complexity number, EV(G')=1 for a structured program. The examples shown in figures 2.5 and 2.6 make McCabe's metric more clear.

Figure 2.5

```
      READ X
      IF (X.LT.10) GO TO 10
      Y = F(X)
      GO TO 20
   10 PRINT* X ,Y
   20 STOP
```

The flow graph representation of this code is given in figure 2.6.

Figure 2.6

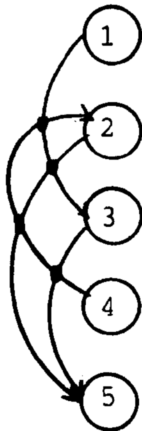
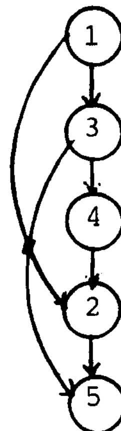


V(G) = e-n+2p

= 5-5+2

=2.

However, McCabe's metric will always give a certain value for any type of flow graph, i.e. planar (a graph is planar if it can be drawn in a plane such that no two edges intersect except at vertices.), non-planar (a graph is non-planar if it cannot be thus drawn in a plane.), connected (a graph is connected if there is at least one path between every pair of vertices.), non-connected (a graph is non-connected if no path exists between some pair of its nodes.), etc.. This value can not reflect the "context" of the nodes in a certain flow graph, and hence, misleading results may be obtained. For example, a program can become more structured when its statements are rearranged in a certain way and hence both it and its flow graph become less complex. This problem can be overcome by applying other metrics to the flowgraph. The knot metric K is a candidate in such situations. Figure 2.7 (a) and (b) illustrates this.

Figure 2.7 (a)	2.7 (b)
	
V(G) = 3	V(G) = 3
K = 4	K = 1

More comments about McCabe's metric are presented in chapter 3.

2.1.2.2 Knot's Metric -

The Knot's metric is a measure of structuredness which has been proposed by Hedley et al [8]. Their Knots's metric is based on control flow edges drawn on the actual sequential source program. The Knots measure is denoted by K and it is calculated as the number of unavoidable crossing points of the control flow edges. This is defined mathematically by Woodward et al [8] as: if a jump from line A to line B is represented by the ordered pairs of integers (A,B) then the jump (X,Y) causes a Knot to occur, if any of the following two conditions is satisfied:

1. MIN{A,B} < MIN{X,Y}, < MAX{A,B}

   AND MAX{X,Y} > MAX{A,B}

   OR

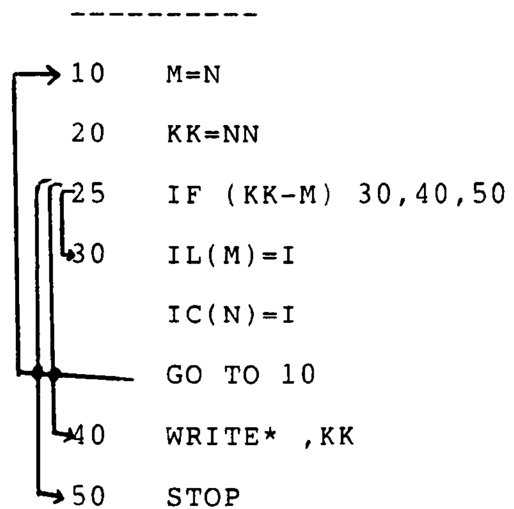
2. MIN{A,B} < MAX{X,Y}, < MAX{A,B},

   AND MIN{X,Y} < MIN{A,B}.

Woodward et al [8] used the idea of control graph reduction to define what they called the "essential knots" of a program. A control flow graph as defined earlier is a directed graph, G, in which the nodes represent the basic blocks and the edges represent control flow between the blocks. A control flow graph of a computer program can be represented as a directed graph. Such a directed graph can be reduced by replacing the subgraphs corresponding to admitted primitives of structured programming by a single node. This technique can be used to discover the lack of structure in a program by continuing graph reduction until no further reduction is possible, and then applying the knot metric to the resulting graph G' to give a measure of program unstructuredness. Woodward et al [8] stated that " a structured program will be reducible to a single node with zero knots. This leads to a definition of the remaining knots as the essential knots of the program". However, this approach is analogous to McCabe's essential cyclomatic complexity number, EV(G')=1 for a structured program. A program given in figure 2.8 is an easy example of a FORTRAN program which makes the knot metric more clear. This program contains two Knots, which is

which is the measure of complexity of the whole program.

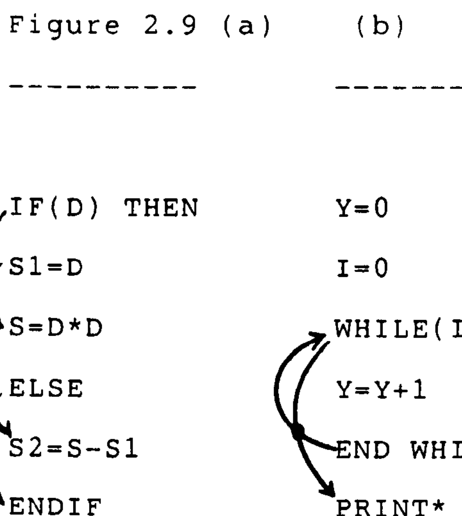
Figure 2.8



If a program has m modules, then the complexity can be computed as the sum of the Knot counts of the modules calculated individually.

There are some possible situations where the Knot's metric fails to distinguish between certain code constructs. For example, the Knot metric gives the same values for the FORTRAN inclusive IF...ELSE constructs and iteration constructs. Such a situation is shown in figure 2.9 (a) and (b). The value of the Knot's metric is the same for the two given programs.

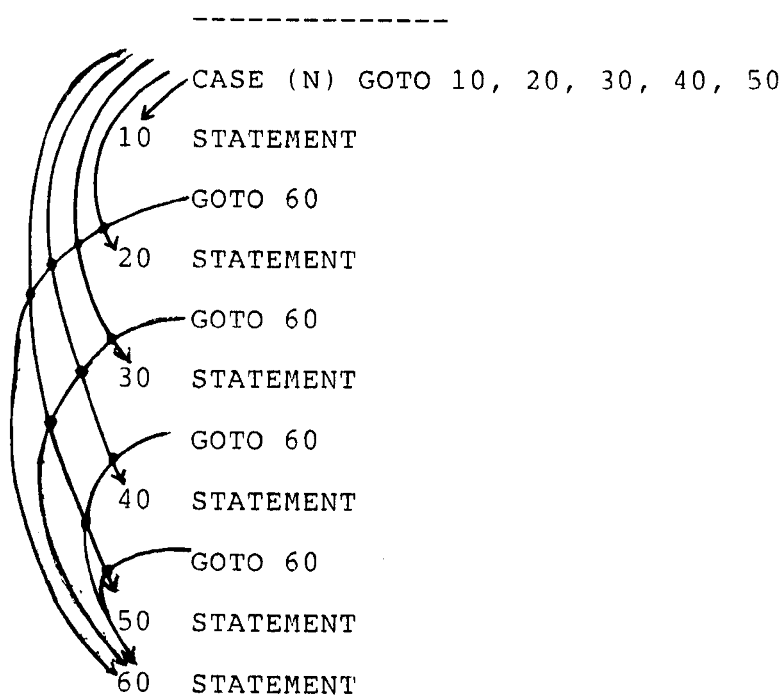
Figure 2.9 (a)   (b)



Note that the number of Knots = one for the above given two programs.

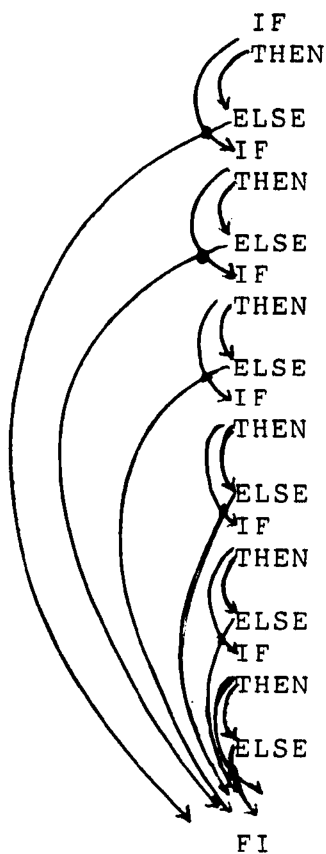
Further, the purpose of the CASE STATEMENT in programming languages is to simplify the control flow in situations that require more than 2-way selection. Therefore, using an n-way CASE STATEMENT should not be penalised in the same way as using the equivalent nested IF...THEN...ELSE structures. The value of the Knot's metric will increase, in most situations, for the CASE construct. For example, the code in figure 2.10 (a) will generate 10 Knots. The equivalent nested IF..THEN..ELSE code will generate 6 Knots which is shown in figure 2.10 (b). Thus, Knot's metric has distinguished between the two constructs but penalised the wrong one i.e CASE STATEMENT.

Figure 2.10 (a)



The value of the Knot's metric K = 10

Figure 2.10 (b)



The value of the Knot's metric K = 6.

The Knot's metric is language dependent in detail. This is because the structure of some languages is similar to a tree structure where no intersections are involved such as LISP. It can be concluded that the Knot's metric is also language independent in principle. This is because, it is based on control flow edges drawn on the actual sequential source program.

2.1.2.3 Discriminant Cohesion Metric -

Emerson [21] has developed a metric to measure module cohesion which works at the level of statements in a program. It is based on the graph theory approach which is used to quantify the closeness of interactions between control flow paths and reference variables. He considered those variables which represent a value that can change overall in time, i.e., he excluded literal and symbolic constants. Essentially what is done is to construct a flow graph for any given module m in two stages:

1. draw a directed graph G which is a standard flow graph. Each node in G represents one executable statement in module m and the edges represent the execution order of a program statements,

2. add a terminal node to the graph and make it accessible by an edge from every node in the graph which corresponds to a STOP or RETURN statement,

3. draw a reduced flow graph F by deleting from the first one any node corresponding to an executable statement of the module m which does not contain a reference to a variable. The incoming edge of a deleted node in the reduced flow graph F will terminate at the successor of that node.

The Emerson's metric is defined as an average cohesion of the reference sets of that module, and it is computed as:

$$K(F) = \frac{1}{n} \left[\sum_{i=1}^{n} \frac{|R_i| \dim R_i}{|VF-\{T\}| \dim F} \right]$$ whereas;

K(F) denotes the cohesion metric,

n = the number of the variables in the module,

F = the flow graph of the given module,

R_i = is the reference set for variable i in a flow graph F, i.e., R_i is the set of nodes of F corresponding to executable statement, which refer to the ith variable,

VF = the total number of nodes of the flow graph F,

T = is an added terminal node of the flow graph F,

dim F is the maximum number of independent paths through F that cover all vertices in F,

dim R_i is the maximum number of independent paths through F that cover all vertices in R_i.

Emerson's metric may become more clear by an example which is shown in figure 2.11 (a), (b) and (c).

Figure 2.11 (a)

```
      DIMENSION STR(26)
      INDEX=1
10    IF(STR(INDEX).EQ.EOF) GO TO 20
      INDEX=INDEX +1
      IF(STR(INDEX).NE.C) GO TO 10
      GO TO 30
20    INDEX=0
30    WRITE(5,50) C
50    FORMAT(9X,A)
      STOP
```

The standard flow graph of the above program is given in figure 2.11 (b) and the reduced flow graph is given in figure 2.11 (c).

Figure 2.11 (b)

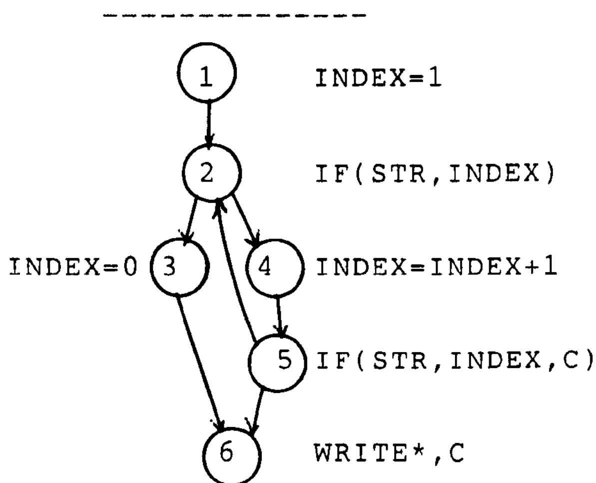
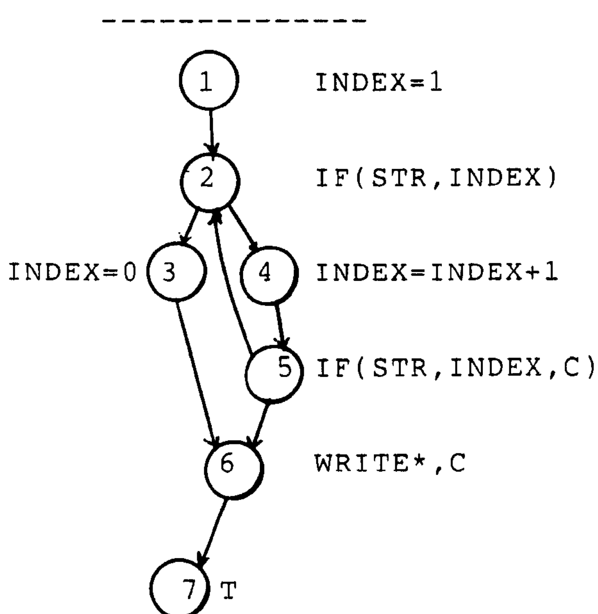


Figure 2.11(b) is a standard flow graph of the code given in figure 2.11 (a) where the variables are displayed opposite to the nodes of the flow graph.

Figure 2.11 (b)



In the given reduced flow graph in figure 2.11 (c),(the nodes in which the variables were not referenced are discarded) the referenced variables are displayed opposite to the nodes of the flow graph. The reference

sets will be constructed as following:

R_1 contains the variable INDEX which appears in nodes 1, 2,3 ,4 , and 5.

R_2 contains the variable STR which appears in nodes 2 and 5.

R_3 contains the variable C which appears in nodes 5 and 6.

According to Emerson the cohesion of the above sample program will be calculated as:

$$K(M) = 1/n(\sum_{i=1}^{3} (|R_i| \dim R_i)/(|VF-\{T\}| \dim F).$$

where n=3 in the given example, i = 1, n,

$|R_1| = 5,$

$|R_2| = 2,$

$|R_3| = 2,$

$\dim R_i = 3,$

$\dim F = 3,$

$|VF-\{T\}| = 6,$

$$K(M) = 1/3[(5*3/6*3) + (2*3/6*3) + (2*3/6*3)],$$

$$= .5.$$

Emerson [21] in his discriminant metric classified the module cohesion levels into three sets comprising the module cohesion levels which are given by Yourdon [7]. These module cohesion levels are of a subjective nature which are difficult to determine. Therefore, his metric fails to give a consistent indication about certain software philosophy. For example, the program which is shown in figure 2.12 is considered to be a purely functional module. The value of Emerson discriminant's

metric will be 0.5 whereas according to his metric, this value must be nearly 1.

figure 2.12

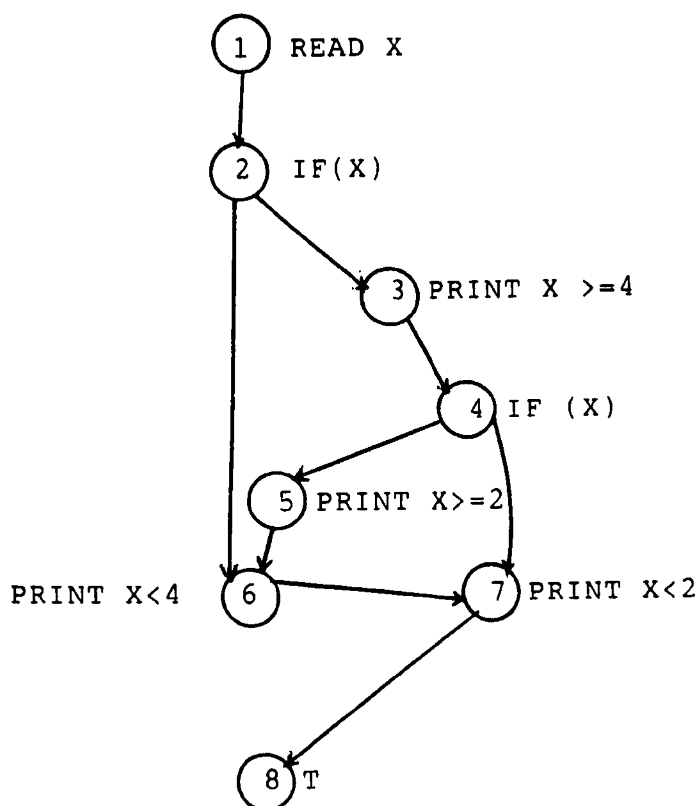
```
    READ N
    IF (N.LE.0) GO TO 10
    IFACT=1
    DO 20 I=1,N
    IFACT=IFACT*1
20  CONTINUE
    PRINT* ,IFACT
10  PRINT* "IFACT=1"
    STOP
    END
```

Further, Emerson's discriminant metric can yield a value of one for an infeasible code. For example, the code in figure 2.13 has a value of one which is not sensible for such infeasible code.

figure 2.13

```
   READ X
   IF (X.LT.4) GO TO 10
   PRINT* "X>=4";X
   IF (X.LT.2) GO TO 20
   PRINT* "X>=2";X
10 PRINT* "X<4";X
20 PRINT* "X<2";X
```

figure 2.14



The path corresponding to the false branch in the first IF statement of the code given in figure 2.13, and the true branch in the second IF statement is infeasible. This is because it would be followed if X>=4 and X<2, which is contradictory. All the modules which were examined by Emerson were small and were functional modules. Therefore his metric was not validated on a large scale problem or on modules with different types of cohesion.

2.1.2.4 Scope Metric -

Harrison W. et al [39] have developed a metric that they call the scope metric. This metric is based on the control flow graph representation of a program. It is constructed as follows:

1. construct a flow graph F of a certain program, with a single entry node and a single exit node,

2. the exit or terminal node in the flow graph is given a weight of complexity equals to zero,

3. each non-selection node is given a weight of complexity equals one,

4. for each selection node in F a subgraph F' is constructed consisting of all the nodes in the different paths which can be taken at the decision node and which pass through a node x, x is = selection node, (including a selection node itself,

in a self loop). Subject to the restriction F' should be the smallest such subgraph. The adjusted complexity of that selection node is calculated to be: the number of nodes in subgraph F' plus one (excluding the selection node itself and x). The overall complexity of the flowgraph F is the sum of the complexities of both the selection and non-selection nodes. The Scope metric is calculated as:

$$r = 1 - \frac{(n-1)}{w} \quad \text{where;}$$

n is the total number of nodes in the flow graph

w is the complexity of the flow graph.

The scope metric will become clearer by studying the example shown in figure 2.15 , table 2.4 and table 2.5

figure 2.15

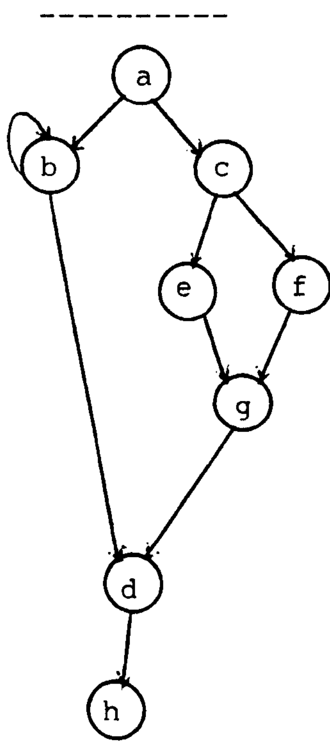


Table 2.4

	decision nodes		
	a	b	c
subgraph	b, c e, f, g, d	b, d	e, f g
complexity	6	2	3
nodes included in complexity	b, c e, f, g	b	e, f

Table 2.5

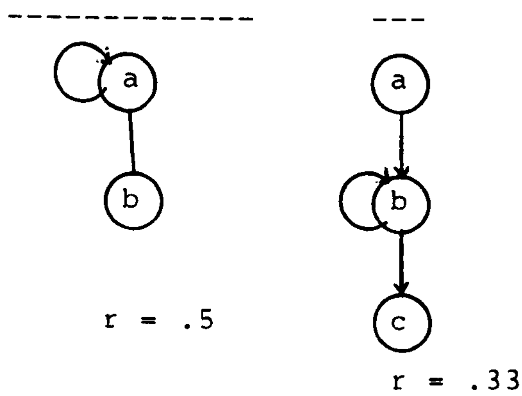
node	complexity
a	6
b	2
c	3
d	1
e	1
f	1
g	1
h	0
TOTAL	15

The scope metric is computed from this example as:

$r = 1 - 7/15 = .53.$

However, the scope metric is not always acceptable. For example, it is not reliable in some cases where the programs can be rearranged to give flow graphs with different values of the scope metric. This is shown in figure 2.16 (a), (b).

figure 2.16 (a) (b)



In figure 2.16 two flow graphs are given with one decision node. Although both graphs depict one decision, the number of nodes involved, and hence the scope metric is not the same for each of them. Further more, if the flow graph has a number of disconnected components, it is not given how the scope metric can be calculated.

2.1.2.5 Oviedo's Metric -

Oviedo [56] has developed a software metric based on control flow and data flow complexity. He computed the total program complexity as:

C =	aCF + bDF where;
CF	is a control flow complexity of the flow graph of a given program,
DF	is a data flow complexity of a given program,
a and b	are appropriate weighting factors.

The control flow complexity CF of a flow graph of a program is computed as the total number of branches connecting program blocks. That is, simply the number of

edges in the flow graph. To compute the data flow complexity of a certain program, the following definitions are required:

1. a variable definition occurs when a variable is assigned a value , either through an assignment statement, input statement, or subroutine call, e.g. x = 1, read y, etc..

2. a variable is referenced when the value of a variable is used, either in assignment statement, e.g.
   x = y + b,
   or in an output statement, e.g. write x, etc..

3. a computer program block is defined by [56] as:
   a. a program statement, or
   b. an end statement, or
   c. a boolean expression with its associated keywords that is IF B1, WHILE B1 DO, UNTIL B1, or
   d. a group of sequentially executed statements.

4. a variable is locally available for a block if that variable is defined within the block.

5. a variable is locally exposed if it is referenced in a block and not defined in that block.

6. a variable defined in a block n can reach a block n if there is a path from n to n such that the variable is not locally available in any block on the path.

The data flow complexity of node n , denoted by df , is

the number of prior definitions of a locally exposed variable in n that can reach n . Data flow complexity for the whole program is computed as:

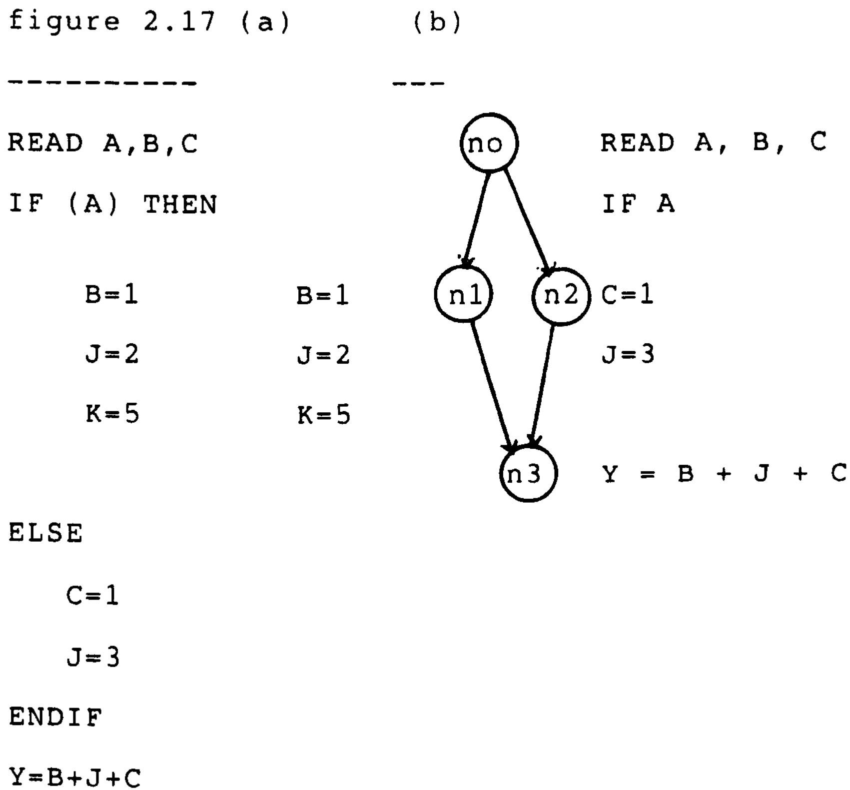
$$DF = \sum_{i=1}^{n} df_i, \text{ where } i = 1, |n|,$$

where n is the number of nodes in a given program.

The complexity of the program will be calculated by:

$$C = aCF + bDF.$$

Oviedo's metric will become clearer by studying the example shown in figure 2.17 (a) and (b).



The data flow complexity of the given program is computed as:

DF = complexity of n0 + n1 + n2 + n3,

n0 = 0, since no prior definition reaches to this block,

n1 = n2 =0, since they have no locally exposed variables,

n3 = 6, since node n3 has three localy exposed variables, i.e B, C,and J. For each locally exposed variable, the following number of prior definitions reach node n3:

B = 2,

C = 2,

J = 2.

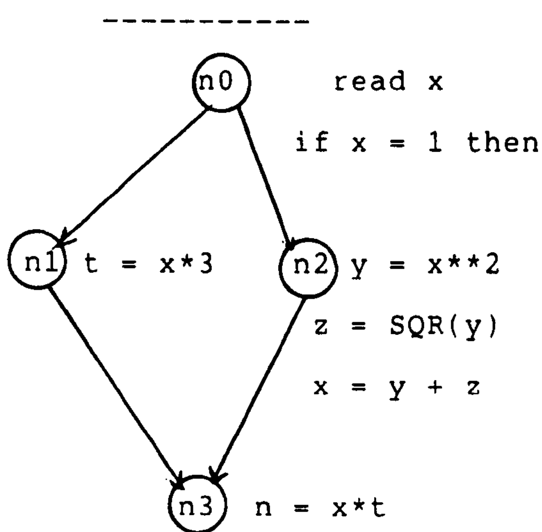
Therefore, the program has a data flow complexity of 6, and since there are the 4 branches, the total program complexity will be:

C = 4 + 6 = 10,

where a and b have been chosen to be unity.

However, the first part of Oviedo's metric fails to consider the context of each edge in the flow graph representation of a program. For example, in the flow graph which is shown in figure 2.18 the context of the edge (n0,n1) may be different from the context of the edge (n0,n2), etc..

figure 2.18



The second part of Oviedo's metric, the data flow complexity, is not supported by any research which ivestigates its effects on the program complexity. Also, the weighting factors a and b are not easy to determine and they depend on a personal assessment. For example,

Oviedo in his initial study considers a = b = one. The metrics discussed above are some important examples of control flow based metrics.

2.1.3 Structured Metrics

The structured metrics deal with measuring the structured properties of software design. Many authors have described how to measure or assess the qualities of a system by measuring properties such as connectivity and cohesion [16]. Yourdon [7] has used cohesion and coupling as metrics to measure the qualities of modules. Researchers have attempted to predict resources (errors, coding time, etc...) expenditure during detailed design, integration testing, and maintenance phases [16]. Such metrics can have significant impact on the software design and development task. This is because structured metrics can be taken early in the life cycle of software system development task [9]. The following are some of the important structure metrics:

2.1.3.1 Haney's Stability Metric -

Haney [47] has developed a metric for determining the stability of large systems which depends on module connections. Haney's metric is based on the assumption that the intermodule connections are the main causes of high cost and delayed delivery dates. These type of problems usually occur in systems where modules are heavily connected (higly coupled) and any change to a

single module causes subsequent changes in most of its connecting modules. The system's resistance to such changes is called system stability. Haney assumed that a system consists of n modules and P_{ij} is the probability that a change in module i induces a change in module j. Further, with each module i, there is an associated number N which is the number of changes that must be made in module i upon the integration with the system. The probability that a change to module i propogates to module j in two steps is given by:

$\sum_{k=1}^{n} (P_{ik})*(P_{kj})$ which represents the sum of probabilities that a change in module i is propogated to module k and then to module j.

In general, the (ij)th element of the probability matrix P raised to the kth power represents the probability that a change in module i will propogate to module j in k steps.

Haney has computed the total number of changes made to all modules during the integration phase as:

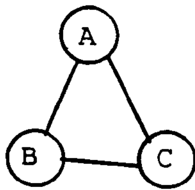
C = V (I + P + P**2 + P**3 + ...) where:

I is the n*n identity metric,

V is a row vector, where V represents the initial number of changes made to module i when it is integrated into the system.

Derivation of Haney's metric becomes clearer when an example which is given in figure 2.19 is considered. In this example three modules A, B and C interact.

Figure 2.19



The metric is calculated by the following steps:

a. for each pair of modules i,j estimate the probability that a change in module i will force a change in module j. These changes constitute the probability matrix which is shown in table 2.6.

Table 2.6 probability matrix (P).

	A	B	C
A	0	.2	.1
B	.2	0	.3
C	.1	.3	0

b. construct a vector V by estimating for each module i the number of changes required at integration time, V_i. This is given in table 2.7.

Table 2.7.

$$V = \begin{matrix} A \\ B \\ C \end{matrix} \begin{bmatrix} 3.1 \\ 4.5 \\ 2.1 \end{bmatrix}$$

c. compute the total number of changes as:

$C = V*(I-P)^{-1}$.

$$C = \begin{bmatrix} 3.1 \\ 4.5 \\ 2.1 \end{bmatrix} * \begin{bmatrix} 1 & -.2 & -.1 \\ -.2 & 1 & -.3 \\ -.1 & -.3 & 1 \end{bmatrix}^{-1}$$

$$C = \begin{bmatrix} 3.1 \\ 4.5 \\ 2.1 \end{bmatrix} * \begin{bmatrix} 1.07311 & .27123 & .18868 \\ .27123 & 1.16745 & .37736 \\ .18868 & .37736 & 1.13208 \end{bmatrix}$$

$$C = \begin{bmatrix} 4.94 \\ 6.88 \\ 4.66 \end{bmatrix}$$

d. sum up the elements of the column vector C to obtain the total number of changes N, which is equal 16.48 in this particular example.

2.1.3.2 Myer's Metric -

Myer, G.J., [88] has developed a structured metric which depends on the degree of interdependence among the components of a program. The major step in calculating this metric is to develop a complete dependence metric (CDM) which describes all dependencies among all modules. Once such a matrix is obtained, the following can be determined easily:

i. The summation of all elements in the matrix divided by the dimension of the matrix (no. of the modules) can give the expected number of modules that must be changed when any single module is changed. The same metric is used by Myers [41] to compute the complexity of the overall program.

ii. The summation of all elements in any row i in the matrix can give the expected number of the modules that must be changed when module i is changed.

To compute the above metric the complete dependence matrix must be derived. To derive the complete dependence matrix, the first order dependence matrix must be derived. The first order dependence matrix is derived using the following steps.

1. Evaluate the coupling among all of the modules in the program. Table 2.8 contains the values of module coupling levels which are suggested by Myers [88]. These values can be used to evaluate the coupling

among all modules in the program.

Table 2.8

Coupling	Value
content	0.95
common	0.70
external	0.60
control	0.50
stamp	0.35
data	0.20

2. Using table 2.8 construct an M*M coupling matrix, C, where; M denotes the number of modules in the program.

3. evaluate the strength (cohesion) of each module in the program. Table 2.9 contains the values of module cohesion levels which are suggested by Myers [88].

Table 2.9

Cohesion	Value
coincidential	0.95
logical	0.40
classical	0.60
procedural	0.40
communicational	0.25
informational	0.20
functional	0.20

4. Using table 2.9 construct a vector S of M elements which corresponds to the cohesion levels of the given modules.

5. Determine the first order dependency matrix (D) by the following formula.

$$D_{ij} = 0.15\ (S_i + S_j) + 0.7\ C_{ij} \quad \text{if } C_{ij} \neq 0,$$
$$D_{ij} = 0 \quad \text{if } C_{ij} = 0,$$
$$D_{ij} = 1 \quad \text{if } i = j.$$

D_{ij} is called the dependency matrix of the modules, and D is the probability that module j will have to change when module i is changed.
S_i and S_j are the strength of the two modules i and j.

C_{ij} is the coupling between modules i and j.

D_{ij} = 0 if there is no coupling.

6. Derive a complete dependence matrix (E) as following

   a. find all the paths between modules i and j. The probablity of a path is the product of all the probability of edges in that path,

   b. if there is only one path, then $E_{ij} = E_{ji} = P(x)$ where; P(x) is the probability of the path.

   c. if there are two paths, then

   $E_{ij} = E_{ji} = P(x) + P(y) - P(x)*P(y)$ where;

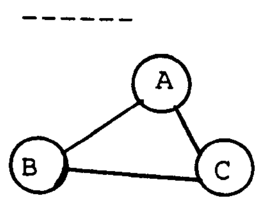
   P(x) and P(y) are the two path probabilities.

   d. if there are three or more paths, find the three paths with the highest probablities. Call these paths x, y and z. Then

$$E_{ij} = E_{ji} = P(x) + P(y) + P(z) - P(x)*P(y) - P(x)*P(z) - P(y)*P(z) + P(x)*P(y)*P(z).$$

The above metric becomes clearer by considering the example shown below in the figure 2.20 which illustrates the relationship between three modules A, B and C.

Figure 2.20



In the above example assume that module A and B, A and C are data coupled and C and B are external coupled. Also assume that A has procedural cohesion, B has functional and C has classical cohesion. The Myers's metric can be determined by the following steps:

1. using table 2.8 evaluate the coupling among all the modules in the program.

2. construct an MxM coupling matrix, where M is the number of modules in the program. This matrix is shown in table 2.10.

Table 2.10 coupling matrix of M*M order.

	A	B	C
A	1.0	0.2	0.2
B	0.2	1.0	0.6
C	0.2	0.6	1.0

3. using table 2.9 evaluate strength of each module in the program. This is given in table 2.11.

Table 2.11

A	0.4
B	0.2
C	0.6

4. construct the first order dependency matrix D by the formula given above.

The matrix D is shown in the table 2.12.

Table 2.12

	A	B	C
A	1.00	0.23	0.29
B	0.23	1.00	0.54
C	0.29	0.5	1.00

Using this the 1st order dependence matrix derive a complete dependence matrix. The complete dependence matrix is given in table 2.13.

Table 2.13

	A	B	C
A	1.00	0.35	0.38
B	0.35	1.00	0.32
C	0.38	0.32	1.00

From this complete dependence matrix the overall design metric can be obtained by summing up all the elements of the complete dependence matrix and dividing the sum by the number of modules (the dimension of the matrix).

Unfortunately, Myers's metric has not been validated [90]. Moreover, the metric is based on unestablished assumptions such as the module cohesion, the module coupling, the symmetric relation between the modules interaction, etc. The computation becomes very difficult to perform for a medium or large scale systems. The first order and complete dependence matrices are assumed to be symmetric. Practically this is not neccessarly true because the effect of a module A on B is generally not the same as the effect of a module B on A.

Moreover, the problem with Haney's metric [47], Myers's metric [88] and similar methods such as those of Schuster [14], and N.L. Soong [15] is that for large systems model validation is difficult. This is because the model inputs have not been automatically obtained. One more weakness of these metrics is the assumption that all modifications to a module have the same ripple effect.

2.1.3.3 S. Henry And D. Kafura's Metrics -

Structured metrics based on information flows have been developed by S. Henry and D. Kafura [6]. Their metrics measure:

1. Procedure complexity:

   The procedure complexity depends on two factors;

   a. the first factor is the internal complexity of the procedure. This is based on counting the number of lines of code in the procedure,

   b. the second factor involves the complexity of the procedure's connections to its environment. This involves the information flow connections of a procedure to its environment. The information flow can be determined by the fan-in and the fan-out. The fan-in and the fan-out represent the total possible number of combinations of an input source to an output destination.

   The whole procedure complexity is computed as:

P(C) = L* (fan-in*fan-out)**2 where;

L is the procedure length expressed in lines of code,

fan-in is the local flows into a given procedure plus the number of data items from which the procedure reads,

fan-out is local flows coming out from a given procedure plus the number of data items to which the procedure writes.

According to Henry et al [6], the local flow of information from module A to module B occurs if one or more of the following conditions hold:

1. if module A calls module B,

2. if module B calls module A and A returns a value to B, which B subsequently utilizes, or

3. if module C calls modules A and B passing an output value from A to B.

The above procedure complexity can be helpful in locating the stress point (the procedure with heaviest data traffic) of the software system.

ii. Module complexity metric:

The module complexity is computed as the sum of the complexities of the procedures within the module. Where a module is defined as: with respect to a data structure D the module consists of those procedures which either directly update D or directly retrieve

information from D [6].

iii. Interface complexity:

Interface complexity depends on two factors; the interfaces which connect the system components, and the number of information paths used to transmit information between the components of the system. The interface complexity between module A and module B is computed as:

$$S(A \rightarrow B) = (E + I)*N \quad \text{where;}$$

S (A-->B)	denotes the strength of connections from module A to module B,
E	is the number of procedures exporting information from module A,
I	is the number of procedures importing information into module B,
N	is the number of information paths between modules A and B.

The above formula is used by Henry et al [6] to determine the coupledness between any two modules.

However, the following comments can be made about Henry et al's metrics:

1. the procedure complexity involves counting lines of code in a procedure and may thus be considered a weak metric because lines of code may be interpreted in

different ways. Therefore, the complexity of a procedure may not be determined correctly by the metric.

2. the fan-in and fan-out are inadequate for computing the complexity of a procedure. This is because a procedure which contains complicated code may be called only once. On the other hand an easy code procedure may be called many times. These cases are not distinguished.

3. The metrics which focus on the interfaces which connect the system components are based on Myers's approach [88] in which he divides the module's coupling into several levels. Henry et al have recognised two of them. That is, content and common coupling levels. These two module coupling levels are classified by Yourdon [7] as poor ones. Moreover, such a classification has a subjective nature, where the quantification may not be possible.

2.1.3.4 Yau And Colofello's Logical Stability Metrics -

Yau and Collofello [45] have developed stability metrics for software maintenance which calculate the stability of modules within a system as well as total system stability. They are based on recording the ripple effects resulting from program modifications. They have divided the logical stability of a module into two aspects. The first aspect concerns the intramodule

change propogation. This involves the number of interface variables affected in one module by the change in a variable i. The other concept concerns the intermodule change propagation. This involves the number of modules affected by changes in the interface variables.

Intramodule change propagation is used to construct the set Z_{ki} of interface variables which are affected by logical ripple effect as a result of modification to variable definition i in module k. This set contains global variables which are referenced by module k, input parameters to modules called by module k and the output parameters of module k. Intermodule change propagation is used to construct the set X_{kj} which consists of the number of modules affected by change in interface variable j of module k. Both intramodule and intermodule changes are used to compute the expected impact of a primitive modification to a module on other modules in the program. A metric is constructed to evaluate the dimensions of the ripple effect which occur as result of modifying a variable definition. This metric is associated with each variable definition in order to determine the impact of modifying the variable definition during maintenance. This metric will compute the logical complexity of modification for each variable definition i in every module k and is denoted by LCM_{ki}. Moreover, a set of the modules involved in the intermodule change propagation as a result of modifying variable definition

i of module k can be constructed as:

$$W_{ki} = \bigcup_{j \in Z_{ki}} X_{ki}.$$

The other metrics developed by Yau and Colofello are given below:

a. The logical complexity of modifications is computed as:

$$LCM_{ki} = \sum_{t \in W_{ki}} C_t \quad \text{where;}$$

C_t is the complexity of a module t,

W_{ki} is number of modules affected by the change in variable i.

b. The potential ripple effect of a modification in module k is calculated as:

$$LRE_k = \sum_{i \in V_k} [\ P(ki) * LCM_{ki}] \quad \text{where;}$$

P(ki) is the probability of change in variable i in module k.

c. The logical stability of a module k is calculated as:

$LS_k = 1 / LRE_k$ for each module k.

d. The potential ripple effect of a program modification is calculated as:

$$LREP = \sum_{k=1}^{n} [\ P(K) * LRE_k\ ] \quad \text{where;}$$

P(k) = 1/n which is adopted as the probability of a change in module k among a total of n modules.

e. The logical stability of a program is calculated as:

LSP = 1 / LREP.

These metrics can be determined by the following procedure which is given by Yau and Colofello [45]:

1. determine the set V_k for each module k, which consists of all variables which are defined in module k,

2. determine the set T_k for each module k, of all interface variables in module k, i.e., global variables, the variables which are an input parameter to a called module and the variables which are an output parameter of module k,

3. determine the set Z_{ki} of interface variables which is a subset of T_k, affected by modifying the variable definition i of module k.

4. for each interface variable j of set T_k, compute a set X_{kj} consisting of those modules which are affected by the modification of the interface variable j.

5. compute the set

   $$W_{ki} = \bigcup_{j \in Z_{ki}} x(kj) \text{ for each variable definition}$$

   i in every module k. The set W_{ki} consisting of the set of modules involved in intermodule change propagation as a consequence of modifying variable definition i of module k,

6. compute

$LCM_{ki} = \sum_{t \in W_{ki}} C_t$ for each variable definition i, in every module k, where C is the McCabe's complexity measure of module t,

7. for each variable definition i in every module k, compute

P(Ki) = 1 / (number of element in V_k) where;

P(Ki) is the probability that a particular variable definition i of module k will be selected for modification,

8. compute

LRE_k = [P(Ki)* LCM_{ki}] and

LS_k = 1 / LRE_k for each module k,

9. compute LREP and LSP as follows:

$$LREP = \sum_{k=1}^{n} [P(k)*LRE_k]$$ where;

P(k) = 1/n, n is the number of modules in the program,

LSP = 1 / LREP.

However, Yau et al., have not yet validated their metrics. This is admitted by Yau et al themselves [45]. Further, the problems with their metric are:

1. they assume that the probability of the modification

is the same for all variables of the module,

2. they assume that the probability of the modification is the same for all modules of a program.

Additionally, it is very difficult to determine their metric manually and it becomes extremly so in the case of large systems.

CHAPTER THREE

CHAPTER 3

EVALUATION OF THE SOFTWARE METRICS

3.1 INTRODUCTION

Recent estimates suggest that up to 90 percent of annual expenditure on systems in a large organisation is devoted to developing and maintenance of software systems [68, 28]. One of the reasons for this is that the software components of these systems were developed without applying the correct concepts of software quality. For example, most of them did not consider the importance of having desired quality attributes in the software system, such as clear documentation, maintainability, etc.. Also the nonavailability of ideal quality metrics to measure the software quality attributes has led to increases in the cost of developing and maintaining software systems. A solution to this problem is to improve software quality. This can be done by introducing clear concepts of software quality and applying software quality metrics during and after the development of the software system.

A survey of the software literature revealed that in the last 10 years many researchers have made considerable efforts to establish such metrics. For example, Halstead [2] has developed software metrics which received much attention in the literature of software sciences. McCabe [5] has developed another software metric which appears to be dominant, at least for measuring the control flow complexity of a program. Many attempts have been made to quantify the control flow complexity, for example, Woodward et al's Knot count [8] and Harrison et al's Scope m etric [39].

Further, many other researchers have developed a number of structured software metrics. These metrics have been created mainly to examine up to which level the components of a software system are connected. For instance, Haney [47] has developed a metric based on module connection which measure the stability of a software system.

However, they have not given any proven evidence in support of claims they have made about their metrics. It is therefore difficult to say how reliable these claims are.

There are very many software metrics available, and it is a difficult problem to evalute and select a suitable and reliable one. Most of these metrics measure the complexity attribute of a control flow graph of a computer program. Baker et al [23] chose three software

metrics for the purpose of comparing and evaluating them. These metrics were Halstead's metrics [2], McCabe's metric [5], and the Knot metric [8]. Baker et al have showed some basic properties of each of them. However, the drawback of their approach was that they did not consider all the necessary aspects of these metrics such as the validity, applicability, etc. Therefore Baker et al's descriptions are not enough to select a suitable metric. Further, they did not mention any thing about the structured metrics which are very important at least for the management of software systems.

A similar study was carried out by Sinha et al [38]. In their study they selected the Scope metric [39], McCabe's metric [5], and the Knot metric [8]. They suggested a list of properties for the purpose of comparing between these metrics. However, they considered only one attribute of software quality: the complexity of a program control flow graph. Moreover, they studied a sample from only one category of software metrics, i.e. the one which depends on graph theory.

Kitchenham [87] has described and discussed Halstead's and McCabe's metrics briefly. The purpose of this study was to assess the ability of these metrics to provide an objective indicator for selected subsystems of the ICL operating system VME/B. Some results of her study are as follows:

1. McCabe's and Halstead's metrics offer little help in the evaluation of the VME/B subsystems,

2. McCabe's and Halstead's metrics are good measures of program complexity which is based on the size of the program.

However, the Kitchenham study does not identify the conditions under which these metrics may or may not be used beneficially.

A further study was carried out by Hocker et al [90] in which 50 software metrics were described and ivestigated. Their description consist of the following points:

1. the name of the metric and a brief explanation of what the metric can measure,

2. the method of constructing the metric,

3. a scale by which a given metric can be judged,

4. a description of the measurement process,

5. identification of the quality factor to which the given quality metric refers to.

Hocker et al [90] have identified goodness criteria for the selection of suitable metric(s). Some of their criteria were enlarged in this study, e.g. validity, the other criteria were either not included, e.g.

reliabilty, or replaced by another criteria of goodness, e.g. economy. Validity was used by Hocker et al [90] to indicate "the extent to which a mesure actually measures that quality characteristic which it should measure or which it claims to measure". The term validity was enlarged in this study and it was interpreted appropriately in order to determine; the size of the system on which the metric(s) can be validated and the purpose/ reason which it was designed to assess. The reasons for such enlargement are the following:

1. to show those metrics which were validated on small size programs and are difficult to be validated on large software systems, e.g. Haney's metric [47], Myers's metric [44], etc..

2. to show those metrics which were considered to measure the complexity of the program and fail to do so, e.g. Halstead's metrics.

The criteria of goodness objectivity and reliability which are given by Hocker et al [90] are not included in this study. This is because of the following reasons:

1. Hocker et al describe the objectivity criteria as "the extent to which the quality values obtained by means of the measure are free from subjective influences of various evaluators" [90]. Since it is impossible for a software engineer to be free from a subjective influence and the contribution can not be

measured so this criterion of goodness was not included in this study. For example, some people consider the length of code as a complexity measure whereas other people do consider it as a weak measure of complexity.

2. Hocker et al describe the reliability criterion of goodness as: "the degree of exactness to which a product property or a combination of properties is measured, regardless of whether the quality characteristic to be measured is actually identified by this". This definition requires a knowledge of both the measured value and the exact value. It would seem that using the value solves the problem. This definition is therefore inadequate. Moreover, reliability is one of the software quality atrributes which has many aspects and may be assessed differently by different users. Therefore, it was felt to be unwise to include it as one of the criteria of goodness.

The "economy" criterion of goodness is used by Hocker et al [90] to indicate "the efforts which are required for obtaining quality values by means of the measure". Since efforts depend on individuals and their ability, amount of automation, understanding and experience, and the criterion of goodness economy has many other aspects such as cost, time, manpower, etc., which make the comparison study very difficult, so it is

replaced by the criterion simplicity which means the ease of computation of a software metric(s). However, the above approach may not be enough to choose the candidate metric of interest. This is because Hocker et al's study [90] is too abstract to be considered for metric selection. They have not considered examples which show the validity or the invalidity of any described metric.

Boehm et al [29] have selected metrics based on the correlation to quality, and the expected gains of using metrics. However, their criteria are specific since they are mainly concern with the FORTRAN programming language.

McCall et al [46] selected metrics using statistical evaluation (rating) of the relationship between quality factors and the quality metrics using regression and cross-validation. There is no single metric which can be considered to give a universally useful rating of software quality and further software quality metrics are not comprehensive [29]. Therefore, the overall result of rating would be more suggestive than conclusive.

In the following section a set of detailed criteria of goodness are given against which each software metric can be evaluated. The set of developed criteria of goodness may not be comprehensive but at least it covers most of the points which have been mentioned. Further, it is applicable to the three different categories of software metrics which are considered in this study. These criteria of goodness are discussed in the next

section.

3.2 CRITERIA OF GOODNESS

The main objective of this section is to generate a list of criteria of goodness for software metrics. These criteria of goodness can be used to evaluate the existing software metrics and to rate one metric against another. Such criteria of goodness can be classified in two categories:

a. general criteria of goodness,

b. specific criteria of goodness.

3.2.1 General Criteria Of Goodness

General criteria of goodness are those criteria which can be applied to any metric for the purpose of evaluation. They are discussed below:

i. Applicability: the term applicability means the suitability of metric(s) to the output of different phases of the system development life cycle, i.e., number of the software life cycle phases in which the metric under study can be applied.

ii. Validity: the term validity is used and interpreted appropriately in order to determine; the size of the system on which the metric(s) can be validated and the purpose/reason which it was designed to assess.

iii. Sensitivity: the term sensitivity is used to show the capability of a software metric for being responsive to the changes in the structure, environment, properties, etc., of the software system.

iv. Procedurising:

A procedure is defined in [107] as: "A body of a program, written out once only, named with an identifier, and available for execution anywhere within the scope of the identifier". An example of procedure is a subroutine of FORTRAN, a procedure of ALGOL, a function of C, etc.. Procedures are used to reduce the amount of coding where several statements are duplicated at different parts of a program. Generally the complexities of these procedures constitute the complexity of the whole program [66]. Therefore, to compute the complexity of a program in a software system it is necessary to compute the complexity of each procedure which is called into the program. A procedure may be called into the main program and then the following assumptions are possible:

1. assume that each procedure is only called once and the control flow returns to the point of call. In this case the complexity of the whole program can be calculated as:

$$C = C' + \sum_{i=1}^{n} (C_i - 1) \quad \text{where;}$$

C' is the complexity of the main program,

C_i is the complexity of each procedure,

n is the number of procedures.

2. assume that a given procedure is called more than once, then the following strategies may adapted:

   2.a) add the value of the complexity of a given procedure every time the procedure is called. In this case the complexity of the whole program can be calculated as:

   $C = C' + \sum_{i=1}^{n} N_i * (C_i - 1)$ where;

   C' is the complexity of the main program,

   N_i is the number of calls of the ith procedure,

   C_i is the complexity of each procedure,

   n is the number of procedures.

   2.b) consider that only one copy of the procedure flow graph is used and the edges are added to the main flow graph for calls and returns. Almost always the overall program flow graph becomes non-planar and the complexity of the whole program may be increased by:

   $C_i + \sum_{i=1}^{n} (2N_i - 3)$.

The second alternative produces an unsatisfactory

solution because the procedurising issue, which is a desirable criterion of goodness, is excessively penalised whenever possibility 2.a or 2.b is used. It can be concluded from the above that the first assumption is the best solution. Therefore, the first assumption is recommended and considered in this study. It may be beneficial to examine the behaviour of these metrics against the criterion of goodness procedurising.

v. Language independency: language independency means the software metric(s) should be computable for any software system, independent of the language in which the system is written. There may be an exception for the systems which are written in some functional languages such as PROLOG, APL,.etc..

vi. Simplicity: simplicity means the ease of computation of a software metric(s). The efficiency of the computation of such metric(s) is excluded. This is because at this stage, the worthiness of these metrics is the subject of debate. Therefore, the comment is made only on the difficulty of software metrics computation. It is better to have a metric which is easy to compute.

vii. Positivity: the term positivity is used here to mean that when a software metric has a numeric value then that value must always be equal or greater than zero. The value of a software metric can be zero for a null

program, otherwise it must be greater than zero.

viii. Modularity:

A module can be defined as: a collection of executable statements, procedure declarations, data structure declarations, and operators declarations. A module may contain zero or many procedure bodies, e.g. the segment of CORAL 66, the package of ADA, the program of COBOL, the module of RTL/2, etc. [106]. Modularity is derived from module and it can be defined as: the extent to which a software system can be decomposed into modules provided that a change in one module has a minimal impact on other module. Modularity can be achieved by isolating frequently occurring sequences of duplicate code [55]. It is worthwhile knowing how much the modularization issue can affect the value of the software metrics.

3.2.2 Specific Criteria Of Goodness

The specific criteria of goodness are those which can only apply to a particular type of metric such as McCabe's metric [5], Woodward et al's Knot metric [8], etc.. Example of some of these criteria of goodness are given below:

1. Linearization : this is a process for expressing a two-dimensional flow graph algorithm as a one-dimensional set of program statements. The

purpose of linearisation is to enable a particular machine to execute that algorithm, i.e. a transformation from a graph to a coding. This criterion of goodness is revelant to the knot metric.

2. Unstructuredness: the term is used to mean that the program is not restricted to specific style or programming techniques such as structured programming. Some available software metrics can determine and recognise the unstructuredness of a software system.

3. Structuring transformation: the structuring transformation can be defined as a design strategy in which the structure of a system can be expressed in a better and simpler way. For example, the more a software system deviates towards a pure tree structure the better the design under some criteria. One of these structuring transformation techniques is node-splitting. It will be useful to know to what extent such techniques can help to decrease or increase the value of a certain software metric

3.3 COMPARISON BETWEEN THE PRESENT METRICS

The purpose of this section is to give a comparison between software metrics. This comparison is based on the criteria of goodness, which are defined in previous section. These criteria of goodness were generated after performing a comprehensive study of a selection of the

most popular metrics.

These metrics are available in the literature, for example in [2,5,6,7,8,21,39,41,47, 48, 49, 51, 52, 59, 86, etc.]. It was not possible to cover all these metrics. Hence in this study a sample of software metrics from each category was taken, so as to carry out the comparison study.

3.3.1 Primitive Software Metrics

Since Halstead metrics have received considerable attention in the software literature, it was decided, in this study, to apply the above criteria of goodness, to Halstead metrics.

3.3.1.1 Applicability -

Applicability as defined earlier means, the number of phases of the system development life cycle in which the metric(s) can be applied. There are several life cycles models [64], [65] , etc.. The life cycle model which is referred to here, is the one which was developed by M. U. Shaikh [50], and the summary of this life cycle is given in appendix [G]. The reason for selecting this life cycle model is its global nature and applicability to a wide class of problems.

Halstead's metrics which are based on collecting the lexical tokens in a program, are applicable to the implementation phase of the life cycle. This is accepted by Halstead [2]. According to Baker, et al [23] the software science metrics are generally developed to measure overall program complexity. It would be possible to apply Halstead's metrics to the output of each phase of the life cycle provided that at each phase notations can be represented as lexical tokens. However, the main difficulty which may arise is the decision about which entity should be treated as an operand and which as an operator. This difficulty causes problem in the well known language ALGOL 68. For example, Halstead [2] considered that, the operands are the variables and constants in a program. Woodward [81] has shown by giving an example from the NAG library (this is shown in figure 3.1) that the procedure call dot(x,y) counts as an operator occurrence but the REAL variable local to the procedure, also called dot, is in this case counted as an operand.

Figure 3.1

```
PROC dot=(REF[]REAL a,b)REAL:
BEGIN
   REAL dot:=0.0;
   .
   .
   .
end;
.
.
s:=dot(x,y);
.
.
```

The above example shows only a simple case of confusion in counting operands and operators. There are other situations which also cause confusion. For example, a function reference may serve as an operand and operator at the same time.

3.3.1.2 Validity -

It is observed that most of the software metrics were validated on small programs due to limitations placed on researchers, such as time available and size of the program allowed [43].

However, it is possible that most of Halstead's metrics can be validated on large systems, because these metrics are mainly developed from counting the number of operators and operands, in the program. This counting is easy to perform. For example, a compiler does this in its symbol table [4]. But the question is whether these metrics are measuring what they were designed to assess?. The answer is unfortunately no. This is because overall program length cannot represent the real difficulty that the programmers may face. The program length cannot represent the real complexity which the programs possess. For example, a small program which contains, predicate nesting and sequence statements is more difficult and more complex than a very big program which contains only a large sequence of assignment statements. The program level (the difficulty of understanding a program) is computed in Halstead [53] as:

$$L = \left(\frac{2}{n1}\right)*\left(\frac{n2}{N2}\right)$$

This definition has been changed later to L = (V'/V), see Halstead [2], and the original definition renamed to L'=(2/n1)(n2/N2). Halstead considered L and L' to be identical. This has resulted in a doubtful situation where all software science developers compute L but derive their algebraic arguments in terms of L' which nobody ever measures. Additionally, see table 2.1 in appendix [C], which shows that the values of L and L' are not identical and the calculation of their values were approximated. The metric of the time equation is computed as:

T=E/S, where

T= the time is needed to generate a program

E= the software science metrics of effort

S= 18 (stroud number)

In the above equation the definition of T is largely speculative and it needs empirical confirmation [22]. It seems to be that Halstead's metrics are not achieving the actual aims, which they were designed to assess.

3.3.1.3 Sensitivity -

It is obvious that Halstead's metrics are sensitive to the changes in the structure of the system which cause an increase in the number of operands and operators in the program(s). However, the changes which do not

increase the number of operands and operators but do alter their frequency of occurence will not effect the value of all of Halstead's metrics. Only those that depend on the number of ocurences of the operators/operands. Further, Halstead's metrics are sensitive to the errors which cause an increase in the size of the code in the software system. Otherwise, the errors which do not increase the number of operands, operators or their frequency of occurence will not effect the value of Halstead's metrics. Thus modifications and restructuring may not be measured.

3.3.1.4 Procedurising -

Generally the calling of the procedure, functional statement, subroutine, subprogram, etc.. will increase the number of tokens in the program. This is because the procedure call has been treated as an operator in the field of software science. Hence, the value of E (the effort metrics) and the value of V (the volume metrics) will increase. So the performance of Halstead's metrics are not in favour of the procedurising issue. This is an unsatisfactory result, because procedurising is generally a desirable practice, and it is penalised by these metrics.

3.3.1.5 Language Independency -

Halstead's metrics n (program vocabulary), N (program length) and V (the program volume) are obviously language independent according to the definition given. According to the more usual definition of language independency they are language depedent. This is because these metrics are directly depend on counting the number of operators, operands, and their occurrence in the program, and the definition of operators and operands varies from language to another. For example, the metric V is expected to increase as it is translated from a high level language to low level language. This result is shown by Fitzsimmons et al [4], where a FORTRAN implementation of an interchange sort algorithm had a volume of 204.4 bits, whereas the same algorithm had a volume of 328.5 bits, when it was implemented in Assembly Language. It is believed that the metric L (program level) will decrease with both an increasing number of operators and an increasing number of recurrent uses of operands [2], therefore, L will be language dependent. The metric E (program effort) is calculated as E=V/L. Since the volume is inversely proportional to the level of abstraction "L". Then as the volume of a program increases and the level of a program decreases, the effort should increase. Therefore E is language dependent. The metric I (intelligence content) is claimed to be a constant for all implementations of an algorithm [2], i.e. I is independent of the language in

which a certain algorithm is coded. Halstead proved his claim by an experiment of implementing Euclids algorithm for finding the greatest common divisor using eight languages. The values of I were not exactly equivalent. Further, Christensen et al [93] performed the same experiment using slightly different languages. The values of I were not equivalent. The two experiments are presented in tables 2.2 and 2.3 in appendix [C]. However, the above experiments were made for small programs, and even then the values of I were not equivalent.

Further, Halstead [2] calculated the intelligence content(I) as:

$I = L' * V$ where;

V is the volume of the program,

L' is the program level,

V obviously varies with both the algorithm coded and the language used. Hence, it can be concluded from the above and the tables 2.2 and 2.3 which are given in the appendix [C], that most likely I is language dependent.

3.3.1.6 Simplicity -

The Halstead's metrics E, V and N are the most frequently referred to and discussed in software literature such as [23, 55, 58, etc...], and these are easy to compute. For example the operands of a program can be identified and tabulated easily provided knowing what is constitutes the operands and operators. This

identification and tabulation can be done by a compiler's symbol table [4]. Therefore, it is clear that most of Halstead metrics are easy to calculate provided that decisions about what constitutes an operator and what constitutes an operand are clear and are made in advance.

3.3.1.7 Positivity -

The values of Halstead's metrics always have a non-negative value. This is because he counts operators and operands in the program to calculate the values of these metrics.

3.3.1.8 Modularity -

Modularity may cause a reduction in a program's observed length, and therefore Halstead's metrics involving this will be affected by the modularization issue. For example in a program which calculates the net pay of hourly workers and salaried workers, there will be a part somewhere in the program to calculate the normal deductions for the hourly paid workers. It may also possible to have a similar part in another place in the program to calculate the normal deduction for the salaried workers. According to modularity principles, the two similar parts must be combined in one subunit which must be isolated and then interfaced with the other subunits of the program. This modularity principle will cause a reduction in the observed program length, hence the value of Halstead's metrics will be decreased.

Therefore, Halstead's metrics do reflect the reduction in the code which occurs as a result of applying a modularity technique. This is because the number of tokens will be decreased in the case of modular programming. However, there is an exception in case of housekeeping statements such as declaration which may increase the number of tokens in a program.

3.3.1.9 Linearization -

To execute, any algorithm represented by a two-dimensional flow graph, on a particular machine, that must be linearized. There may be many possible linearizations of a particular flow graph [57]. The code of one linearization may have a shorter length than the code of another linearization. Baker et al [23] claimed that Halstead's metrics Vand E reflect such difference in linearizations. He showed this by an example which is given here in figure 3.2 (a), (b) and (c).

Figure 3.2 (a)

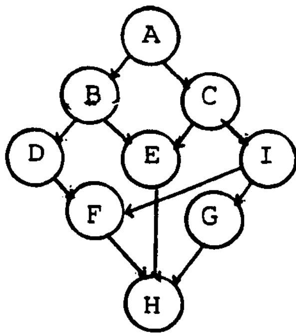


Figure 3.2 (b)

```
         IF (A) THEN
           IF (B) THEN D
            ELSE
E:           IF (E) THEN NULL
             ELSE GOTO H
             END IF
            END IF;
F:          F
         ELSE
           IF (C) THEN GOTO E
           ELSE
             IF (I) THEN GOTO F
             ELSE G
             END IF
           END IF
         END IF
H:       H
```

Figure 3.2(c)

```
      IF (A) THEN
        IF (B) THEN D; GOTO F
        ELSE GOTO E
        END IF
          ELSE
            IF (C) THEN
E:            IF (E) THEN NULL
              ELSE GOTO H
              END IF
            ELSE
              IF (I) THEN NULL
              ELSE G; GOTO H
              END IF
            END IF;
F:          F
      END IF
H:    H
```

The code given in figure 3.2(b) and (c) are equivalent linearizations of the flow graph in figure 3.2(a). Baker et al concluded that, as the code in figure 3.2(b) is shorter than the code in figure 3.2(c), therefore V (program volume) and E (program effort) will be lower for the code in figure 3.2(c).

However, the code of different linearizations of a flow graph can be equal in length, have the same vocabulary but, may be different in the ordering of the statements. Such orderings, which are the points of issue of the linearization criteria of goodness, will strongly affect the program's complexity, readability, clarity , etc.., whereas the length of the code will not detect such effects. Therefore, Halstead's metrics do not correctly reflect the real difference between different program linearisations.

3.3.1.10 Unstructuredness -

It is possible to construct a program using the following control structure:

1. Sequence

2. Selection

3. Repetition

There is a general agreement that using these structures to a certain extent will make the control flow easy to follow. Halstead's metrics do not recognise such structures. For example, if a program is written with the use of only IF ... THEN, REPEAT ... UNTIL, and DO...WHILE, the program will contain a certain number of tokens. If the program contains any other structures, the count will yield another number of tokens. In both cases Halstead' metrics will tell nothing about

unstructuredness of the program. For example figure 3.3 (a) and (b) are two different programs to sum up the numbers from 1 to 10.

Figure 3.3 (a)	Figure 3.3 (b)

```
I=0                       ISUM=0
ISUM=O                         DO 20 I=1,10
WHILE (I.LE.10) DO             ISUM=ISUM+I
ISUM=ISUM+I                    20 CONTINUE
I=I+1                          PRINT* ,ISUM
END WHILE
PRINT* ,ISUM
```

The program in figure 3.3(a) is using DO...WHILE to solve the problem. The program in figure 3.3(b) is using DO...loop, to solve the same problem. The number of occurences of operators and operands are more for the program in figure 3.3(a) than the program in figure 3.3(b), as shown in table 3.3 and 3.4, hence,the values of Halstead's metrics will be more for the DO..WHILE program than the DO..loop program, whereas the WHILE structure is much more powerful than the DO loop [96].

Table 3.3

i	operator	f1,i	i	operand	f2,i
1	EOS	6	1	I	5
2	=	4	2	ISUM	4
3	WHILE	1	3	10	1
4	END WHILE	1	4	0	2
5	PRINT	1	5	1	1
6	.LE.	1			
7	+	2			
8	(	1			
9	)	1			
10	,	1			
11	*	1			
12	DO	1			
n1=12		N1=21	n2=5		N2=13

Table 3.4

i	operator	f1,i	i	operand	f2,i
1	EOS	4	1	ISUM	4
2	=	3	2	I	1
3	DO	1	3	0	1
4	PRINT	1	4	1	2
5	CONTINUE	1	5	10	1
6	+	1	6	20	2
7	*	1			
8	,	1			
n1=8		N2=13	n2=6		N2=11

However, it may happen that Halstead's metrics have minimum values when the program is a structured one. This occurs when the number of tokens are decreased in the program. But nothing more can be inferred from the metrics. Therefore, Halstead's metrics are not sensitive to the unstructuredness criterion of goodness.

3.3.1.11 Structuring Transformations -

One of the well-known structuring transformations is node splitting. Node splitting is a technique by which duplication of code is involved to produce a structured flow graph of that code [23]. Halstead's metrics do not perform nicely when they are evaluated in the light of node splitting. This is because node splitting duplicates the code which will increase the value of Halstead' metrics. Moreover, the increase in the value of Halstead' metrics will give no indication whether the program structure is improved or not. But still the value of the metrics will be more, in case of applying node splitting transformation, which is not a good indication according to Halstead [2].

3.3.2 ABSTRACT METRICS

There are many software metrics having a theoretical basis in graph theory which have been proposed to quantify the control flow complexity of a program. These metrics are available in the software literature [5, 8, 21, 23, 37, 39, 40, 56, 59, etc.].

McCabe's metric [5] and Woodward et al's Knot metric [8] will be considered as examples for a comparison, since they have received the most empirical attention.

3.3.2.1 Applicability -

McCabe's and the knot metric are designed to measure the control flow complexity of a program. Therefore, they are applicable to the implementation phase of the software development life cycle. The output of the design phase may be the following: graphs, tables, data design, program design, module design, ... etc. McCabe's metric can be applied to some of the components of the output of the design phase, where the data flow graph and control flow graph are involved. For example, the design of each module, may be documented by a module input/output diagram, a module control flow diagram, a module data flow graph, etc., therefore it is possible to apply McCabe's metric to measure the complexity of the module control flow diagram. The result, is a certain McCabe number, which has an indication about the design phase.

The knot metric can only be applied to linearisations of data flow graphs and control flow graphs. A knot in the linearisation of a two dimensional graph occurs when two control flow lines must cross. It seems to be that both metrics may be applied to those phases of the system development life cycle where the control flow graph, data flow graph, etc., are involved. This is usually only at design and implementation phases. Both might also be used for assessing the complexity of petri-net (and similar) specifications.

3.3.2.2 Validity -

McCabe has recommended a maximum value of "10" for module complexity. This value is used as a "yardstick" for deciding that a program is big and should be split into separate routines. However, this number may be invalid in the case of large programs such as compilers, which could usually include a large case statement. Therefore, McCabe's metric becomes less relevant and the given upper limit should be re-evaluated. McCabe's metric is not realizing all the objectives which, the metric was developed to achieve. For example, adopting McCabe's suggested value of 10 as a condition for a program to be decomposed, will have no effect on a large module dealing with straight line code. Furthermore, it is generally agreed that adopting a structured programming technique will make the control flow graph easy to follow, and hence will decrease the value of McCabe's cyclomatic complexity. It is shown by [61] that some programs have a higher value in structured form than in their original unstructured form.

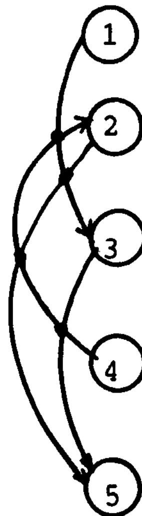
Regarding the knot's metric, where a knowledge of program control flow jumps in terms of line number is involved, it becomes very difficult for a programer to draw a mental map for a very large program. Therefore the knot's metric becomes less applicable to large-scale systems. The knot's metric was designed to measure the complexity and unstructuredness in a piece of code. The knot's metrics will always represent the complexity

by alternation in FORTRAN-like languages. The knot metric may give irrelevant indications about control flow complexity incurred by iteration, or straight line code [23]. Furthermore, the value of the knot metric will represent the unstructured form of the program. This can be achieved when the intersections caused by the use of IF...THEN...ELSE or CASE statements in the program are ignored [8]. Generally, the knot metric assesses what it was designed to assess.

3.3.2.3 Sensitivity -

Relatively, McCabe's metric is insensitive to program restructuring. This is shown by Woodward et al [8] by the example which is given in Figure 3.4(a) and (b).

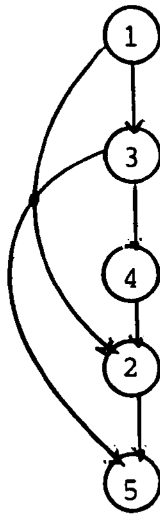
Figure 3.4 (a)



V(G)= 3

K = 4

Figure 3.4 (b)



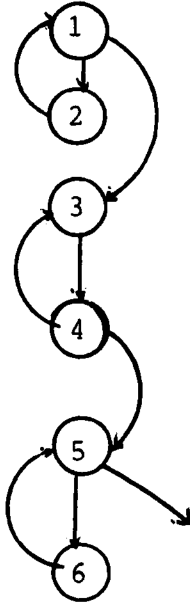
V(G)= 3

K = 1

In the given example McCabe's measure V(G)=3 for both flow-graphs before and after restructuring.

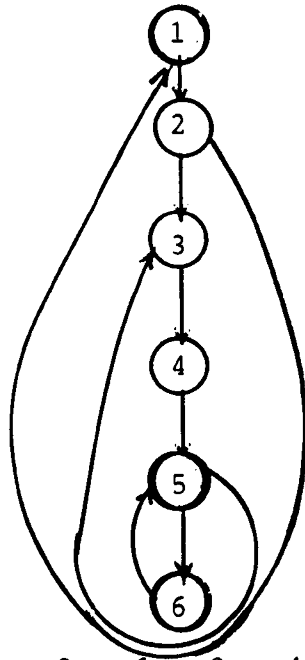
McCabe's metric takes no account of program nesting. For example, Figure 3.5(a) and (b) shows two flow-graphs for which McCabe's measure is identical. Even though it is more easy to analyse and understand the sequence of loops in Figure 3.5(a) than those in Figure 3.5(b).

Figure 3.5 (a)



$$V(G) = 8 - 6 + 2 = 4.$$

Figure 3.5 (b)



V(G) = 8 - 6 + 2 = 4.

Generally it can be said that McCabe's metric is insensitive to the changes in the structure of the system. The Knot's metric will not be sensitive to those programs which are well structured (using only inclusive IF...THEN...ELSE and DO WHILE control flow operators). Therefore, the Knot's will be insensitive to the changes in which edges crossing are not involved. Otherwise, the knot metric is sensitive to the changes in the structure of the system in cases where crossing of edges control are originated. Moreover,the knot metric is very sensitive to the unstructuredness form of a program. For example, figure 3.4 (a) and (b) shows such sensitivity. This is because the knot metric was intended to measure the unstructuredness of the program.

3.3.2.4 Procedurising -

Calling of a procedure, functional statement, module, etc., which does not generate any control transfer, should not affect the McCabe and knot metrics. This can be a valid assumption when a CALL statement which invokes a subprogram is treated as an assignment statement.

3.3.2.5 Language Independency -

McCabe's metric is language independent, since it is based on the flow graph representation of any program.

The knot metric is language dependent in detail, that is, dependent on the detailed syntax of the language. However, it is language independent in principle. This is because it is based on control flow graph edges drawn on the actual sequential source program.

3.3.2.6 Simplicity -

McCabe presents two simple ways to calculate the program complexity (V(G)) these are given below:

1. V(G)= number of predicates + 1

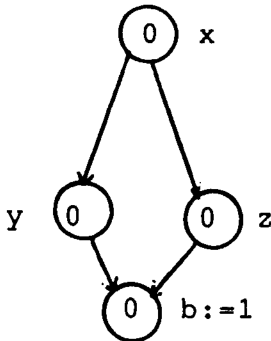
2. V(G)= number of regions in a planar graph of the control flow.

Difficulties may arise in deriving the control flow graph of a program, for it is not always clear how certain program construct should be represented. For example, in ALGOL-68, the program fragment:

<u>if</u> x <u>then</u> y <u>else</u> z <u>fi</u> ;

b:=1;

can be represented as:



but how should the segment:

a:= <u>if</u> x <u>then</u> y <u>else</u> z <u>fi</u>;

be represented. If such a construction is reordered or restructured before the control flow graph is derived then this graph represents an equivalent program. However, given a control flow graph, the above two formulae are easy to compute. Therefore, McCabe's metric is easy to obtain and it can also be generalised, so that it can be applied to any graph.

The knot metric is also easy to compute and it becomes easier when there exists an automatic software tool to scan program written in FORTRAN-like languages.

3.3.2.7 Positivity -

The value of McCabe's metric V(G)>=1 for any program, it is equal to one for a program which has only a sequential code. Hence it is non-negative.

The knot metric also has a non-negative value for any program. It is always zero for a sequential program, and will remain zero for any program as long as there is no control flow paths intersections in that program. It can be concluded from the above that the knot metric satisfies the positivity criterion of goodness.

3.3.2.8 Modularity -

McCabe has suggested a value of "10" as the maximum complexity measure for a module to be manageable. According to McCabe [5] if the complexity of a module is greater than or equal "10" a further redesign is needed for that particular module. Further, McCabe represented the control flow graph of a program as a directed graph. This directed graph can be reduced by replacing the proper subgraphs (a proper subgraph G' of G can be defined as a subgraph of G with the condition that G' not= G) in a program's flow graph with single entry and exit nodes. By continuing graph reduction until no further reduction of the graph, G, is possible, and then applying McCabe's metric, the obtained measure is the essential cyclomatic number. McCabe's essential cyclomatic complexity number can be used to discover

whether a certain program needs further modularization or not. This is can be done by applying the essential cyclomatic number technique to a module in a program. If that module is highly complex, and its cyclomatic number could not be reduced to less than or equal "10", then a further modularization would require the redesign of that module. This shows that McCabe's metric in such case can be affected by the modularity criterion. However, in some cases the relevance of the number "10", which is the upper limit of module complexity can vary depending on whether the code is sequential or deeply nested. This shows that McCabe's metric may fail to reflect the modification which is caused by modularisation.

The value of the knot metric will decrease as a result of program modularization. For example let m be the number of "knots" of a piece of code repeated n times and hence separated from a program with k knots. Since the modularization issue recommends the separation of the repeated code and and its call as a procedure, subroutine, functional statement, etc., the overall complexity of the program after modularization will be k-(n-1)m, which is always less than k. Therefore, the knot metric may reflect the modification which is caused by modularisation. However, in some cases where the complexity produced due to sequential code or deep nesting the knot metric may fail to reflect the modification which is caused by modularisation.

3.3.2.9 Linearization -

McCabe's metric does not reflect the complexity incurred by a particular linearization. This is because the cyclomatic complexity is computed using the flow graph which carries no information regarding the linearization issue. The knot metric, on the other hand, specifically makes use of the linear ordering of nodes in the flow graph to capture the control flow complexity difference. For example, figure 3.4(a) which was given before shows a program with four knots before linearization. Figure 3.4(b) shows the same program with one knot after linearization.

3.3.2.10 Unstructuredness -

Generally a structured program can be separated into one entry and one exit units. This can be done when no arbitrary transfer of control into or out of the body of such units is allowed. In this case they become easy to understand. The arbitrary transfer of control into or out of the above units will make the program unstructured. Such unstructuredness should be recognized by the control flow complexity metrics. McCabe's metric does recognise unstructured forms of a program. Consider the following program constructs:

1. SEQUENCE,
2. IF,
3. WHILE,
4. UNTIL,

5. CASE.

McCabe argues that if a program is restricted to the above constructs, then, the program will be structured. The complexity of such a program can be reduced by replacing each proper subgraphs in a flow graph of that program by a single nodes. By continuing graph reduction until no further reduction of the graph is possible, and then applying McCabe's metric to measure the cyclomatic complexity, the obtained measure is the essential cyclomatic number, which shows the program unstructuredness. This shows that McCabe's metric in such case can discover the program's unstructuredness. However, if McCabe's metric is computed by the number of predicates plus one then it may not discover the program unstructuredness. This is because McCabe's metric will be concerned with the number of predicates which may or may not be the same for structured or unstructured programs. Oulsnam G. [60] stated that McCabe's metric can not serve as a reliable measure for unstructured programs.

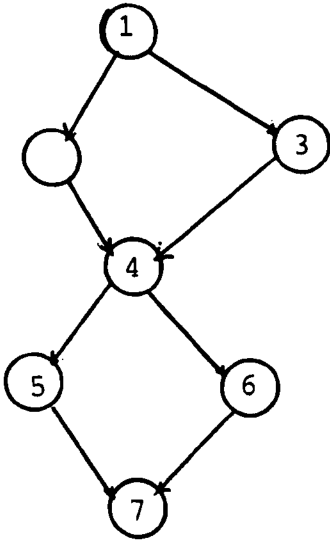
The knot metric can recognise unstructured forms of a program. This can be shown by applying the essential knot measure which is similar to McCabe's essential measure. Therefore, the knot metric can be considered as a measure of unstructuredness. This is because a structured program can be reduced to single node which contains no knots. But usually there will be unavoidable intersections due the IF...THEN...ELSE or CASE statements

alterations in the program. If these intersections are ignored, then any other intersections will correspond to the unstructuredness of a program [8].

3.3.2.11 Structuring Transformation -

Generally McCabe's metric will be affected by the duplication of the code which is incurred in the program due to a structuring transformation. For example node splitting may decrease the cyclomatic complexity of a flow graph. This is shown in the example which given in figure 3.6 (a) and (b).

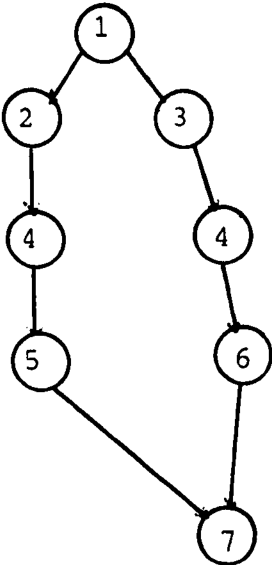
Figure 3.6 (a)



$V(G) = 8 - 7 + 2$

$= 3.$

Figure 3.6 (b)



$V(G) = 8 - 8 + 2$

$= 2$

The knot metric will also decrease with every application of a structuring transformation [38]. Thus a structuring transformation which duplicates the code can be sensed by both the knot metric and McCabe's metric.

3.3.3 STRUCTURED METRICS

The structured metrics are mainly used to measure the following:

1. the stability of a software system. This metric is developed by Haney [47] which is based on module connectivity.

2. the degree of interdepedence among the components of a program. This metric is develop by Myers [88].

3. the information flow between system components. This metric is developed by Henry et al [6].

4. the logical stability of a program. This metric is developed by Yau et al [45] and it is based on recording the ripple effect resulting from program modification.

The common factor between the above metrics is that all of them are measuring up to which level the components of a software system are connected. The researchers have attempted to use software structured metrics to predict resource expenditure during the design and the implementation phases of the software system life cycle.

Among the above metrics, Henry et al [6] claimed that their metrics have been validated on the source code of the UNIX operating system. Further, the information flow technique reveals more of the software system connections than other metrics such as [47, 88, etc.,].

For the above two reasons, it was decided in this study to apply the generated criteria of goodness of section 2 to Henry et al's metrics [6].

3.3.3.1 Applicability -

Henry et al's structured metrics are based on the measurement of information flow between system components. Certain metrics are developed to measure procedure complexity, module complexity and module coupling. The procedure complexity metric is based on counting lines of code in a given procedure and the information flow connection of a procedure to its environment. The module complexity metric is based on the sum of the complexities of the procedures within the module. The module coupling metric is based on the extent to which two modules are coupled to each other.

Henry et al's metrics [6] may apply to the design phase of the software system life cycle. This is because the major elements in the information flow analysis can be determined at the design phase. This needs a precise design language to be used. Moreover, it also needs sufficient information to generate the information flow

relations. However,the following difficulties may arise:

i. there is not a sufficient and precise known design language which can be used as a tool so that the developer will be enabled to apply such metrics to the design phase.

ii. some design methodologies use only graph notation without any details. They justify their method by saying that one graph is worth one thousand words. Thus, Henry et al's metrics may be difficult to be applied to such design methodologies.

The code length metric can be applied to an implementation phase of the software system life cycle. This is because the code length metric was defined by Henry et al [6] as " the number of lines of text in the source code for the procedure". This is equivalent to counting the number of statements in a program. However, such a metric cannot represent the full complexity of a procedure. This is because a procedure with small length but which contains predicate nesting and iterating statements is more complex than a procedure with a large length which contains simply a sequence of assignment statements. The coupling metric can be used as a tool during the implementation phase to indicate the effect of modifying a module on the other modules of a software system.

3.3.3.2 Validity -

Once a metric has been developed it must be validated on a practical software system so that it can be assured that the metric does measure what it is intended to measure. Henry et al [6] claim that their metrics were validated on the UNIX operating system. The validation effort involved a correlation of procedure complexity with the occurence of changes in the UNIX code. Henry et al [6] show that the complexity metrics obtained through information flow analysis indicate a high correlation to actual changes for the UNIX operating system.

3.3.3.3 Sensitivity -

If a software metric is to be used to assess the software system then it is essential that it is responsive to a software system change. Henry et al's metrics will be sensitive to those changes which increase the following:

a. procedure code,

b. interfaces (fan-in and fan one) between the procedures.

This is because the developed procedure metric depends on the above two factors. Moreover, Henry et al's metrics can be sensitive to those programming techniques which minimise the value of their metrics. For example, the languages which involve no procedures interfaces. However, it is not shown by Henry et al which programming techniques minimize the metrics and how these metrics

behave with such techniques. It is believed that such programming techniqes which minimize the metrics should be examined to assure that reduction in the metrics consistenly produce improvement in the program [67]. Also a programming technique which modifies the program in a desirable way with respect to an attribute must not produce undesirable side effects in another attribute.

3.3.3.4 Procedurising -

Henry et al [6] have considered the complexity of a software system produced from a small number of procedures which are heavily interfaced to each other. Their metric of information flow assigns a value of complexity to each procedure. As a demonstration of the usefulness of their metric, Henry et al applied their metric to UNIX operating system. They discovered that their metric produced a high value for those procedures which are highly interconnected. It can be concluded from the above that Henry et al's metrics heavily penalise the procedure with a large number of interconnections. This is an unpromising result, because procedurising is generally a desirable practice and it is heavily penalised by Henry et al's metrics.

3.3.3.5 Language Independency -

Henry et al's metrics deal directly with the software system connectivity by observing the flow of information among software system components. Since such information flow can be determined for any language therefore Henry et al's metrics are language independent.

3.3.3.6 Simplicity -

Henry et al's metrics are based on information flow connections termed fan-in and fan-out. Such information flows are determined by generating a set of relations for each procedure which express the flow of information through a procedure from input parameters and global data structure to output parameters and global data structures. Henry [6] stated that: "Using the relations generated on a procedure-by-procedure basis it is possible to combine these relations to form the complete stucture of information flow by a simple structure process". It may be infeasible to generate such information flow manually. This is because for a large scale system it is a time consuming operation. However, Henry et al [6] claim that the information flow method is a completely automatable process using fairly standard data flow analysis techniques which are developed in [97] and [98]. If this claim is true then it is clear that imformation flow metrics are easy to compute.

3.3.3.7 Positivity -

The value of Henry et al's metrics procedure is computed as:

P(C) = L * (fan-in *fan-out)**2.

This value is always positive. This is because L represents the procedure length expressed in lines of code which is always greater than zero. Moreover, the information flow terms are squared which is also always greater than zero. The coupling metric is computing as:

(S-->B) = (E + I) * N where;

E is the number of procedures exporting information from module A,

I is the number of procedures importing information into module B,

N is the number of information paths between the modules A and B.

From the above formula, it is certain that the value of the modules coupling metric will always be non-negative. A value of zero for P(C) may, however, be obtained for quite complex procedures if either fan-in or fan-out is zero, e.g. a zero value may be obtained for procedures which are called by users rather than called by other procedures. However, the upper limit for Henry et al's metrics is not defined.

3.3.3.8 Modularity -

It is one of the goals of modularisation to ensure that each procedure occurs in one and only one module [67]. When a procedure is located in more than one module, the modularisation becomes improper, this is because coupling will increase between the modules. The above problem can be discovered by Henry et al's module metric. This is because those procedures which violate the modularity principle will increase the value of the module metric and should be more prone to errors due to their connections to more than one module. However, those procedures which are proportionaly distributed among modules may minimise the value of the module metric. To have a minimum value of the module metric is considered by Henry et al as a satisfactory result towards modularity. However, this is not the only way to minimise their metrics. For example, the value of a module's metric will be minimum when writing a program as a single module with no procedures, and hence, no flow of information between procedures.

3.3.3.9 Linearization -

Henry et al's metrics do not measure the complexity incurred by a linearization process. This is because the value of their metrics is computed using the information flow between procedures. Such a flow of information carries data from one procedure to another and nothing regarding the linearization issue.

3.3.3.10 Unstructuredness -

Henry et al's metrics can not differentiate between structured or unstructured programs. This is because their metrics depend on procedures interfaces. Such interfaces may be the same for structured or unstructured programs.

3.3.3.11 Structured Transformation -

The structured transformation which is considered here is node splitting. This node splitting involves a duplication of code. Generally Henry et al's metrics will be affected by the duplication of code. This is because node splitting generally will increase the number of procedures connections and modules coupling and hence the value of Henry et al's metrics will be increased.

3.4 COMMENTS AND CONCLUSIONS

As mentioned in the previous chapter, the software metrics can be divided into three categories. That is, primitive, abstract and structured. The following comments may arise on the first category which is represented by Halstead's metrics:

i. Most of the linear relationships which are assumed in the software science metrics seem to be non-deterministic. For example, in a deterministic system the mass of an object is calculated as:

M = D*V

where;

D is the density of an object,

V is the volume of an object.

This relation can almost always be determined correctlly for any object. In the case of Halstead's metrics most of the relations between the parameters are nondeterministic and dependent on an estimation. Further no attempt has been made to use operational research or probabilistic models for the relationships. Even though the existence of a high correlation between the measurements of program length, volume, size and the number of bugs in the program can be demonstrated, this does not ensure that program length, size and volume are essentially good predictors of errors. Neither can it be suggested that errors can be reduced by reducing the program length, size and volume [67].

ii. The software science metrics are mainly derived from four measurements, i.e., number of unique operators, number of unique operands, total number of operator occurrences, and total number of operand occurrences. Halstead assumed that all types of operators in a program are the same whereas this is not true. Operators may have two different meanings in syntax and semantics. For example, real multiplication, integer multiplication, and logical AND. The same occurs with real addition, integer addition, and

logical OR. Further, in some languages such as ALGOL-68, it is very difficult to differentiate between operands and operators. For instance, figure 3.7 (a) and (b) illustrates such difficullty.

Figure 3.7 (a)

```
-------------

procedure sub = (ref real z,real x,y) void:
z := x + y;
sub(c,a,b);
```

(The two ALGOL-68 statements can be made equivalent to the FORTRAN assignment statement c = a + b).

Figure 3.7 (b)

```
-------------

procedure sub = (ref real z,real x,y) void:
z := x * y;
sub(c,a,b);
```

(The two ALGOL-68 statements can be made equivalent to the FORTRAN assignment statement c = a * b).

The use of these operators should be treated more carefully.

iii. Halstead tried to develop his metrics by combining theories from both computer sciences and Psychology. Some of the psychological assumptions used in his work are difficult to justify for the purpose for which he applied them. For example, Halstead considered the metric of the time equation T to be a good estimate of time needed by a programmer to generate a single module in a program. This metric is derived analogously to a hypothesis in Psychology presented by John Stroud [94]. He states that the mind is capable of making a limited number of elementary discriminations per second. This number is denoted by S and it ranges from 5 to 20. Halstead selected S to be equal to 18 claiming that this number gave the best results during experimentation. However, the Stroud number which is used to derive the time equation metric has not been generally accepted among the Psychologists [43]. Indeed, Curtis [1] states, "computer scientists would do well to immediately purge from their memory the Stroud number 18 mental discriminations per second". This is because some complicated tasks such as program generating and understanding may not be restricted to the suggested number.

iv. According to Hammer et al [22] the empirical results of Halstead's metrics are inaccurate.

The second category of software metrics(those which are based on the graph theory) are related to the order in which the various statements of a program are executed. Any alteration in the program statements sequential flow can be used as a measure of control flow complexity. Some of the control flow metrics are related to the important criteria such as the number of errors present in a piece of code and the time available to find and correct these errors [3]. Static measures have been created, in terms of which the source programs are analysed, and the software metrics are obtained and quantified. The researchers have also attempted to create a dynamic measure by introducing data flow, data structure, and analysing the program performance during execution.

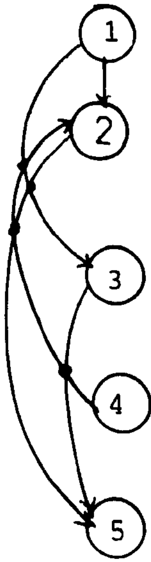
Examples from this category are McCabe's metric and the knot metric. McCabe's metric can be computed easily by the formula:

V(G) = number of predicates plus one

in a well structured program.

The ordering of the program constructs is not important in McCabe's metric. It cannot detect the difference between two arrangements of a program's construct. The arrangement of the program's construct in a certain order may improve the program structure. This difference can be detected by some other metrics such as the knot metric etc.. This is shown in the example given in figure 3.8 (a) and 3.8 (b)

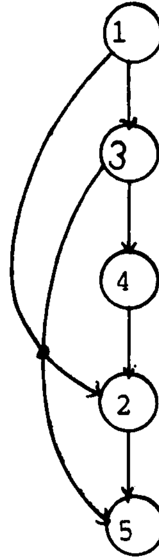
Figure 3.8 (a)



V(G)= 6-5+2=3

K=4

Figure 3.8 (b)

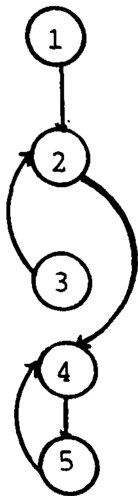


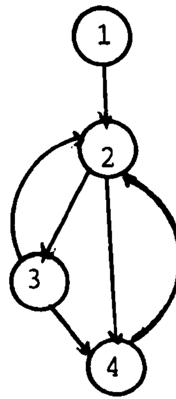
V(G)= 6-5+2=3

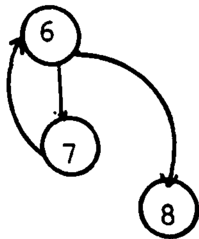
K=1

Additionally the structure of certain programs could be better than others but they could have the same McCabe's measure. For example, the two flow graphs which appear in figure 3.9 (a) and (b) depict the structure of two programs. The graph in figure 3.9 (a) appears to be more structured than the graph in figure 3.9 (b), even though they have the same McCabe's cyclomatic number = 4.

Figure 3.9(a). Figure 3.9(b).







$V(G) = 6 - 4 + 2 = 4.$

$V(G) = e - n + 2*p$

$V(G) = 10 - 8 + 2 = 4.$

McCabe's metric is language independent. It does not distinguish between the structured and unstructured programs.

The knot metric which is based on control flow jumps has the following defects:

1. it does not show any program complexity incurred by the straight line code of the program,

2. it is language dependent in detail but not in principle for example, the COBOL performed paragraph increases the number of knots essentially.

The common weak points of the second category metrics are the following:

i. their relevance decreases for large scale systems,

ii. they have considered the correlation coefficient to be the sole measure of the relationship between two variables. For example, the correlation between errors and complexity, errors and the size of the code, etc., whereas a high correlation indicates only that some relationship exists, but does not show what the relationship is. Hamer et al [22] have warned of the pitfalls in using the correlation coefficient to confirm an identity relationship between two variables.

iii. no clear distinction is drawn between different types of bugs and the origin of those bugs. Jackson [10] has showed that many bugs originate from a mismatch between the structure of the problem and the structure of the solution of the problem.

The final category of software metrics has certain advantages for a manager's overall understanding of system complexity and its impact on system costs and performance. For instance, the metrics [42] which is measuring interconnectedness among segments of a software system will enable the manager to predict the maintenance cost of the software system. This is because such metrics deal with a macro-level of the system which may

be easy to understand [42]. Although the last category is based on structure properties of software design and seems to have some strong attractions, there is not sufficient research to give a true assessment of its value. The general problems which may arise with structured metrics are the following:

1. for a large system model validation is difficult, because the model inputs have not been automated,

2. the assumption that all modifications to a module have the same ripple effect is not relevant.

Generally it can be concluded that, no approach at present can be considered as a standard and true measure of software systems. The metrics which are available now are not sensitive to errors. Metricians never show the negative results of their metrics whereas positive ones are published proudly.

CHAPTER FOUR

CHAPTER 4

SOFTWARE QUALITY

4.1 INTRODUCTION

Software quality is one of those terms in Software Engineering which has not yet been defined precisely and properly for all the purposes despite numerous attempts. In recent years software developers have made considerable efforts to define the quality aspects of software systems such as [29], [46], [70], etc.. In section 1.1 below, there is a brief critical review of the definitions and concepts of the term software quality which are used or defined by various authors.

This chapter is divided into six sections. Section 4.1 contains the introduction and several definitions of the term software quality defined by various authors. This section also contains the criticism of these definitions. Section 4.2 contains the suggested approach to make software quality more clear and visible. This section splits into another subsection that is, 4.2.1 which contains the definition of the various terms used in this study. In Section 4.3 efforts have been made to identify and define 60 important software quality

attributes. These attributes may have impact over the quality of the software system. Section 4.4 contains the "internal" views of software quality. In section 4.5 a detailed quality plan is presented which contains the following phases which will be described later.

a. software quality requirements phase,

b. software quality factors phase,

c. a software quality model phase,

d. validation and verification phase.

Finally, in section 4.6 conclusions of the overall chapter are presented.

4.1.1 Definitions And Criticism

The IEEE standard glossary [13] has defined the term software quality as:

i. "the totality of features and characteristics of a software product that bear on its ability to satisfy a given need".

ii. "the degree to which software possesses a desired combination of attributes".

iii. "the degree to which a customer or user perceives that software meets his or her composite expectations".

iv. "the composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer".

In the above definitions, certain terms or words are used without definition. For example, the terms "characteristic" and "attribute" have been used without giving its definition in the glossary. This may confuse the users, management and developers.

Jones [27] has used the term quality to denote the absence of defects in the software. He uses error detection and removal tasks to indicate a certain quality. He suggested two ways to minimise program defects:

a. prevention of defects from occurring,

b. detection of defects and removal.

From Jones analysis it can be concluded that a better software system will be a result of less defects in a software program. From this it is clear that Jones regards a defect free program as being high quality. Other parties may disagree arguing that say usability is more important. Further, prevention and detection of defects is not sufficient alone without the quality management. Frewin et al [26] stated that " while an effective prevention programme can thus reduce the overall cost of quality management, it should never be used as a reason for dismissing the need for such management: faults will always be with us, and it will always be an asset to be able to increase confidence in the belief that a process or a product has few if any faults -without a full quality management programme, there can be no such confidence". However, it is

important in defects prevention and detection to apply quality assurance techniques and try to limit the propogation of defects through the different phases of the system development life cycle. For example, the defects wich occur in earlier phases should be corrected soon (if possible) before moving to the further phase in the life cycle. Some defects for example, may be avoided by adapting a specific programming style. For example, structured programming.

Yourdon [7] has divided up a software system into a collection of interacting modules. He pointed out the interrelationships between the modules, and suggested certain criteria for evaluating the quality of a software design. According to Yourdon's criteria, the quality of a module is classified into seven levels of cohesion. These levels are functional, sequential, procedural, communicational, temporal, logical and coincidental. For example, one of his criteria to determine the functional cohesion module is to sum up a module's intended function by a precise verb-object name as, READ, CALCULATE, DEDUCT, etc.. He considered a high quality software system design, as one which consists of modules having a high degree of functional cohesion. However, Presland S. G. [11] argued that generally identifying software functionality by simple method such as the one given by Yourdon is not successful in a large system. Also, the other criteria which are given by Yourdon to determine the other module cohesion levels are difficult to apply

due to the subjective nature of the definition of these module cohesion levels. Further, Yourdon does not consider the role of defects.

Boehm et al [29] have not given any specific definition of software quality. However they have developed a hierarchy of software quality characteristics. This hierarchy is shown in figure 4.1. At the top level of this hierarchy is the general utility of a software product which reflects the actual uses to which software quality is evaluated. Then the intermediate level of the hierarchy represents the quality characteristics such as portability, reliability, efficiency, etc., which are important for a certain software system. At the lower level of the hierarchy are certain software quality characteristics, called primitive characteristics such as, accessibility, accuracy, etc.. These primitive characteristics are used as a set of necessary conditions for the intermediate level characteristics.

Figure 4.1 software quality characteristics tree

```
01 General Utility
   02 maintainability
      03  testability
          04  accountability
          04  accessibility
          04  communicativeness
          04  self-descriptiveness
          04  structuredness
      03  understandability
          04  legibility
          04  conciseness
          04  structuredness
          04  self-descriptiveness
          04  consistency
      03  modifiability
          04  augmentability
          04  structuredness
   02 as-is utility
      03  reliability
          04  self-containedness
          04  accuracy
          04  completeness
          04  robustness/integrity
          04  consistency
      03  efficiency
          04  accountability
          04  device efficiency
          04  accessibility
      03  human engineering
          04  robustness/integrity
          04  accessibility
          04  communicativeness
   02 portability
      03  device-independence
      03  self-containedness
```

In such a hierarchical structure the characteristics are related in a one way direction. For example according to Boehm [30] if a program is maintainable this implies that it is also understandable and testable. Figure 4.2 illustrates this relation.

Figure 4.2

```
maintainability------>understandability
                \-----> testability
```

In the given example the arrows are restricted to a one way direction.

The above claim is not necessarily true. It can be easily shown that the relationships between the characteristics of two different levels may be interdependent. It depends on which characteristic the viewer chooses to stress. For example, understandability may be more important than maintainability. Thus, the characteristic understandability may appear at the top level of the hierarchy and maintainability may appear at the intermediate level of the hierarchy. This is because if a software system is understandable this could imply that it is also maintainable. It is also possible that the characteristic testability may appear at the top level of the hierarchy and the characteristics understandability and maintainability at the intermediate level of the hierarchy structure. This is because a software system which is testable is also maintainable and understandable. Figure 4.3 and 4.4 illustrate such interdependency relationships.

<u>Figure 4.3</u>

```
understandability----->maintainability
```

<u>Figure 4.4</u>

```
testability--->understandability
              \-->maintainability
```

A similar study, to define software quality aspects has been carried out by McCall, Richards and Walters [46]. They have defined quality as: "a general term applicable to any trait or characteristic, whether individual or generic; a distinguishing attribute which

indicates a degree of excellence or identifies the basic nature of something". In the above definition, McCall, Richards and Walters have not given any definition for the terms characteristic and attribute which are used in their definition.

A recent study has been carried out by Bowen, Wigle, and Tsai [79], to explain software quality. It is observed that Bowen et al have not explicitly defined the term software quality. However, they have increased the number of software quality factors. This was due to an expansion in the software quality criteria and software quality metrics.

Garvin D.A.[92] has synthesised five various distinct views of product quality. He derived these views from philosophy, economics, marketing, and operation management. These views are:

a. the transcendent approach of philosophy,
b. the product-based approach of economics,
c. the user-based approach of marketing,
d. the manufacturing-based approach and
e. the value-based approach of production.

These views are entirely general and not related to any particular product. However, it needs more care when applying them to a software product.

Kitchenham et al [70] have extended these views against a background of software system development products and they have given wider, more useful and

general views of software quality. These views are described below briefly:

i. transcendent view: the quality in this case cannot be measured, but it can be felt and recognized through experience. An example is hi-fi music.

ii. product-based view: the quality in this case is related to the ingredients of the product. This implies that higher quality can be obtained at higher cost.

iii. user-based view: the quality here is related to the user requirements and user needs. The primary focus of this view of quality, is clearly, external to the producing organisation.

iv. manufacturing-based view: in this case the quality is related to certain specifications which must be available in the product.

v. valued-based view: quality in this case is composed of the two previous views, that is, manufacturing-based view and user-based view.

However, Kitchenham did not give any specific, explicit, or precise definition of the term software quality. Further, even within one of these views there are further different perceptions of quality. For example, consider the user-based view in which user requirements, may be maintainability, usability, etc.. These qualities may have further such classification, that is, the external

view and the internal view. For example, an internal view of maintainability may be knowing how many paths contain a particular statement, whereas an external view can be the time to fix a bug in a piece of code.

Generally, software developers, users, management, etc. deal with the external views of the software quality. Few people have attempted to consider the internal views of software quality. Therefore, one of the objectives of this study, is to deal with quality from an internal viewpoint.

4.2 DESIRABLE APPROACH

Research into software quality is still in a state of flux. Software quality researchers have defined and used the concept and term of quality in its general and external meanings, where by external is meant the view of users/buyers of software, and the internal view relates to the structure or construction of the software. However, few researchers have considered the internal views of software quality. The last point and the discrepancies in the concept and the term quality and its use compelled the author to develop a uniform, consistent, unambiguous and comprehensive approach by which software quality may become more meaningful and visible. This approach consists of the following:

i. the development of clear definitions for the terms software quality, quality characteristic, software quality property, software quality attribute, software quality factor, software quality criteria, software quality metric and software quality plan.

ii. identification and definition of various quality attributes,

iii. giving the internal views of software quality,

iv. development of a software quality plan.

This approach will enable the developers to understand the concepts of software quality and help them to apply and control it according to the expectations of users, management, customers etc.. The clear definitions of these terms may help to prevent the misinterpretting of these terms. These definitions will also serve as a touchstone against which new ideas can be tested.

4.2.1 Definition Of The Used Terms

The terms software quality, quality characteristic, property, attribute, factor, criteria, metric and quality plan are used frequently by various authors of Software Engineering. For example, in the IEEE glossary [13], McCall et al [46], Kitchenham et al [70] etc. These terms are being used in various different meanings. This is because, some of these terms are very difficult to define and in particular very difficult to distinguish.

The study of the software quality literature revealed that these terms are used without clear and explicit definitions. Therefore, there is an acute need to have clear definitions of these terms which are commonly used when discussing software quality.

The following are proposals for the definitions of some important terms related to software quality. These definitions have arisen from papers in the software literature such as [29, 32, 33, 34, 70, 99, etc..] The definitions supplied are either inferred from the original documents or are supplied by the author and coworkers. These terms are used consistently in this study. Moreover, a glossary of some of the used terms are defined in appendix [A]. Any term which is not defined here, is used in its conventional meaning.

4.2.1.1 Software Quality -

Software quality, or quality as it shall be termed hereafter, is an abstract concept. Quality can be defined as a degree of excellence which depends on a number of characterised properties, e.g. reliability, modularity, readability, etc.. These characterised properties when possessed by a software system, will enable the software system to satisfy certain constraints which may be imposed by its developers, customers and users. For example, the constraints may be the cost, construction time and conformance to certain predefined specifications, etc.. Following Garvin [92], quality

will be classified into:

a. transcendent view,

b. product-based view,

c. user-based view,

d. manufacturing-based view and

e. value-based view.

The internal view which is of interest here is primarily by not wholly related to the manufacturing-based view.

In order to make quality more tangible certain properties are attributed to it. These properties or attributes depend on the viewpoint or interest of specific observers. It will be assumed in this study that attributes and properties are equivalent. These attributes will also be referred to generically as qualities. In order for properties to become quantifiable or measurable they must be characterised. That is, a description, notion or similar device must be used in order to provide a basis for characterisation. For example, if length is such a property then it can be characterised by, lines of code, number of lexemes, number of characters etc.. Moreover, it is possible to have a number of properties which reflect a given quality perspective and each property may be characterised in different ways.

In common with other areas of sciences it is appropriate to refer to a characterisation as if it is the property itself in cases where the distinction is not significant. Thus, line of code in the above example may

be considered in general to be a property of a software system. The distinction only needs to be made when the details are under consideration. Frequently, there are properties of quality which are of considerable interest but which are too complex for direct consideration. In these cases it is often possible to discern factors which clearly contribute or relate to the property of interest. For example, readability is affected by the factors such as legibility, vocabulary, type font, etc.. In general, each factor must itself be characterised with the informal understanding that references to the characterisation will usually be understood to refer to the factor itself. Many of these factors may be assessed in more than one way. For example, with the type of font there are issues of size, style and density, etc.. Thus for each factor there may be a number of metrics and for each metric there must be some criteria which must be satisfied. These criteria are predefined standards against which the quality factors may be judged [77]. The incorporation of the required level of these quality factors into the software development life cycle can be achieved by constructing a quality plan.

Sometimes certain characterised properties, such as reliability, maintainability, etc., may be used as a definition of product quality [91]. Sometimes qualities can be meaningful only before or after certain time. For example, feasibility is meaningful before the establishment of the system, and reliability is

meaningful when the system is working [80].

To make the software system successful, reputable, and compliant with required standards, the following issues must be considered:

1. the quality must be managed,
2. the achievement of the quality must be the responsibility of every department and every individual in a given organisation,
3. there must be a continuous search for quality improvement. However some companies stop improving the quality of their products when they feel that their products prove reputable and have excellent markets,
4. quality must satisfy all the desired attributes,
5. the quality should be considered first while developing a software system keeping into account the long term profit.

4.2.1.2 Software Quality Characteristic -

A characteristic can be defined as: a term used to give form, notion, description etc., to some abstract concept. It is very essential to characterise the software properties so as to determine and to evaluate the degree or level to which software system approaches the desired qualities. An example of these properties

are; maintainability, reliability, etc..

Note: In other sciences, characteristics are used as if they were properties themselves.

4.2.1.3 Software Quality Property -

A software quality property is defined by Kitchenham [105] as: "a non-functional feature of a software system which is exhibited to an extent(i.e. to a degree). The extent to which a software system exhibits a particular software quality property bears on its ability to meet given needs".

This use the word "quality" in the above definition does not reflect the concept of quality as the sum of characteristics by which a thing may be identified, neither does it equate quality to inherent worth. It is related more to the concept of grade, whereby products, processes or services intended for the same functional requirements may meet different sets of needs [104]. It should be noted that the above definition does not exclude price and delivery time from any list of software quality properties.

4.2.1.4 Software Quality Attribute -

A software quality attribute is defined by Kitchenham [105] as: "a software quality property which is characterised in abstract terms only, (i.e. by a verbal definition which does not include directly

observable or measurable properties)".

The above definition equates software quality attributes to what Boehm et al call "intermediate quality characteristics" [28], and McCall et al call "quality factors" [46].

4.2.1.5 Software Qualities -

Software qualities is a generic term used to refer to a group of software attributes. The term "a software quality" is used as a synonym for " a software quality attribute" [105].

Software qualities is not the prural of software quality. Software quality, as it defined in the IEEE glossary, cannot be used in the prural, neither can it be qualified by the indefinite article.

4.2.1.6 Software Quality Factor -

A software quality factor is defined by Kitchenham [105] as: "a property which can both be characterised in terms of one or more directly observed and/or measurable features, and be related to some facet of a software quality attribute".

This definition implies that both software quality attributes and software quality factors are inherently multi-dimentional. There is no implication that a software quality attribute can be completely described in terms of its related software quality factors, neither is there any implication that a software quality factor relates to only to a single software quality attribute. The above definition equates software quality factors to what Boehm et al call "primitive quality characteristics", and what McCall et al call "quality criteria".

4.2.1.7 Software Quality Criteria -

Software quality criteria is defined by Kitchenham [105] as: "software quality criteria is the plural of software quality criterion. A software quality criterion is that value or characteristic which determines whether or not a particular feature of a software quality factor conforms to the requirements set on it".

For software quality factors which are characterised in terms of measurable features, the software quality criteria correspond to the measurement values required (i.e. targetted/planned) for that feature. A software quality factor can be said to have reached its required "level", if all the feature that relate to the software quality factor conform with their respective software quality criteria [105].

4.2.1.8 Software Quality Metric -

A software quality metric is defined by Kitchenham [105] as : "a quantitative feature of a software product or process used to characterise a software quality factor. A software quality metric must be named, and defined in terms of the software products and processes to which it is relevant, its units, and measurement collection procedures. A software quality metric value may be obtained by applying the measurement procedures to the relevant software product or process".

N.b a software quality criteria is a software quality metric value obtained not by measurement, but by reference to a standard, a plan, a requirement, or a target in order to judge a value obtained by measurement [105].

A software quality metric can be an objective one where numerical values can be obtained (e.g. McCabe's metric [5]). Alternatively, it can be based on a

subjective ranking scale (e.g. module cohesion and coupling [7]). As an example of these definations consider the software quality attribute "complixity" which may be defined as "the degree of intricacy of a software system or its components". A quality factor related to complexity may be "structured design complexity" and defined as "the intricacy of a structure diagram of the call relationships between system procedures" and characterised in terms of the maximum fan-out from procedures". The software quality metric providing a measure of "maximum fan-out" is the maximum number of lines emanating from a box in a structured diagram of the system. The software quality criteria used to judge the value observed in a particular system must be determined in advance by the project or quality manager, the developers and the users. It may for example be set at 7. Failure to limit the maximum fan-out to 7 or less would prompt a redesign of parts of the system (e.g. introducing an extra level, with associated procedures, into the design). It should be noted that "structured design complexity" might also be characterised in terms of the "average fan-out from procedures", or indeed any other metric related to structured charts, in each case a different software quality criteria would be necessary to judge the observed metric values, and by inference to judge the software quality factor.

There is still some potential difficulty in distinguishing between the definition of software quality factors and identification of software quality metrics, but it is believed that these definitions have provided a useful starting point for the research reported in this thesis, and it is hoped that they provide a basis upon which more refined definitions can, if necessary, be built.

4.2.1.9 Software Quality Plan -

A software quality plan may be defined as an aggregate of methods, procedures, approaches, constraints, measures and metrics, etc., which can guide in developing a software system which is adequate and achieves a specific set of software quality factors.

4.3 IDENTIFICATION AND DEFINITION OF VARIOUS QUALITY ATTRIBUTES.

The effort is made to identify and define certain important software quality attributes. During this effort 60 possible quality attributes were defined. These definitions are given in appendix [B]. Considerations were made to include all the attributes which can improve or effect the quality of a software system to any extent and at any phase of the system development life cycle. It was observed that out of these 60 attributes some of them are synonyms. The list of these synonyms of the software quality attributes are

generated in this study. These synonyms are in terms of their general meanings as defined in [82]. This list of synonyms will help the user and developer/management to communicate easily without misinterpretation of the meaning of given attributes. The list of synonyms is given in the appendix [D]. In addition to providing definitions of the quality attributes and list of synonyms, it was also decided to investigate the extent to which experienced software engineers were familiar with, and consistent in their use of, software quality attributes. This was done by asking coworkers in Liverpool University and LDRA to indicate what they understand to be the nature of the relationship between each of the different software quality attributes. The relationships they were asked to consider were:

* independency,
* interdependency,
* direct dependency,
* indirect dependency.

Independency occurs if two software quality attributes are mutually independent, e.g. readability might be judged to be independent of accuracy and vice versa.

Interdependency occurs if two software quality attributes are mutually dependent on each other, e.g. error tolerance might be judged to be dependent on robustness and vice versa.

Direct dependency occurs if a software quality attribute depends on another, but the convers is not true, e.g. accuracy is necessary condition for correctness, but the reverse is not true.

Indirect dependency occurs if a software quality attribute occurs as a by-product of other software quality attributes, e.g. reliability may be viewed as a side-effect of correctness, testability and understandability.

The coworkers who took part in the experiment were each given a relationship matrix, and asked to identify the nature of the relationships among the software quality attributes as shown in appendix [E]. They were not, however, shown the definitions of the software quality attributes given in appendix [E], nor were they shown the list of synonyms shown in appendix [D].

A composite response was constructed from each individual response by including all the relationships which were agreed unanimousely. The composite response is shown in appendix [E]. The following points may be observed by inespection of the composite relationship matrix:

1. the relationship matrix is internally inconsistent, i.e. interdetermancy relationships which should be, by definition, symmetrical, are not so in all cases, and the dependency relationships which should be, by definition, asymmetrical, are symmetric in some

cases,

2. the relationships identified in the relationship matrix are not consistent with the definitions in appendix [B] or the synonyms in appendix [D]. For example, portability and transportability which are defined in very similar terms and are shown as mutual synonyms in appendix [D], are identified as independent in the relationship matrix. In addition, reliability and usability which have very different definitions, and no synonyms in common are identified as interdependent.

It can be concluded from this experiment that:

a. software quality attributes are not self-defining,
b. software engineers will use their own unstated definitions of software quality attributes in the absence of agreed standard definitions,
c. unstated definitions will be mutually inconsistent.

4.4 INTERNAL VIEWS OF SOFTWARE QUALITY

The quality of a sofware system may have at least the following internal views. These views are related to sofware systems:

i. information-based view: the quality of a sofware system, in this case, is related to certain attributes which make the software system well documented and informative. For example, attributes such as accessibility, communicativity,

comprehensiveness, self-descriptiveness, etc.,

ii. relationship-based view: the quality of a software system is related to certain attributes, which convey relationships between the components of the software system. For example, cohesiveness, connectivity, consistency, intraoperability, modularity, etc.,

iii. performance-based view: quality in this case, is related to certain attributes which indicate that how the software system does its tasks effectively. For example, accuracy, conciseness, efficiency, predictability, stability, etc.,

iv. variation-based view: the quality of a software system is related to those attributes which make the software system capable of being changed. For example, adaptability, augmentability, changeability, elasticity, expandability, maintainability, modularity, etc.,

v. safety-based view: the quality of the software system is related to certain attributes which prevent a product induced accident which may be effect human life, welfare or property. The attention is directed to a product which is accident free in operation and to any indirect impact upon safety - the so called side effect -. For example, the quality of a medical software system which shows drugs interactions and side effects is related to attributes such as integrity, security, trustworthness, etc.,

vi. generality-based view: the quality of a software system is related to certain attributes which make the system capable of being used for various classes of problems or environments. For example, flexibility, generality, portability, reusability, etc.,

vii. time-based view: the quality of a software system is related to those attributes which have a temporal significance. In this case, the attributes are meaningful at a particular point of the system development life cycle. For example, feasibilty, reliability, etc..

4.5 THE DETAILS OF THE SOFTWARE QUALITY PLAN:

4.5.1 Introduction:

The complex nature of software quality is one of the reasons that software systems are difficult to build according to the expectations of the users, management and developers. Without a software quality plan, the user/ management/developer may not know how to establish and communicate the quality requirements for a software system. The user generally does not know which software attributes should be selected to achieve the desired level of quality. The project manager may not able to determine which/what quality attributes have been achieved in his or her software system . Frewin et al [26] has discussed the quality plan in brief as a part of

quality management without giving any definition of the quality plan or any detail.

The suggested quality plan in this study can guide which quality attributes should be incorporated in a software system to achieve the desired purpose. For example, a quality plan especially addressing tactical software will contain quality attributes which are not common to general software such as pay-roll system. Therefore, it is very essential to establish a software quality plan in advance to help in such problems as well as in other problems. The software quality plan as defined earlier is an aggregate of methods, procedures, approaches, constraints, characteristics, attributes, etc. which can guide in developing a software system which is adequate and achieves a specific set of qualities. This is an advance preparation of all the actions necessary to obtain a satisfactory product.

This quality plan can be used for the establishment of software having desired qualities. This is because this quality plan gives guide lines to choose quality attributes which pertinently contribute to the quality of the software system. This quality plan may help to achieve the possible quality which is acceptable to the customer at the price he is prepared to pay, and at certain wanted time.

This quality plan is mainly concern with the software systems, but it may be used for the other systems. This quality plan may also be useful to the software quality researchers.

4.5.2 Purposes Of The Quality Plan:

The purposes of this quality plan are given below:

i. to develop a structure, set up or skleton which can help the management and software system developers in obtaining a desired level of qualities of a software system,

ii. to provide clear strategy, policy and criteria to select the quality attributes in order to achieve software of desired qualities,

iii. to provide the ways, tactics, means, etc. which can incorporate the selected quality attributes and can obtain a software of desired qualities,

iv. to provide a frame work and uniform guide lines which can direct the software developers to examine the possible effect or behavior of the quality attributes.

v. to provide a conceptual shape against which the delivered software system can be examined.

4.5.3 The Clarification Of The Terms Used In The Quality Plan

To have clear definitions of terms and concepts which are used in the quality plan are very essential. This is because different parties i.e. users, managenent or developers may differ in their interpretation of these terms. This may particularly happen when language and definition of these terms are not clear. The following are essential precautions which should be taken into consideration while developing a quality plan:

i. certainity of terms i.e. the quality requirements must be stated in a precise, definite and clear terms,

ii. consistency of terms i.e. the same terms must be used for the same purpose to refer to characteristics, factors, etc. through out the quality plan,
Note: most of the other people are making such mistakes e.g. Boehm [29], McCall [46], etc.,

iii. do not use terms which cannot define an exact level of performace i.e. qualities must be specific and capable of measurement to the accuracies specified. For example, terms like highy reliable, highly accurate, good performance suitable for the purpose intended etc. should be avoided [103]. However, there are some

attributes which are impossible to measure in a universally expected way. Such attributes can be excluded from this restriction. For example, accessibility, readability, etc..

4.5.4 Tools And Support

In this section there is brief discussion about the tools and support which can be used in this quality plan. The following tools and support are suggested to be used in this quality plan:

a. a list of the defined attributes,

b. synonyms list of the quality attributes,

c. relationship matrix between the quality attributes,

d. classification list of the quality attributes,

e. further classification list of the quality attributes.

4.5.4.1 The Defined Attributes -

The definitions of these quality attributes may help to prevent the misinterpretting of them. These definitions will also serve as a guide book to the developer/user/ management against which new ideas can be tested. The definitions of these attributes are given in appendix [B].

4.5.4.2 Synonyms List -

A list of synonyms for the defined quality attributes is given in appendix [D] for the purpose of use in this quality plan. These synonyms are in terms of their general meanings as defined by [82]. This list of synonyms will ease the communication between user, management, developer, etc. without misinterpretation of the meanings of given attributes. As it stands in appendix [D], the synonyms list should not be used to covert a user's requirements autamitically into the developers terminology because it does not indicate which of the natural language synonyms are rendered inapplicable by the particular definition given to a term within the context of software quality attributes. For example, the definition of stability given in appendix [B] is " The extent to which software system is capable of resisting the changes that are made to its components". In appendix [D], the natural language synonyms for stability are given as consistency and uniformity. It seems clear that the stated definition of stability does not include the concept of uniformity. It therefore important that a user is not led to be believe that a requirement for uniformity is being met by the developers working towards achieving stability.

4.5.4.3 Relationship Matrix -

The concept of a relationship matrix to indicate the nature of relationships between software quality attributes is a potentially powerful tool to assist product managers and developers to determine whether user's software quality attribute requirements are feasible, and to develop a plan to achieve the desired attributes.

The results of the experiment described in section 4.3 make it clear that to develop a practical relationship matrix which could be used as a project management tool, it is necessary work from agreed software quality attribute definitions. Production of a complete and consistent relationship matrix for the software quality attribute definitions in appendix [B] is, however, beyond the scope of this thesis.

4.5.4.4 Classification List Of The Quality Attributes -

The quality attributes are classified in this study with respect to three main phases of the software system development life cycle. That is, analysis, design and implementation phases. The reason for this is to investigate whether quality issues change as the life cycle progresses. All the terms have been extracted from literature addressed solely to the each phase.

a. Analysis Phase Attributes

The analysis phase attributes have been developed by

Shaikh M.U [95] as a part of the ALVEY project test specification and quality management. The definitions of these attributes can help the developer to have a clear concept about the quality at the analysis phase. Moreover, the developer will be able to identify which attributes can be included in the analysis phase. Further, the developer may assess the progress of the work being done so far by quantifying these attributes (if possible). These attributes are given in the appendix [F].

b. Design Phase Attributes

The design phase attributes have been developed in this study by the author. These design attributes have been observed in a set of Software Engineering documents such as [7, 17, 18, 19, 24, 34, 36, 62, 78, 85, 89, etc.,]. These attributes have been collected, defined and interpreted with respect to the design phase viewpoint. These attributes can be used by the developer to have a clear idea and definition about the software qualities at the design phase. Further, these attributes can be used as a basis to judge the quality of the design phase of a software system. These attributes are given below:

1. Documentational attributes

D1. Accessability — the degree of ease with which software components(modules, subsystems, etc.) can be accessed while looking at the design documentation. For example the extent to

which the nodes of the data flowgraph have to be visited. The most accessible design might be the one which tends towards a tree structure.

D2.Communicativeness — the level to which software design is able to convey and transmit informations, ideas, concepts, etc..

D3.Comprehensiveness — the extent to which all the essential information is available in the design of a system.

D5.Legibility — the clarity of software design e.g. environment, objects and their presentation, etc..

D6.Readability — the degree of ease with which the design can be read.

D7.Self-descriptiveness — the extent to which software design contains adequate comments about what the design is, and what is its purpose. For example the design of the a system must show enough information to the user

to determine its objectives, assumptions, constraints, etc.

2. Relational attributes

R1.Cohesion

the extent to which the software design components, activities, functions, purposes etc., are related to each other within a module, subsystem, procedure, etc. The higher the cohesion the better the design.

Troy et al [89] have given the following features as a measure of cohesion

1. the number of interrelations(effects) listed in the design document within a module,
2. the average fan-in in the design structure chart,
3. the number of possible returned values.

R2.Complexity

the extent of intricacy of the system design. Factors influencing the design's complexity may include the scope of control of each module and the relative size of the design.

The following are some of the complexity metrics for a design:

1. the maximum depth of the design's structure chart,
2. the average fan-in in the sructure chart,
3. the number of selections, and loops in the structure chart.

R3.Connectivity — the extent to which the software design components, activities, function, purposes, etc. are easily linked together.

R4.Consistency — the extent to which the software design components contain a uniform notation.

R5.Coupling — the extent to which the software design modules are interrelated. The degree of coupling depends on how complicated the connections are and the type of the connections.

R6.Modularity — the extent to which the software design can be decomposed into smaller units provided that a change in one unit has minimal impact on other units.

R7.Structuredness	the extent to which the software design is using certain techniqes which will reduce complexity and improve clarity.
R4.Traceability	the ability to trace the consequences of possible changes in the design.

3. Performance attributes

P1.conciseness	the extent to which a software design is free from unnecessary details.
P2.Effectiveness	the extent to which a software design performs its task completely and efficiently.
P3.Predictability	the extent to which the software design predicts the attributes of the software products.

4. Change attributes

C1.Adaptability	the extent to which software design is capable of being changed for other purposes, without having any ripple effects.
C2.Augumentability	the extent to which additional features can be added to the design.

C3.Modifiability — the capability of accommodating unexpected changes.

5. Evaluation attributes

E1.Auditability — the extent to which the "entities" and their "actions" can be checked or audited in order to evaluate their outputs or effects.

E2.Measurability — the extent to which a software design is capable of being measured in terms of number of boxes, fan-in, fan-out, in a structure chart, etc..

6. Generality attributes

G1.Flexibility — the extent to which a software design is capable of being expanded, with respect to some environments.

c. Implemetation Phase Attributes

The implementation phase attributes have been developed by Hennell M.A [34]. These attributes may be considered as a superset of the analysis and design phases attributes. This is because most of the software quality attributes are more meaningful in the implementation phase. These attributes are defined in the terms of their general meanings which are applicable to the implementation phase of the software system development life cycle. These attributes can be used by the software

developer to have a general idea about the qualities of the software system which may be necessary at the implementation phase. Further, some of the attributes may be selected for the purpose of quantifying them. These attributes are given in the appendix [H].

Further classification of the quality attributes are presented in the following section:

4.5.4.5 Further Classification Of The Quality Attributes

In this section all the 60 generated software quality attributes are classified into static, dynamic and both static and dynamic qualities. The reasons for such a classification are the following:

1. to ease the modelling of various attributes,

2. to divide and conquer,

3. to assess the dependency of these attributes,

4. to determine the difficulty in using these attributes for assessing the quality of the output of the system development activities.

Some more details about the quality attributes classification are given below:

b1. Static attributes

A static quality attribute is one which in no way depends on times or specific execution paths, e.g. readability. The following are some of static quality attributes:

Augmentability
Coercion
Conciseness
Coupleding
Expandability
Feasibility
Intraoperability
Legibility
Modifiability
Modularity
Readability
Redundancy
Self-descriptiveness
Simplicity
Structuredness
Unambiguity
Understandability
Uniformity

b2. Dynamic attributes

A dynamic quality attribute is one which becomes meaningful with a working system e.g. reliability. The following are some of dynamic quality attributes:

Accuracy

Compatibility

Correctiveness

Error-tolerance

Independence

Integrity

Interoperability

Operability

Portability

Reliability

Resiliency

Reusability

Robustness

Sensitivity

Survivability

Testability

Transportability

Trustworthy

Usability

Veracity

b3. Static and dynamic attributes

A static and dynamic quality attribute is one which can be applied to a working system as well as to system which is under development. Such attributes may behave differently for the same system. For example, the quality attribute efficiency (with respect to storage) will be static in case of a system which is written in the FORTRAN language where the storage has to be assigned previously before running the system. The same attribute will be dynamic in the case of a system which is written in ALGOL like languages because storage is allocated at run time. This behavior can cause a problem in assessing a certain attribute.

Accessability

Adaptability

Auditability

Cohesion

Communicativness

Completeness

Complexity

Comprehensiveness

Conformance

Connectivity

Consistency

Effectiveness

Efficiency

Elasticity

Flexibility

Generality

Maintainability

Measureability

Predictability

Security

Stability

Traceability

4.5.5 The Life Cycle Of The Quality Plan

The life cycle of the quality plan can be defined as: a set of distinguishable activities occuring in a certain order during the development of the quality plan. The time periods during which these activities occur are called phases. The life cycle approach to a quality plan may help to ease the achievement of qualities by separating different aspects of the quality plan into separate parts. This simplifies the means of achievement of quality. This life cycle makes quality plan development easy and systematic. This because the life cycle approach divides the quality plan into separate phases which can be easily managed. It enables the software developer to detect the basic errors earlier in the development task and decreases the cost of their elimination. This can be done by knowing the behaviour of each attribute during the different phases of the software system. The life cycle of the quality plan may comprise the following phases :

Note: the numbering of these phases is done according to the

scheme that:

a. the first figure in the heading followed by a star is the principle phase number,

b. the second figure in the heading followed by a star is the subphase/step number,

c. the verification and validation phase is of special case which is preceeded by the letter 'V'

1*. software quality requirements phase,

2*. software quality factors phase,

3*. software quality model phase,

V. validation and verification phase.

4.5.5.1 1* Quality Requirements Phase: -

The quality plan should arise as a result of imposing certain specific requirements from the user/management. The quality requirements can be defined as some thing which is deemed to be needed. These requirements may be imposed at the begining of the project or may be changed at a later phase. Such changes of the requirements occur because the demands of the user/management may not be constant during the system development phases. Before starting the development of a software system people concerned with the system have to agree which quality attributes are important and must be available in the software system. Such agreement is very essential, because different people may have different opinions about which quality attributes are important and

should be in a software system for a desired purpose. For instance, a project manager may be concerned with the attributes which make the quality of a product compliant with a standard. He may be concerned with the implementation of functional requirements within cost and schedule constraints. The end-users may look for those attributes which can make the software easy to use and give a quick response time. A maintainer may look for clear documentation and understandable code. The quality attributes as expressed in the requirements generally depend on the opinion of the people who are expressing the requirements. The quality requirements may vary significantly in various phases of the life cycle [35]. For example, quality requirements may be different, interrelated, or overlaped in the analysis phase and design phase. Early interpretation of the quality requirements will reduce the chances of misunderstanding at a later stage of the software system development, where the cost will be very high [25]. Therefore if the quality requirements are ill-defined during the earlier stages, the software may have unsatisfactory results. The quality requirements phase may consists of the following subphases:

1.1* quality requirements proposal,

1.2* study of proposal,

1.1*Quality Requirements Proposal

In this phase the user/management propose various quality requirements which they think should be in the software system. They may propose many things which they want.

They may propose a set of quality attributes which should be in the software system. They propose verbally, in writing, etc.. There may be more than one proposal. Before submitting such proposal(s) the user/management should consider at first what are the quality requirements which may be needed in the software system. The user/management may have their own list of quality attributes. These proposal(s) will be submitted to the software developer for technical considerations. For example, proposals may be to have the following:

i. ease of use of the software system,

ii. optimal resources utilisation by the software system,

iii. a list of quality attributes which should exist in the software system,

iv. etc..

Some of the above proposals may be essential and possible to achieve. For example, ease of use. This proposal can be solved by training, producing manuals or both. Some of the above proposals may be difficult to obtain and may increase the cost. For example, it may not beneficial to satisfy the user proposal(s) when he/she is asking for all existing quality attributes to be achieved in the required software system. The output of this subphase may be a set of quality requirements which may be proposed by the user and management jointly keeping each others interests.

1.2* Study of Proposal

Usually user and management face difficulties in expressing correctly and precisely their exact requirements. The developer must explain to the user/management what are the relevant and beneficial proposals. In this subphase, developer and technical personnel must study the proposal given by the user/management. The purpose of the study will be to determine their actual needs and to record possible benefits that can be achieved, taking into consideration how to satisfy these needs, satisfy standards, cost manpower constraints, etc.. In order to achieve all this the developer may use different available tools. The developer may consult the relationship matrix, list of synonyms and classification list of quality attributes which are developed in this study, or any other suitable tool. The output of this sub phase is an initial list of quality attributes studied by the developer and technical personnel. This initial list of quality attributes may follow some changes in the coming phases(if needed).

4.5.5.2 2* Quality Factors Phase: -

After reaching agreement on the quality attributes, a set of the quality attributes will be selected by technical personnel in consultation with the user/management. These selected attributes are the ones which may be characterised for the purpose of quantifying them. These are called quality factors. More reasons

for such selection may be the following:

1. to satisfy the technical and management requirements,
2. to satisfy the technical and management standard.
3. to satisfy certain constraints such as cost, manpower etc.

The selection of the quality factors may depend sometimes on the status of the software system. For example, if a software system is an experimental one, where software specifications will have a high rate of change, the flexibility factor will have more weight than any other factor. If network and communication capabilities are required or system to system interfaces are needed, then the interoperability factor becomes extremely important. If the life cycle of the software system is expected to be very long, then the maintainability factor will be important, etc.. Sometimes tailoring or limiting the achievement of the quality level becomes desirable. This tailoring may be done as a result of certain constraints, which are mentioned before, such as cost, time, manpower etc., purpose of the software. Such a modified quality plan will represent the set of selected attributes for which the quality plan will generally be acceptable. However, to achieve this it may necessary to reduce specific needs, reduce scope etc., of the quality plan.

It is essential to establish a set of criteria which can be compared with predefined standards to judge quality factors.

4.5.5.3 3* Quality Model Phase: -

A software quality model is of great concern to the developers and users of a software system. This is because most of the computer programs can be viewed as models, which are increasingly used in software systems [71]. The quality model can be defined as: an abstract representation of agreed required quality concepts, views, postulates, factors, etc., which can be a guide in developing a software system of desired quality. The software quality model may help the user, management or customer to evaluate and validate some of their requirements. The quality model, which is considered in this study, consists of the following steps:

3.1* the objectives of the quality model,

3.2* the scope of the quality model,

3.3* the development of the quality model,

3.4* the framework of the quality model.

3.1* Objective of the Quality Model

A software quality model may have the following objectives:

a. to guide the managers in the control of software quality,

b. to produce a consistent standard of software quality,

c. to optimise the available constraints such as: cost, time, manpower, etc., of a given project.

3.2* Scope of the Quality Model

The span of the quality model may be over all the phases of the software system development life cycle.

3.3* Development of the Quality Model

In this section a set of quality factors is selected for the purpose of quantifying them. The following possibilities should be considered while the software quality model is under development:

i. some quality factors may be intangible, e.g. readability,

ii. some quality factors may conflict with each others. For example, conciseness may conflict with readability.

iii. some quality factors are interdependent. For example, usability, and reliability. If the system is not usable then the reliability is meaningless, or vice versa,

iv. some quality factors may be synonyms. For example, clarity, readability, legibility, etc.,

v. some quality factors may improve the quality, others may increase the customer satisfaction, e.g. cheapness,

TEXT
BOUND INTO THE
SPINE

The reasons for choosing such quality factors may the following:

i.they are easy to characterise,

ii.they can be quantified from an internal viewpoint.

The considered phases of the life cycle, in this study, are; the analysis, design and implementation phase. However, a particular factor may correspond to more than one phase of the software system life cycle.

<u>b. Quantifying the Quality Factors</u>

For each phase of the system development life cycle there may be a number of distinct factors which could be quantified in some way. The criteria to decide which quality factors are relevant to any particular phase must depend very much on the reason why a measure of that phase should be required.

In the previous section, the quality factors are selected for the purpose of quantifying them. In this section an example is given in figure 4.5 which is an attempt to show the way of measuring different quality factors through out analysis, design and implementation phases.

Figure 4.5

Quality Factor	software system main phases		
	Analysis	Design	Implementation
Communi-cativity	metric is not defined	(a) the metric C1 is defined as the number of communication lines between the identified subsystems. (b) The metric C2 is defined as $C2 = \frac{e+1}{n}$ where; e is the no. of the communication lines in flow graph of the system design and n is the no. of nodes in flow graph.	the metric C3 is defined as the the number of interprogram unit communications which occur when the program is used.

Note: in figure 4.5 the metric C2 is applicable for the software system design which is based on structured decomposition. For example, Yourdon's structured design system [7]. This is because the design in this case can be represented as a control flow or data flow graph.

Figure 4.6

Quality Factor	software system main phases		
	Analysis	Design	Implementation
Legibility	metric is not defined.	metric is not defined.	the metric I is defined as the number of the identations in a program.
Portability	metric is not defined.	metric is not defined.	the metric P is defined as: P = |S2 - S1|, = number of lines of code changed plus the number of lines of code inserted in order to produce the new system. S1 is the system before making changes, S2 is the system after making changes.

Note PFORT verifier which developed by Ryder B. G. [100] can be considered as a good metric to measure the quality factor portability. This is because PFORT verifier provides a number of facilities in debugging and documentation. Also it produces intraprogram unit error diagnostic, symbol table and cross reference tables. These facilities help to find the number of changes to make a system portable.

Regarding the analysis phase it was not possible to quantify the above quality factors which are shown in figures 4.5 and 4.6. The reasons for that are the following:

a. apparently the analysis phase is too abstract,

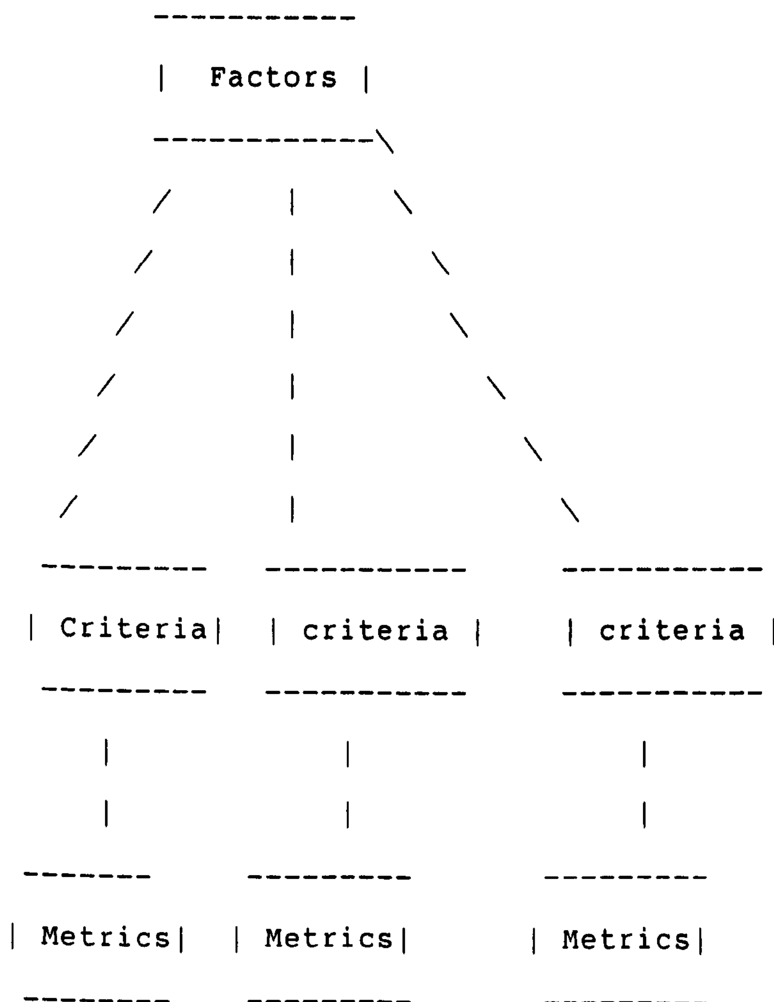
b. a lack of time to dig out the possibility of quantifying such factors.

3.4* Framework of the Quality Model

Boehm et al [29], McCall [46], Bowen [79] have developed a model of hierarchy structure relationship between factors, criteria, and metrics as shown in figure 4.7.

figure 4.7

A model of relationship between quality factors, quality criterian, and quality metrics.



The given Model is an easy one. But it is not showing any details about the quality factor, quality criteria and quality metric(s). Further Boehm, McCall, and Bowen have considered the "external" view of quality factors, criteria, and metrics. That is the view of the users/buyers of the software. They considered the quality factors of the final product, without considering how the overall quality can be affected by the output of other phases. For example, analysis phase, design phase, etc.. They are quiet about which quality attributes correspond to the analysis phase and which quality attributes correspond to the design phase. They selected a group of quality factors which are affecting the quality product. Whatever, grouping is used it is not clear from their model which is given in figure 4.7. However, the following may be considered as important disadvantages with their model:

1. some of the relationships between the quality factors are not well-defined. For example, efficiency and flexibility,

2. the relationship of criteria, metrics, and the software system development life cycle is not well defined. For example, the metrics must be identified according to various phases of the life cycle (analysis, design, implementastion).

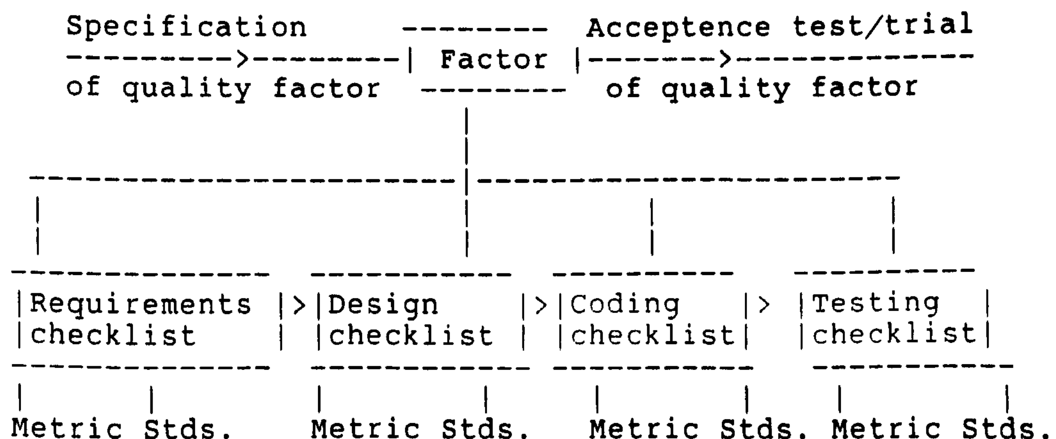
3. some of the criteria which are used as a necessary condition for the various factors seem to be inconsistent. For example, McCall uses the criteria training, communicativeness, and operability as associated criterion with the quality factor usability. However, it will not necessarly true that quality factor is associated with the given criteria. This is because the quality factor usability may be more strongly associated with some other criteria such as reliability, understandability, etc..

4. there is no rational distinguish whether the criteria refine the quality factor or identify the requirements that the software system must satisfy.

B.A. Kitchenham [33] has developed another hierarchy model for quality factors, criterion and metrics relationship which is shown in figure 4.8.

Figure 4.8

A hierarichal model of quality factors, criteria, metrics and their relationship.



Kitchenham [33] claimed that the above model shows that achievement of a quality factor depends on the entire developmenet cycle, and that at each stage certain checklists comprising standards and metrics should be used to monitor progress towards the quality factor. However, Kitchenham did not show how to do that. Further, the following points are not considered in the given model.

1. some quality factors cannot be compared directly. For example, efficiency can be defined according to the specified requirements. That is, it can be defined in terms of response time for an enquiry system, or in terms of average C.P.U. utilization for an operating system. The corresponding criterion to be satisfied at the internal phases is not obvious.

2. some systems may have no requirement at all for a particular quality factor. For example, portability would have no meaning to those systems which are not intended to be ported.

The following software quality framework is suggested in this study which may tackle the above mentioned problems.

figure 4.9

Framework of the Quality Model

01 agreed software quality attributes,

02 selected software quality factors,

03.1 software quality metrics,

03.2 metrics collection procedures,

04 software quality criteria.

At the level one of the hierarchy are the quality attributes which represent a superset of an adverbs which the developer, management, user, etc., think have impact over the software quality development task.

At the second level of the hierarchy is a set of selected quality attributes which are called a quality factors. These quality factors are selected by the developer with the consultation of user/management.

At the third level of the hierarchy are the quality metrics and the procedures which are necessary to collect them. At this level the quality factors are characterised so that they become quantifiable. This step is achieved by defining a set of metrics which can be compared with a predefined scale (standard) used to judge the quality factors. The predefined scale is called quality metric. It is possible that each quality factor has a set of quality metrics. However, a certain metric may correspond to more than one quality factor. These metrics are generally meaningful to technical personnel, such as analyst, designer, programmer etc..

At the fourth level of hierarchy are the software quality criteria. For software quality factors which are characterised in terms of software quality metrics these identify the metric values or range of values required in order for the software to be said to have achieved the required level of the related quality factor. When several metrics are related to one quality factor all the criteria must be met before the quality factor can be said to have reached its required level. Achieving the required level of a software quality attributes is, by definition, equated to achieving the required level of all the software quality factors related to that attribute.

The quality model framework can serve as a top-down hierarchy structure, which may be used to:

i. facilitate the establishment of quality concepts in terms of quality attributes by management/user/developer in earlier stages of the system life cycle,

ii. facilitate the establishment of quality goals in terms of quality factors by the developer in any phase of the life cycle,

iii. facilitate the communication of quality goals to the technical personnel in terms of quality metrics,

iv. facilitate the relationships between established goals and metrics through quality criteria,

v. facilitate the establishment of metrics to measure the extent or level to which a certain software system possesses certain attributes which affect the software quality.

On the other hand, the framework of figure 4.9 can be used as a bottom up hierarchy in the following ways:

1. evaluate the product of a software system at metrics levels,

2. combining the results of the evaluated criterion to assess the related quality factors.

3. combining the results of the evaluated factors to establish the related quality attributes.

The model structure which is shown in figure 4.9 is written in a COBOL programming notations to show a possible hierarchical relationship between the quality attributes, factors, criteria, metrics and the procedures to obtain the suggested metrics. This quality model may help to decide which techniques should be used in order to achieve the appropriate levels for the quality metrics. The selection of appropriate techniques and the achievement of the metric values up to an agreed level indicates that the required software quality attribute has been satisfied.

4.5.5.4 V* Verification And Validation Phase: -

The entire task of examining a software product to confirm that it operates as it is intended to do is often refered to as verification and validation. Generally, the people concerned do not differentiate between the two words. These terms in general are often used when discussing testing [31].

The word verification is some times associated uniquely with proof of correctness. Verification is defined by IEEE Glossary [13] as:

1. "The process of determining whether or not the products of a given phase of the software development cycle fulfil the requirements established during the previous phase",

2. "Formal proof of program correctness",

3. "The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services or documents conform to specified requirements" .

Verification is referred to in this study as an activity of comparing the output of quality plan phases with the output of ealier phase(s) of the quality plan. This is done for the purpose of establishment of the correctness of any two outputs. For example, quality factors phase may be verified with respect to the quality

requirements phase. This is can be done by checking whether the qualities are the needed ones or not.

The term validation is defined by IEEE Glossary [13] as: "The process of evaluating software at the end of the software development process to ensure compliance with software requirements". Validation is referred to in this study as the means to evaluate the software development product with the user/managements requirements for that product. It may also used to refer to the customer acceptance testing when the final software system is tested in the environment for which it was intended.

The purpose of the verification and validation phase in the quality plan life cycle is to:

1. guide the user/management and the technical personnel in making decisions about the next step, which may be to go a head or to stop the project,

2. satisfy the user/management that a certain phase in the quality plan is done according to the output of the previous phase,

3. ensure that the software product satisfies the customer requirements.

The verification and the validation phase may be activated at any time and in any phase of the quality plan life cycle. The verification and validation phase

can be achieved by testing, inspecting, reviewing, auditing and implementation.

Any disagreement after the verification and validation phase may be due to the following:

1. user/management have changed their minds about their requirements,

2. the selection of the quality factors was irrelevant,

3. the requirements were not understood correctly and precisely.

Hence, due to any of the above reasons or the combination of any of them, it will be necessary to go back and check the previous phases and do the whole exercise again. The re-usability of the verification and validation phase in the quality plan life cycle can improve the quality of the output of a particular phase in which it is being used and as a direct result the overall quality of the software system.

It is not enough to specify that software system has the qualities which are required by the user(s). It has to be demonstrated that the required qualities are present in the system. In order to demonstrate that the required qualities have being achieved, the following steps must be considered:

1. setting standards for:
    a. determining cost quality,
    b. determining performance quality,
    c. determining safety quality,
    d. determining reliability quality.

2. evaluation conformance; that is, comparing the conformance of the software product to the set standards,

3. comprehensive correction when necessary; that is, correcting problems and their causes throughout the whole life cycle, which influence the user satisfaction,

4. planning and improvements; that is, developing a continuing effort to improve the quality of the software system.

The output of the verification and validation phase will be something which is compared against some standards. For example, the initial quality attributes list which is the output of the requirements phase may be compared with the quality factors phase. This is may be required to verify that the selected quality factors are the needed ones. This output will also ensure that the product of the software system is complying with predefined standards. For example, end product will be validated against the user/management requirements. The output of the verification and validation phase may be

useful for further tasks.

4.5.6 The Diagram Of The Quality Plan Life Cycle

In this section a diagram is given as a guideline for the use of the quality plan life cycle. As shown in figure 10 (a), the quality plan contains four phases. Within each of some phases, a series of subphases or steps are exist. These subphases or steps participate in accomplishing the objectives of the particular phase. The approach is developed to show the input and output of each phase of the quality plan life cycle. The phases and steps depicted in figure 10(a) are each described in another figure which is shown in figure 10 (b), (c), (d), (e) and (f). These figure are presented to provide an outline of the phase, subphase and steps of the quality plan life cycle. The life cycle of the quality plan as presented in section 4.5.5 and depicted in section 4.5.6 is intended to be a guideline, not a rigid set of instructions. It can be use as a reference guide to assist project participants in producing quality results for their assigned tasks.

Figure 4.10 (a).

The Life Cycle of the Quality Plan.

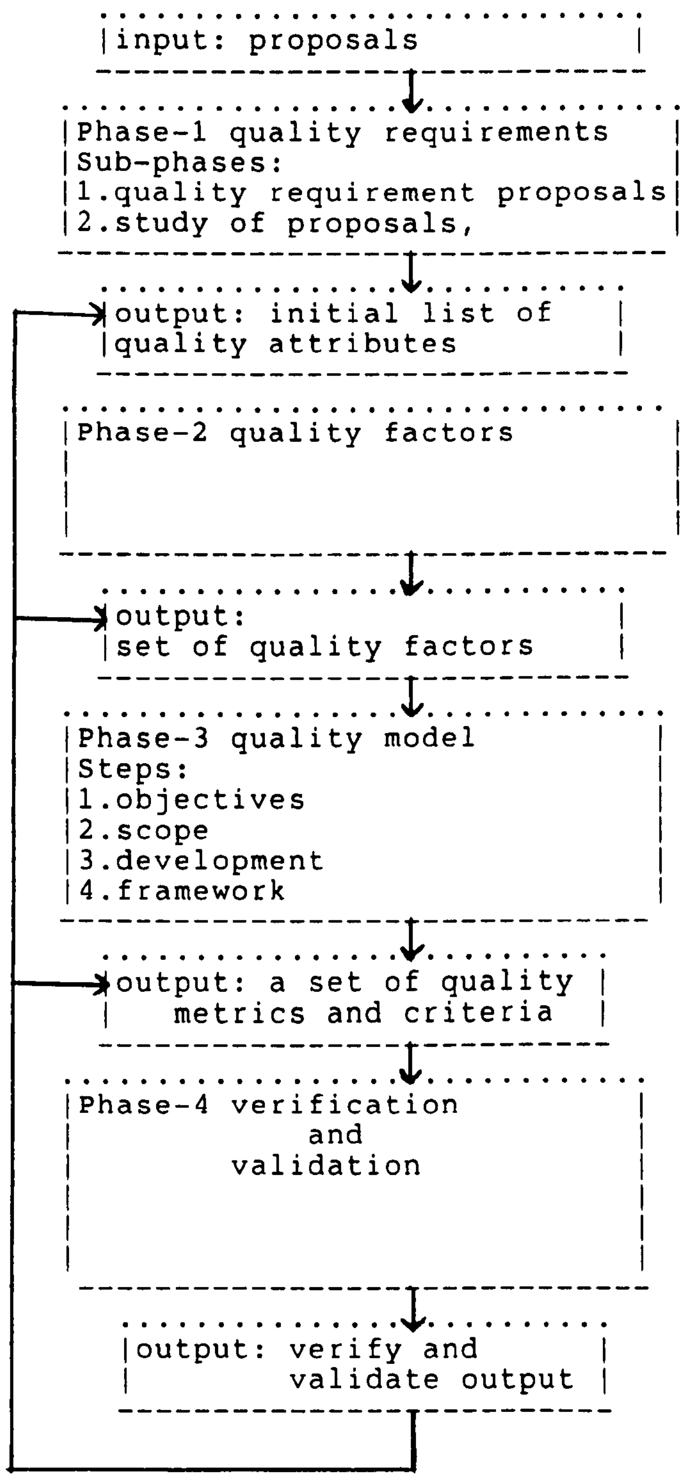


Figure 4.10 (b).

The Life Cycle of the Quality Plan.

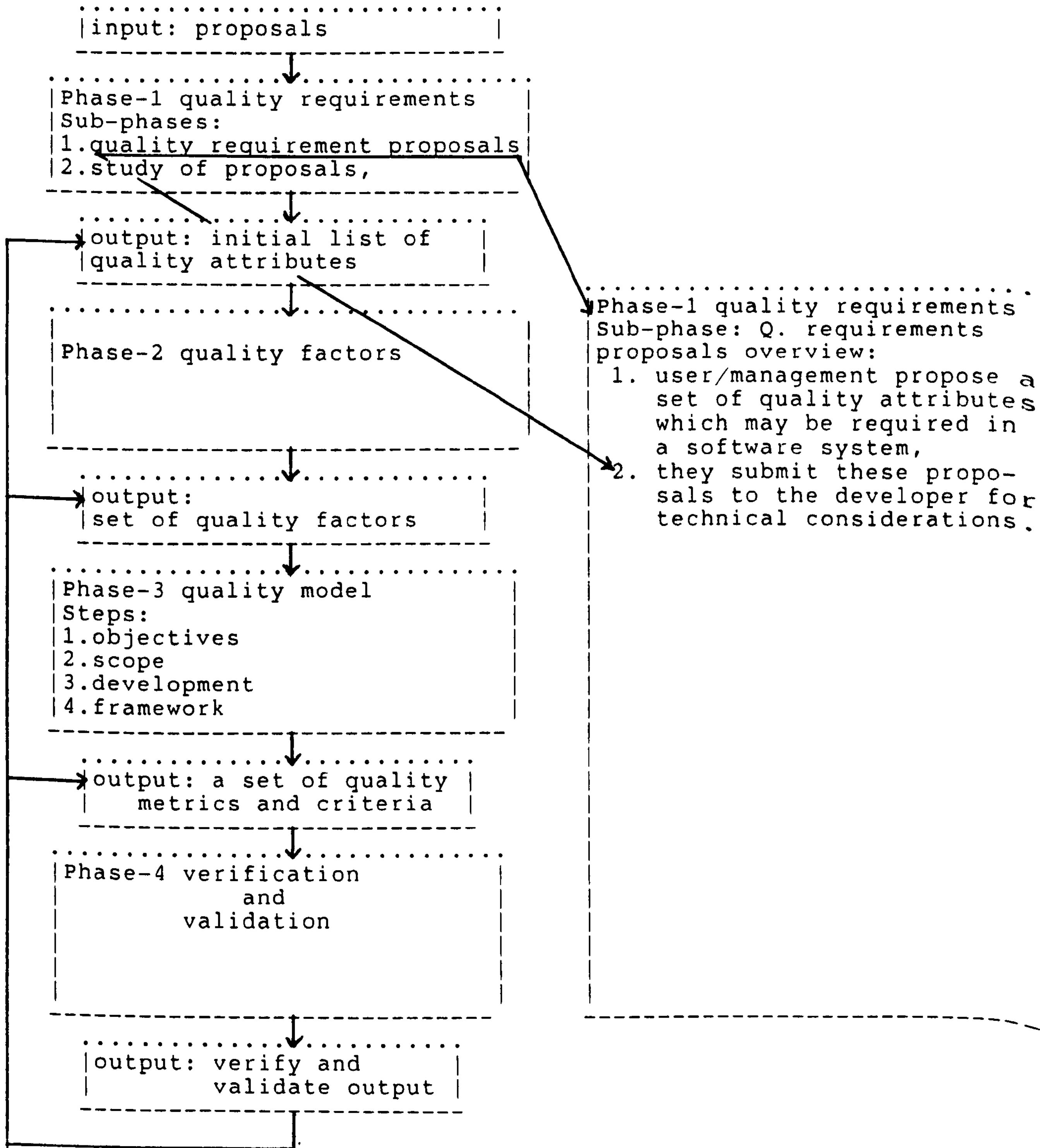


Figure 4.10 (c).

The Life Cycle of the Quality Plan.

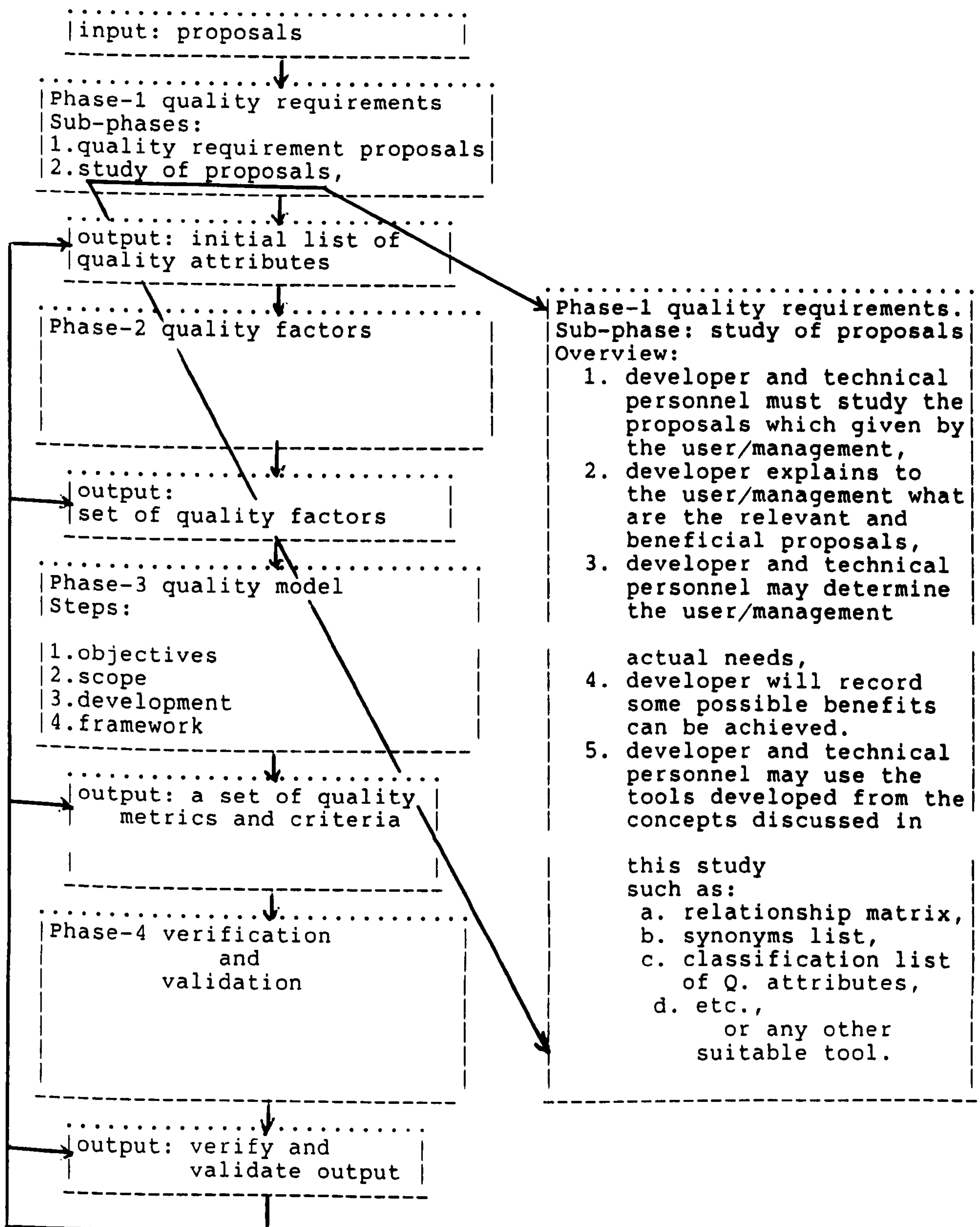


Figure 4.10 (d).

The Life Cycle of the Quality Plan.

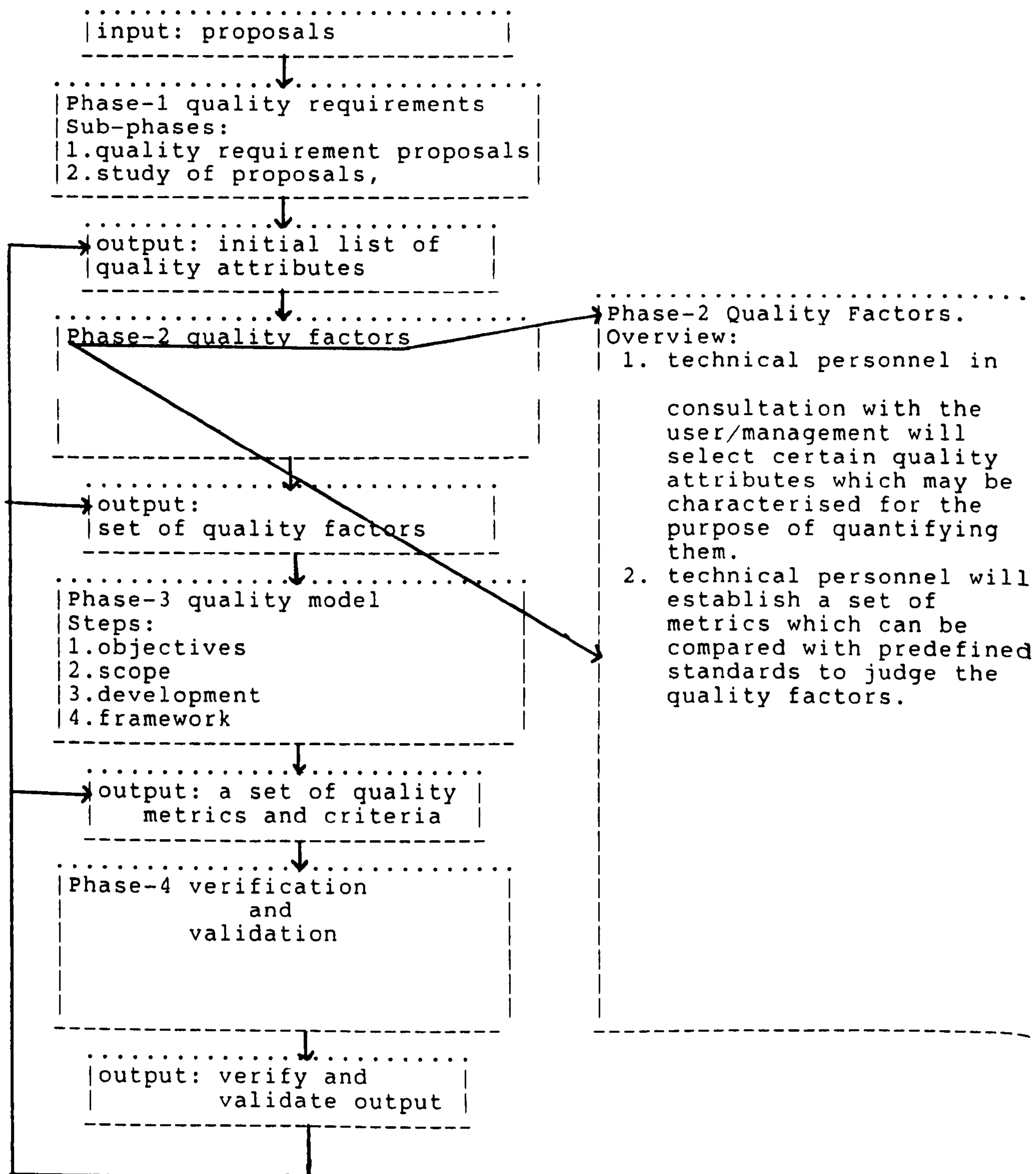


Figure 4.10 (e).

The Life Cycle of the Quality Plan.

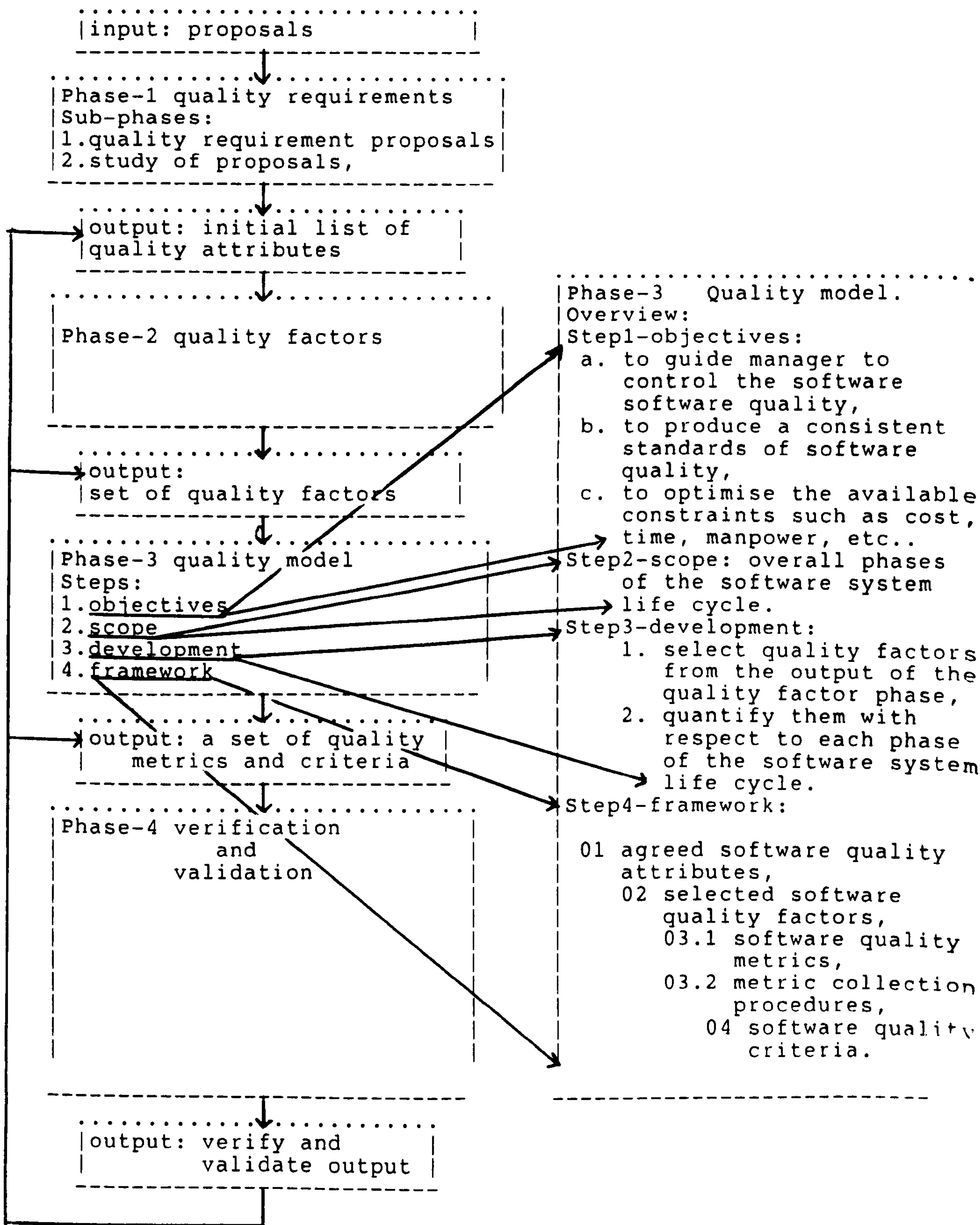
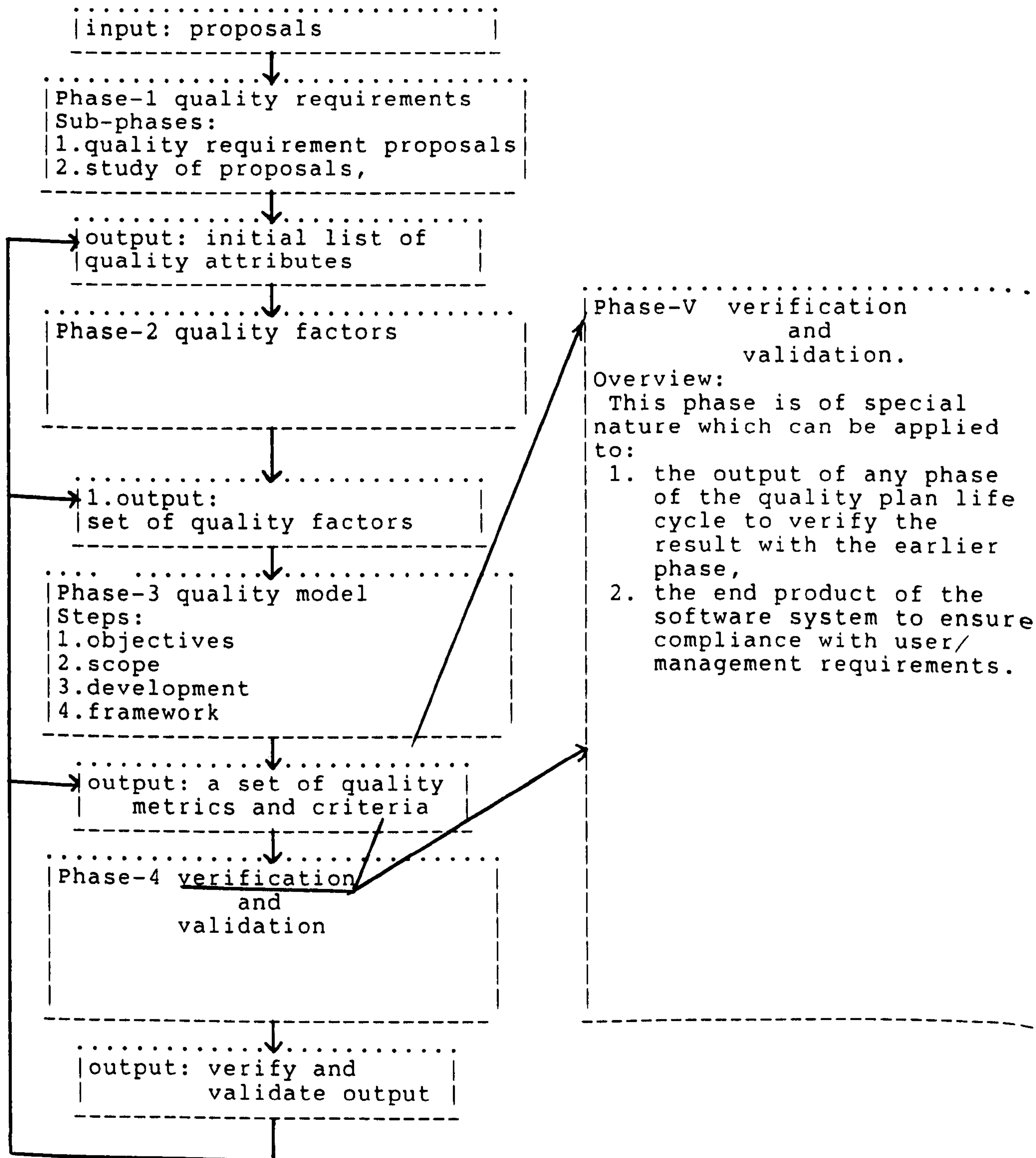


Figure 4.10 (f).

The Life Cycle of the Quality Plan.



4.6 CONCLUSIONS:

A software quality, may become more visible when the following exist;

1. a clear definition of what quality means,

2. a widely agreed attributes of the quality,

3. a detailed quality plan which comprises:

    1. the selection of the important quality factors and metrics which are most related to the environment where the software is to be implemented,

    2. the selection of the methods, techniques, and approaches which will be employed to accomplish, develop, verify and operate the software system.

    3. a verification and validation phase in which the output of each phase of the quality plan can be verified and the end product can be validated a gainst pedefined standard.

    4. software quality metrics which may be employed to:

    a. set up acceptance criteria and standards,
    b. evaluate the level of the quality being achieved against the established requirements,
    c. compare the quality attributes of one system with those of another system (if there is any),

d. predict the level of quality which may be achieved in the future.

A software quality plan may be considered as an instantiation of these points which can be used by the software system implementers to identify specific distinguishing quality attributes which should be built into the software in order to meet the quality goals.

CHAPTER FIVE

CHAPTER 5

SUMMARY AND CONCLUSION

5.1 INTRODUCTION

The first part of this thesis is an attempt to evaluate and compare the behaviour and performance of various software quality metrics. The evaluation of software metrics is very essential for developers, users, management and software engineers, etc., to help them in selecting suitable metric(s), to measure and estimate software qualities. For this purpose, three different categories of software metrics were classified into:

1. primitive software science metrics, which are based on counts of lexical tokens in a program or program interface features.

2. abstract software metrics, which are based on graph theory.

3. structured software metrics, which are based on the software system components connections.

These categories of software metrics are discussed thoroughly in chapter 2. Then an analytical comparative

study was performed to selected metrics from the above classified categories. The criteria of the selection of such metrics were the following:

a. the popularity of the metric and its frequent use,

b. the availability of literature about the metric.

To facilitate such a comparison a set of criteria of goodness was developed against which the selected software metrics were examined and evaluated.

The second part of this thesis is an attempt to:

1. establish clear definitions of the software quality and their related terms. This was achieved in this study by the development of clear definitions for the terms software quality, quality characteristic, software quality property, software quality attribute, software quality factor, software quality criteria, software quality metric and software quality plan. These definitions are given in chapter 4 of this thesis.

2. to identify and define various quality attributes which contribute to software quality. This was achieved by identifying and defining certain software quality attributes. During this effort 60 possible quality attributes were defined. These definitions of the quality attributes are given in the appendix [B]. Considerations were made to include all the attributes which describe improvement to or achievement of the quality of a software system to

any extent and at any phase of the system development life cycle. It was observed that out of these 60 attributes some of them are synonyms. The list of these synonyms of the software quality attributes are generated in this study. These synonyms are in terms of their general meanings as defined in [82]. The list of synonyms is given in the appendix [D]. In addittion to providing definitions of the quality attributes and list of synonyms, it was also decided to investigate the extent to which experienced software engineers were familiar with, and consistent in their use of, software quality attributes. This was done by asking coworkers in Liverpool University and LDRA to indicate what they understand to be the nature of the relationship between each of the different software quality attributes. The coworkers who took part in the experiment were each given a relationship matrix, and asked to identify the nature of the relationships among the software quality attributes as shown in appendix [E].

A composite response was constructed from each individual response by including all the relationships which were agreed unanimously. The composite response is shown in appendix [E].

3. establish an internal view of software quality. In this study seven different internal viewpoints of software quality were identified. These viewpoints are given in chapter 4 of this thesis.

4. develop a detailed quality plan which provides a clear strategy to select the desired quality attributes and provide a frame work to examine the possible affect or behaviour of a certain quality attribute. This quality plan is developed and disscussed in detail in chapter 4. The quality plan comprise the following phases:
    a. software quality requirements phase,
    b. software quality factors phase,
    c. software quality model phase,
    d. validation and verification phase.

5.2 CONCLUSION

As a result of the first part of this study the following conclusions are drawn:

i. during the study of software metrics, it was observed that, up to now, efforts in developing software quality metrics have been concentrated on very few quality attributes such as complexity, stability, etc.. On other hand, for certain important quality attributes such as usability, readability, etc., real metrics are still not available.

ii. the evaluation of the primitive software metrics against the developed criteria of goodness showed that the existing metrics have the advantage that they are applicable at least to the design and implementation(code) phases of the software system

life cycle. However, one of the disavantages of these metrics is that, they are theoretically and practically in a weak state. Therefore, these metrics must be treated with great caution.

iii. the evaluation of the abstract software metrics against the suggested criteria of goodness shows that these metrics can adequately quantify control flow complexity with some exceptions. Some of these exceptions may be the failure of McCabe's metric to cope with the linearization problem.

iv. the evaluation of the structured software metrics against the suggested criteria of goodness showed that these metrics provide certain advantages for a manager's overall understanding of system complexity and its impact on system costs and performance. For instance, the metrics measuring interconnectedness among segments of a software system will enable the manager to predict the maintenance cost of the software system. This is because such a metric deals with a macro-level of the system which may be easy to understand.

v. generally no metric at present can be considered as a standard which is universally perfect. This is because these metrics contain some inherent defects. Also, user/management change their requirements, continuously demanding better quality. Metrics which are available now are not sensitive to errors.

vi. the metrics which are developed so far with few exceptions are not evaluated or examined thoroughly.

vii. the classification of the software metrics which is developed in this study would help to determine which phase of the software system life cycle these metrics can measure.

As a result of the second part of this study the following conclusions are drawn:

1. some desired attributes of software quality can only be satisfied at the expense of other attributes. For instance, reliability may be influenced by cost, size complexity, etc..

2. the quality of software system is environment dependent, thus, it is inadequate to establish a single figure for software quality.

3. there are no common and agreed terms of Software Engineering which are free from misinterpretation. Therefore, an attempt is made in this study to define the software quality and its related terms and use them consistently.

4. software quality becomes more visible when the following exist:

i. clear definitions of software quality,

ii. a widely agreed set of defined quality attributes,

iii. a detailed quality plan which at least comprises:

* a strategy for selection of the important quality factors and metrics which are most related to the environment where the software is to be implemented,
* the selection of the methods, techniques, and approaches which will be employed to accomplish, develop, verify and operate the software system,
* a verification and validation phase in which the values of the software metrics are established. These metrics may be employed to:
  a. set up acceptance criteria and standards,
  b. evaluate the level of the quality being achieved against the established requirements,
  c. compare the quality attributes of one system with those of another system (if there is any),
  d. determine the desirable quality attributes of the software system.
  e. predict the level of quality which may be achieved in the future.

5.3 SUGGESTIONS AND FUTURE WORK

Regarding the software metrics the following recommendations are suggessted for future work, to be

considered:

1. to determine how software metrics can help to find and remove errors in software,

2. to determine how software metrics can help to find the origin of the errors.

3. while developing a software metric the following points should be considered:

   i. it should be of global nature, so that it can be used in different phases of the life cycle by changing the essential variables or factors.

   ii. it should be sensitive to the changes in the structure of the system,

   iii. it should be sensitive to the errors in the software systems,

   iv. it should include some other important attributes such as modifiability, reliability, usability, etc.,

   v. it should be validated on practical software system,

   vi. it should be rigorous i.e., have a mathematical basis,

Regarding the quality issues the following suggestions are recommended for future work:

1. relate the external views of software quality with the internal views of the software quality,

2. establish a new department called quality circle which is a group of employee, usually from the same area of an organisation.

This group should meet periodically to:

a. examine, analyse and solve the problems of software system quality,

b. to enhance the communication between employee and management. This point will have an advantage that the whole organisation will participate in the solution of quality problems.

APPENDIX A

adequate confidence to assure that the quality in the software product and confirm that the product is satisfying the technical and user/customer requirements.

A.3 SOFTWARE QUALITY ATTRIBUTE

(see chapter 4, PP. 4-16).

A.4 SOFTWARE QUALITY CHARACTERISTIC

(see chapter 4, PP. 4-15).

A.5 SOFTWARE QUALITY CRITERIA

(see chapter 4, PP. 4-18).

A.6 SOFTWARE QUALITY FACTOR

(see chapter 4, PP. 4-18).

A.7 SOFTWARE QUALITY METRIC

(see chapter 4, PP. 4-19).

A.8 SOFTWARE QUALITY PLAN

(see chapter 4, PP. 4-21).

A.9 SOFTWARE QUALITY PROPERTY

(see chapter 4, PP. 4-16).

A.10 SOFTWARE QUALITIES

(see chapter 4, PP. 4-17).

A.11 SOFTWARE QUALITY

(see chapter 4, PP. 4-12).

A.12 SOFTWARE SYSTEM

The term software system is defined in this study as:

Any combination of documentation, software and the user/management to fullfil the specific function(s) or purpose(s) in an environment.

A.13 USER

The term user is defined in this study as:

1. As a person who will be using the software system such as an operator, software programmer, quality controller, stock control officer, accountant, etc..

2. As a person who will be using the software system in a supervisory capacity such as a person who is in charge of section in bank, quality control supervisor, etc..

3. As a person or organisation who will be using the software system as the owner, for example, a government ministry, a banking organisation, etc..

APPENDIX B

APPENDIX B

DEFINITIONS OF VARIOUS QUALITY ATTRIBUTES

The following are the definitions of various important software quality attributes. The source of each definition is quoted at the end of the definition. Otherwise these definitions are infered from the references [83} and [106].

B.1 ACCESSABILITY:

The degree of ease with which software components, i.e., modules, subroutines,..etc. can be reached or used. For example the best software system may be the one which is based on a tree structure, where its nodes can be visited easily.

B.2 ACCURACY:

The ability of a software system to produce results within a pr-defined range. Accuracy may be mathematical, e.g. a certain function can calculate a result to an accuracy of five decimal places. Accuracy may be logical, e.g. order of the statements.

B.3 ADAPTABILITY:

The degree to which a software system is capable of being changed for the predefined environments, purposes, functions,..etc..

B.4 AUDITABILITY:

The ease of giving an approporiate diagnoses and showing the way of correcting and monitoring the shortcomings of an established software system. It is necessary in auditing to take samples from time to time from software product for the purpose of study and checking. The auditability may involve the activities like; checking, investigation and inspection [69].

B.5 AUGMENTABILITY:

The extent to which a software system can accommodate expansion in its components.

B.6 COERCION:

The effort made to combine the components which are not close together in a software system. This may happen when a software system is highly complex. [34].

B.7 COHESION:

The extent to which the elements of a single module are functionally related [20]. For example, do the

elements combine together according to some given criteria. Cohesiveness can be measured by using a cohesion metric. For example T.J. Emerson's metric [21] can be used as a measure to show whether the elements of a module are functionally related, sequentially related, or logically related etc... Cohesiveness is a contrast with coupledness.

B.8 COMMUNICATIVNESS:

The level to which a software system is able to convey and transmit information, ideas, concepts, etc. from the given input to the required output. The extent to which a software system provides a useful and friendly interfaces with the reader. The following may cause a poor communication:

1. inadequate coverage of information (not all reliable data are being transmitted),
2. not all the data received by the responsible person,
3. the data are not generated properly at the source,
4. misinterpretation of data,
5. any single or combination of the above.

B.9 COMPATIBILITY:

The extent to which sofware systems are capable of existing together in harmony. For example, the data can

be interchanged, the code can be interchanged,...etc.

B.10 COMPLETENESS:

The extent to which a software product has all its necessary aspects. For example the software product is complete with respect to a given criteria. The software product is complete if all its components are present and each component is fully developed.

B.11 COMPLEXITY:

The degree of intricacy of a software system or its components. The complexity of software system can be measured as:

1. the degree of nesting of its componenets,
2. the total number of fan-in and fan-out of its components,
3. the extent to which certain components overlap.

B.12 COMPREHENSIVENESS:

The extent to which all the needed information is available in a software system [34].

B.13 CONCISENESS:

The extent to which a software product is free from all unnecessary details.

B.14 CONFORMANCE:

The extent to which a software product satisfies the pre-defined standard (if there is any).

B.15 CONNECTIVITY:

The degree of ease with which a software components are linked together [34].

B.16 CONSISTENCY:

The extent to which a software system contains uniform parts. For example, the comments in a program should not be unnecessary, extensive at one place and insufficiently informative at another.

B.17 CORRETNESS:

The extent to which software product is free from errors [13].

B.18 COUPLING:

The degree of interdependence among modules in a software system. It contrasts with cohesiveness.

B.19 EFFECTIVENESS:

The extent to which a software system performs its task successfully and efficiently.

B.20 EFFICIENCY:

The extent to which a system performs its required functions with minimum consumption of resources, such as memory space and execution time. The efficiency can be measured for example in terms of response time for enquiries for seat reservation system, or in terms of average CPU utilisation.

B.21 ELASTICITY:

The extent to which a software system is capable of being extendable without losing its former functions and becoming applicable to more general problems by permitting additional features to be added.

B.22 ERROR-TOLERANCE:

The extent to which a system continues to operate correctly despite internal or external errors.

B.23 EXPANDABILITY:

The extent to which a software system is capable of accommodating expansion, to provide additional functions or data storage, or to increase computational capacity. It can be measured as; the complexity of the system before and after the expansion for the comparison. Expandability is a synonym of modifiability and extendability.

B.24 FEASIBILITY:

The degree to which the software system is capable of being carried out successfully for certain environments, purposes, etc.. For example, developing the software system within a specific time, with specific resources, subject to specific pre-defined conditions and by the personnel available.

B.25 FLEXIBILITY:

The extent to which a software system is capable of changing, or expanding in response to new requirements such as change in modes of operations, in operating environment, interfaces with other software system and improvement.

B.26 GENERALITY:

The extent to which software system is applicable to total attributes, such as; flexibility, expandability, portability, and solving a biger range of problems than the current one.

B.27 INDEPENDENCE:

The extent to which a software system can be executed on computer hardware configurations other than its current one. This attribute is a necessary condition for portability [30].

B.28 INTEGRITY:

The degree to which a software system is secure against unwanted access.

B.29 INTEROPERABILITY:

The extent to which a software system is able to exchange information with other systems.

B.30 INTRAOPERABILITY:

The extent to which a software system has linkage between its internal components.

B.31 LEGIBILITY:

The extent to which software system is capable of being read. The clarity of the objects and their representation.

B.32 MAINTAINABILITY:

The extent to which a software system facilitates location and correction of errors which have not been discovered in the earlier stage of the software development. Maintainability can be measured as a period of time needed to repair a failure starting from the time of the occurrence of the failure.

B.33 MEASURABLITY:

The extent to which a software system is capable of being assessed. For example, a system can be easily measurable if every component in the software system is distenct and clear.

B.34 MODIFIABILITY:

The extent to which a software system is capable of accommodating changes.

B.35 MODULARITY:

The extent to which software system can be decomposed into smaller units provided that a change in one unit has minimal impact on other unit.

B.36 OPERABILITY:

The extent to which a software system is capable of functioning. For example, loading, initiating, executing and transmitting.

B.37 PORTABILITY:

The extent to which a software system is able to operate in more than one hardware or software environment. Portability can be assessed by the effort needed to install the system in a new environment.

B.38 PREDICTABILITY:

The extent to which a software system is giving the same results for the same inputs.

B.39 READABILITY:

The degree to which a software system can be read and understood easily. For example, program readability can be increased as a result of reformating task such as:

1. making every statement in the program start on a new line,
2. puting every GOTO statement and its target on a separate line,
3. writing all the necessary comments in a clear way,
4. etc..

B.40 REDUNDANCY:

The amount of duplicated features in a software system. Generally it is used to increase certain features such as structuredness of a system. Redundancy is often provided to increase security.

B.41 RELIABILITY:

The extent to which a software system can perform consistantly its required purposes or functions under the

stated conditions for a specific period of time without errors.

B.42 RESILIENCY:

The extent to which a software system is able to adjust easily and to recover from specific events [34].

B.43 REUSABILITY:

The degree to which software system entities such as data structures, procedures, modules, subsystems, and programs can be re-used. For example, re-using the software system in some other environment. Re-usability can be measured by considering the number of changes which are needed in a program, module, etc.. According to [33] re-usability of a component may be measured as:

rc = (effort to produce components -
effort to incorporate in a new system) *
(expected number of new system using component).

The re-usability of a system may be measured as:

R = (summation (rc)/total effort to produce system.
over all
reusable
components

B.44 ROBUSTNESS:

The extent to which a software system is able to continue to function correctly despite some violation of

the assumptions in its specification. For example, the ability of a software program to properly handle inputs out of range, or in a different format,..etc. without degrading its performance or functions.

B.45 SECURITY:

The degree to which a software system is protected against unauthorized users. For example, protection of code from additions, or any corruption.

B.46 SELF-DESCRIPTIVE:

The extent to which a software system contains adequate explanatory comments on its functions. For example, a software system containing enough information for its user(s) to determine its objectives, assumptions, constraints, inputs, outputs,...etc can be called self-descriptive.

B.47 SENSITIVITY:

The extent to which a software system outputs are stable with respect to internal software errors [34].

B.48 SIMPLICITY:

The extent to which a software system is readable, understandable and clear.

B.49 STABILITY:

The extent to which a software system is capable of resisting the changes that are made to its components [15]. For example, a system of three components; input, mapping function, and reporting mechanism will be a stable system if any change, in the input component, does not create any ripple effects in any other components.

B.50 STRUCTUREDNESS:

The extent to which a software system is built using easily identifiable and understandable items or techniques such as top down, bottom up, etc.

B.51 SURVIVABILITY:

The extent to which a software system continue to perform despite a portion of the system has failed [79]. The survivability can be measured as the number of survivability-related errors occuring during a certain time and the total number of executable lines of source code. For example, if 3 errors per 1000 lines of executable code is occurred during a specified time, then survivability measure level = (1-3/1000 = .997). The survivability may considered as a feature of software systems which are likely to be used in uncontrolled conditions [70].

B.52 TESTABILITY:

The degree to which defects in a software system can be detected under some given criteria. The extent to which a software system is capable of being tested. Testability considerations should contribute in developing the check list for requirements, design, coding and testing [33].

B.53 TRACEABILITY:

The degree to which a software system can be followed step by step to discover the flow of control, flow of data, flow of information, ... etc.. For example, a structured program is easily traceable. The extent to which the origin of information can be found.

B.54 TRANSPORTABILITY:

The extent to which a software system is capable of transfering from one environment to another. For example, it may be needed to transfer a particular software system into different environments or different software systems into the same environments.

B.55 TRUSTWORTHY:

The extent to which a software system is capable of performming its tasks according to the expectations without having unwanted side-effects [34].

B.56 UNAMBIGUITY:

The extent to which a software system is free of miss-interpretation.

B.57 UNDERSTANDABILITY:

The extent to which a software system is clear in purpose and operation to the person who is inspecting it. For example, whether variable names or symbols are used consistently throughout the system, the modules are self-descriptive, etc.. Understandability can be an outcome of simplicity, self-descriptiveness and modularity. Understandability may also be viewed from the viewpoint of a user as the level to which a software product is clear to him.

B.58 UNIFORMITY:

The degree to which a software system is consistent with respect to the style, notations, documentation, etc..

B.59 USABILITY:

The extent to which a software system needs an effort to understand its inputs and interpret its outputs. Usability can be determined by assessing how efficiently and quickly a user of moderate experience and skill, in a particular environment can understand and use the system successfully. The usability is a prerequisite

for the software re-usability[32].

B.60 VERACITY:

The degree or level to which the output of a software system can be made correct, accurate, reliable, and trustworthy.

APPENDIX C

APPENDIX C

TABLES

Table 2.1 Software Science Parameter for 20 Programmes

Algorithm Number	Observed Parameters n2'	n1	n2	N2	N	Level L	L'
CACM 1	8	10	18	56	104	0.066	0.064
CACM 2	3	14	8	37	84	0.031	0.031
CACM 3	12	18	41	220	454	0.020	0.021
CACM 4	5	16	21	61	137	0.028	0.043
CACM 5	4	15	16	60	124	0.025	0.036
CACM 6	4	15	13	42	99	0.033	0.041
CACM 7	3	10	9	29	59	0.046	0.062
CACM 8	5	15	16	60	133	0.030	0.036
CACM 9	10	19	41	162	312	0.023	0.027
CACM 10	3	9	9	25	48	0.058	0.080
CACM 11	3	9	9	29	55	0.051	0.069
CACM 12	3	11	9	31	62	0.043	0.053
CACM 13	3	11	8	30	61	0.045	0.048
CACM 14	7	15	25	91	187	0.029	0.037
GM 15	32	27	100	301	686	0.036	0.025
GM 28	43	49	214	944	1919	0.016	0.009
GM 36	68	47	329	1318	2642	0.019	0.011
GM 40	58	82	433	1944	3985	0.010	0.005
GM 50	32	35	168	584	1248	0.018	0.016
GM 118	6	13	24	57	122	0.038	0.065

Sources (Halstead [2]), Table 5.1

$V' = (2 + n2')\ Log2\ (2 + n2')$

$V = N*Log2(n1 + n2)$

$L = V'/V$

$L' = (2*n2)/(n1*N2)$

Table 2.2, from Halstead [2] , Table 6.6 intelligence content I of Algorithm CACM13 in various languages;

Languages	I
ALGOL 58	14
FORTRAN	15
COBOL	16
BASIC	15
SNOBOL	16
APL	16
PLI/I	16

Table 2.3, from Chritensen et al [93],information content for programs for Euclids Algorithm for finding the greatest common divisor.

Languages	I
PL/I	12.9
FORTRAN	10.5
CDC Assembler	12.2
ALGOL 68	11.9
TABLE LOOK UP	12.0
POTENTIAL HLL	11.6
BASIC	10.5
A P L	10.0

APPENDIX D

APPENDIX D

THE SYNONYMS OF SOFTWARE QUALITY ATTRIBUTES

D.1 INTRODUCTION

The following are the synonyms of the generated software quality attributes. These synonyms are in terms of their general meanings as defined in Roget's Thesaurus [82] and [83].

<Accessability> := <simplicity><traceability><flexibility>

<Accuracy> := <veracity>

<Adaptability> := <modifiability><conformance><generality><flexibility>

<Auditability> := <testability>

<Augmentability> := <expandability><strength>

<Coercion> := <inflexibility>

<Cohesiveness> := <uniformity>

<Communicativness> := <transmission><accessability>

<Compatibility> := <uniformity><conformance><cohesiveness>

<Completeness> := <comprehensiveness><generality>

<Complexity> := <nonuniformity>>complication>

<Comprehensiveness>:= <completeness><generality>

<Conciseness> := <overconciseness><condense><brief>

<Conformance> := <adaptability><generality><flexibility>

<Connectivity>	:=	<interaoperability><coupledness>
<Consistency>	:=	<uniformity><stability>
<Correctness>	:=	<accuracy>
<Coupledness>	:=	<connectivity><cohesiveness>
<Effectiveness>	:=	<influence>
<Efficiency>	:=	<stability><influence>
<Elasticity>	:=	<resilience><extensibility>
<Error-tolerance>	:=	<completeness>
<Expandability>	:=	<elasticity>
<Feasibility>	:=	<accessability>
<Flexibility>	:=	<elasticity>
<Generality>	:=	<comprehensiveness>
<Independence>	:=	<self-descriptive><unrelatedness> <noncoformity>
<Integrity>	:=	<reliability><trustworthy><veracity>
<Interoperability>	:=	<portability>
<Intraoperability>	:=	<coupledness>
<Legibility>	:=	<comprehensiveness>readability><clarity> <cohesivness><unambiguity><simplicity> <understandability>
<Maintainability>	:=	<auxiliary><improvement><modifiability>
<Measurablity>	:=	<performance><degree><strategy>
<Modifiability>	:=	<changeability><adaptability>
<Modularity>	:=	<mutation><modifiability>
<Operability>	:=	<dynamic><serviceability
<Portability>	:=	<transportability>
<Predictability>	:=	<stability>
<Readability>	:=	<legibility><clarity><unambiguity> <simplicity><comprehensiveness>

<Redundancy> := <plenty>>survivability>

<Reliability> := <stability><unchangeability>

<Resiliency> := <elasticity>

<Reusability> := <serviceability><adaptability>

<Robustness> := <elasticity><resiliency>

<Security> := <safety><clarity>

<Self-descriptive> := <clarity><unamiguity><independence>
<understandability>

<Sensitivity> := <changeability>

<Simplicity> := <uniformity>

<Stability> := <consistency><uniformity>

<Structuredness> := <composition>

<Survivability> := <redundency>

<Testability> := <trustworthy><complexity>

<Traceability> := <accessability><attributable>

<Transportability> := <portability>

<Trustworthy> := <reliability>

<Unambiguity> := <legibility><clarity><unambiguity>
<simplicity><comprehensiveness>

<Understandability>:= <legibility><clarity><unambiguity>
<simplicity><comprehensiveness>

<Uniformity> := <stability>

<Usability> := <adaptability><serviceability>

<Veracity> := <accuracy>

APPENDIX E

APPENDIX E

RELATIONSHIP MATRIX

A relationship matrix is given in this appendix [E] shows the relationships in various ways such as independency, interdepency, direct dependency. and indirect dependency. An idependency relationship is defined as occurring if two software quality attributes are mutually independent, e.g. readability might be judged to be independent of accuracy and vice versa. An interdependency relationship is defined as occuring when two quality attributes are mutually dependent on each other, e.g usability and reliability. A direct dependency relationship is defined as occurring when the availabilty of one software quality attribute is necessary for anathor, for example, accuracy is necessary for correctness but the reverse is not true. An indirect dependency relationship is defined as occurring if a software quality attribute occurs as a by-product of other software quality attributes, e.g. reliability may be viewed as a side-effect of correctness, testability and understandability.

ATTRIBUTES RELATIONSHIP MATRIX

	1 ACCESSABILITY	2 ACCURACY	3 ADAPTABILITY	4 AUDITABILITY	5 AUGMENTABILITY	6 COERCION	7 COHESIVENESS	8 COMMUNICATIVENESS	9 COMPATIBILITY	10 COMPLETENESS	11 COMPLEXITY	12 COMPREHENSIVENESS	13 CONCISENESS	14 CONFORMANCE	15 CONNECTIVITY	16 CONSISTANCY	17 CORRECTIVENESS	18 COUPLEDNESS	19 EFFECTIVENESS	20 EFFICIENCY	21 ELASTICITY	22 ERROR-TOLLERANCE	23 EXPANDABILITY	24 FEASIBILITY	25 FLEXIBILITY	26 GENERALITY	27 INDEPENDENCE	28 INTEGRITY	29 INTEROPERABILITY	30 INTRAOPERABILITY
ACCESSABILITY 1			2	2	2		3	2				2			3			2												3
ACCURACY 2										3				3		3	2		3									3		3
ADAPTABILITY 3					2				3	3							2		3	3	3		2		2	3				
AUDITABILITY 4			3								3	3		3	3	3														
AUGMENTABILITY 5	3		2			3	3	3						3									2		2					
COERCION 6							3				2																			
COHESIVENESS 7	7		3	3				3			3		3	3	2	3		2	3								3			
COMMUNICATIVENESS 8	3			3	3				3			3																	3	
COMPATIBILITY 9																													2	
COMPLETENESS 10				3								2																		
COMPLEXITY 11				3	3													3								3				
COMPREHENSIVENESS 12	3							2		3					3											3				
CONCISENESS 13										3	2			3	3	3				2										
CONFORMANCE 14			3	3																										
CONNECTIVITY 15	2						3	3										2												
CONSISTANCY 16			3	3									3																	
CORRECTIVENESS 17		2																												
COUPLEDNESS 18	3					3					3				3												3			3
EFFECTIVENESS 19								3				3		3						3										
EFFICIENCY 20																														
ELASTICITY 21	3		2		3																		2		2	3				
ERROR-TOLLERANCE 22																														
EXPANDABILITY 23	3		2		2				3																					
FEASIBILITY 24																														
FLEXIBILITY 25	2		2		3																3					3				
GENERALITY 26			3																						3					
INDEPENDENCE 27															3			3					3		2					
INTEGERITY 28														3																
INTEROPERABILITY 29			3																											
INTRAOPERABILITY 30											3				3			2												
LEGIBILITY 31								3					3																	
MAINTAINABILITY 32	3			3																										
MEASUREABILITY 33														3		3														
MODIFIABILITY 34	3		2		2				3														2		2					
MODULARITY 35	3		3	3			3				3				3			2							2					
OPERABILITY 36																														
PORTABILITY 37		2																												
PREDICTABILITY 38			2		2																									
READABILITY 39	3			2	2			2				3	3																	
REDUNDENCY 40													2											3			3			
RELIABILITY 41		1															1													
RESILIENCY 42	3		3		3																	2								
REUSABILITY 43	3		3															3		3	3		3		3	3				
ROBUSTNESS 44																						2								
SECURITY 45																												2		
SELF-DESCRIPTIVE 46	3		3	2	3			3				2											3							
SENSITIVITY 47		3																												
SIMPLICITY 48								2			2		2			2									3	3				2
STABILITY 49																														
STRUCTUREDNESS 50				2	3		3	3			2					2		2			3		3		3	3				
SURVIVABILITY 51										3	3				3			3				2						2		
TESTABILITY 52																	2													
TRACEABILITY 53	2			2	3										2															3
TRANSPORTABILITY 54									2									3											2	
TRUSTWORTY 55		3																												
UNAMBIGUITY 56																2														
UNDERSTANDABILITY 57	2		3	2	2			2					3			2						3		3						
UNIFORMITY 58				3	3							2	2	2		2							3							
USABILITY 59	2							1																						
VERACITY 60		2														2	2													

	31 LEGIBILITY	32 MAINTAINABILITY	33 MEASUREABILITY	34 MODIFIABILITY	35 MODULARITY	36 OPERABILITY	37 PORTABILITY	38 PREDICTABILITY	39 READABILITY	40 REDUNDENCY	41 RELIABILITY	42 RESILIENCY	43 REUSABILITY	44 ROBUSTNESS	45 SECURITY	46 SELF-DESCRIPTIVE	47 SENSITIVITY	48 SIMPLICITY	49 STABILITY	50 STRUCTUREDNESS	51 SURVIVABILITY	52 TESTABILITY	53 TRACEABILITY	54 TRANSPORTABILITY	55 TRUSTWORTY	56 UNAMBIGUITY	57 UNDERSTANDABILITY	58 UNIFORMITY	59 USABILITY	60 VERACITY
ACCESSABILITY 1	2	3	2	2	3	3			2	3										3			2						2	
ACCURACY 2			2				2							3					3			3								
ADAPTABILITY 3				1				3	2				2			3		3		2							2			
AUDITABILITY 4		3		3			3	2					3		3	3				2		2	2					3	3	
AUGMENTABILITY 5				2	2						3									2		3	3				3			
COERCION 6																														
COHESIVENESS 7	3	3			3						3		3		3	3		3		2		3				3		3		
COMMUNICATIVENESS 8			3	3					3		3															3	3		2	
COMPATIBILITY 9								3					3											3					3	
COMPLETENESS 10																													3	
COMPLEXITY 11	2	2						2							3					3							3	3		
COMPREHENSIVENESS 12	3	3	3	2					3							2											3		3	
CONCISENESS 13		3									3				3								3			3	3	3	3	
CONFORMANCE 14		3	2				3															3							3	
CONNECTIVITY 15					3																		3							
CONSISTANCY 16	3	2	3						3																	3	3	2		
CORRECTIVENESS 17	3						3				1										2					3				2
COUPLEDNESS 18			3							2										2										
EFFECTIVENESS 19											3																			
EFFICIENCY 20			3																										3	
ELASTICITY 21																														
ERROR-TOLLERANCE 22											3																		3	
EXPANDABILITY 23				2																										
FEASIBILITY 24																														
FLEXIBILITY 25				3									3										3							
GENERALITY 26																													3	
INDEPENDENCE 27				2				1					2																	
INTEGERITY 28																														
INTEROPERABILITY 29																														
INTRAOPERABILITY 30																														
LEGIBILITY 31		3					3		1							3										1	1			
MAINTAINABILITY 32				3							3											3	3				1			
MEASUREABILITY 33				3					3		3											2	2				3			
MODIFIABILITY 34								2	3		3															3	3			
MODULARITY 35		2									3		2									2	3			3				
OPERABILITY 36																													2	
PORTABILITY 37											2																			
PREDICTABILITY 38																								1						
READABILITY 39	2			3												1											1			
REDUNDENCY 40																					2								3	
RELIABILITY 41							2							2					2		2								1	
RESILIENCY 42											2			3					2											
REUSABILITY 43								3													2		3	3					2	
ROBUSTNESS 44							3				2	3							3											
SECURITY 45																									2					
SELF-DESCRIPTIVE 46	2	2		2					2		3												2			3		2	3	
SENSITIVITY 47							3				2			3					2											
SIMPLICITY 48	2	3	3	3							3					2				2		3	3		3	2	2	2		
STABILITY 49												2					2													
STRUCTUREDNESS 50	3	2			3				3							3		3				3	3				3	2		
SURVIVABILITY 51											2	2		2																
TESTABILITY 52		1					2				3																			
TRACEABILITY 53	3	2	3				3		3							3		3				2				3	3			
TRANSPORTABILITY 54								2																						
TRUSTWORTY 55																													2	
UNAMBIGUITY 56	3	2	3						2							3		2									2			
UNDERSTANDABILITY 57	2	2	3	3			3		1							2				3						2				
UNIFORMITY 58	2								2								3		3							3	2			
USABILITY 59											1		2														3			
VERACITY 60							3				2														2	3				

KEY:
1 = INTERDEPENDENT
2 = DIRECTLY DEPENDENT
3 = INDIRECTLY DEPENDENT
NO ENTRY = INDEPENDENT

APPENDIX F

APPENDIX F

THE ATTRIBUTES OF THE ANALYSIS PHASE

The analysis attributes are developed by Shaikh M. U. [95] as a part of the ALVEY project test specification and quality management. These attributes are given below:

Documentational Attributes

D1.Communicability -how the results are expressed

D2.comprehensiveness

D3.Identifiability -the extent to which analys or its analysis components can identiy.

D4.Conciseness

D5.Legibility

D6.Self-descriptiveness

D7.Self-explainability

D8.Structuredness -the structure of document which express the analysis

	results.
D9.Automation.	??????????????????????
D10.Style	-conventions & procedures followed in the output of the analysis results.
D11.formality	-the formal language followed to express the analysis output.

Relational Attributes

R1.Complexity	-extent & number of constituent parts, degree of complications or intricacy, the extent to which certain attributes overlap; in human terms; of the analysis results, of further actions/tasks.
R2.Simplicity	
R3.Structuredness	the structure showing the relationship of the tasks, subsystems,etc..
R4.Traceability	the ability to trace (a) the needs/requirements of the user/management, (b) the consequences of the possible changes.

Performance

P1.Effectiveness	
P2.Predictability	yielding the results which deliver always the same design.
P3.Speed	yielding the results which may lead towards slow/fast design.

Change

C1.Amendability	the extent to which output can be changed according to the needs.
C2.Augumentability	the extent to which additional features can be added.
C3.Expandability	capability of being expanded with respect to (a) requirements (b) facilities (c) etc..
C4.Extendability	the provision to have extension facilities, applications,etc..
C5.Flexibility	able to adjust according to the circumstances.
C6.Maintainability	capability for (a) correction

	(b) enhancement
	(c) efficiency
	(d) etc..
C7.Stability	the output which is unlikely to be changed or have the minimum chance to be changed at a later stage.

Evaluation Attributes

E1.Auditability	the provision of traces and diagnosis.
E2.Completeness	level of details of the output with respect to something missing e.g. detailed level of deta model, process model etc...

APPENDIX G

APPENDIX G

THE LIFE CYCLE SUMMARY

The summary of the life cycle is given for the purpose of facilitating the determination of the distinct phases of the system development life cycle. The life cycle which is considered in this study is the one which is developed by Shaikh M. U. [50]. This is because this life cycle is of global nature and it is applicable to most of the software systems.

The Summary Of The Life Cycle

Phases	Starting Point	End Product
1 * Project Proposal	Proposal about what the user & Management want to have.	Recommendations about Analysis Method/Methodology, statement of needs & brief plan, estimates about manpower required, costs, time etc..
*Study of proposal	Study of the Proposal given by the User & Management.	comments, notes, etc., about the proposal and informal recommendations, and a report etc..
*Initial Investigation	Investigating the proposal using the informal report, comments, notes, etc..	statements postulating actual needs, costs, benifits, etc..
*Initial Feasibility study	Output of initial investigation Phase.	Recommendations about Analysis Method/Methodology, statement of needs and brief plan, estimates about completion date, manpower required, costs, etc..
2 * Strategy	output Report of Project Proposal Phase i.e. initial feasibility study sub-phase.	Strategy Plan i.e. how to proceed with the Project.
3 * Analysis	Statements of needs, brief plan etc. obtained from Phase 1	Detailed proposals showing definitions of Requirements, Requirements Specification, important parameters, information about environment, etc., statistics about costs, time, manpower etc. and their feasibility to be used for developing System Specifications.

*Analysis of environment	any available information about working, working area, terminologies being used by the User/Management etc..	Short report showing adequate information about the environment working area.
*Requirements Analysis	Report on Analysis of the environment, the statement of needs, brief plan etc..	Statement showing the agreed Requirements of the User/Management.
*Prepare Requirements Specification.	Report of Requirements Analysis	Particular comprehensive and precise statement of Requirements from the User/Management and the environment point of view.
*Check for consistency	Statement of Requirements Specifications	A critically examined report against any contradiction, misleading components, etc..
*Search for existing available System	The output of sub-phase check for consistency	May lead to Implementation Phase if there is available a suitable System which is according to Requirements Specification.
*Feasibility Analysis and Detailed Proposal(s)	The output of sub-phase check for consistency	Set of detailed proposal(s) giving recommendations, conclusions, statistics about costs, time, etc., and possible benifits.
4 * System Specifications	The selected detailed Proposal i.e. output of the Analysis Phase.	The technical Plan having set of different clear steps to be used as a basis for System Design.
5 * Design	System Specifications which are in terms of set of clear steps expressed in technical terms.	Integrated details about Modules, Sub-systems, their interfaces to other Modules or sub systems, their controls, constraints etc.. This is called as frame or skleton.

*Planning & Decission	Use of System Specifications	Basic structure of Software and series of hierarchical steps, definitions, main Functions, Processes, operating parameters, I/O data, etc..
*Conceptual Design	Output of sub-phase Planning & Decission and System Specifications.	Basic assumptions about Design and ideas creating Modules, sub-systems, data bases, etc.. Models about entity types, data element types, data structure, data flow diagram, etc..
*Physical Design	Output of subphase Conceptual Design.	Integrated details about data format, data structure, data base, data dictionary, Functions, Processes, Modules, sub-systems, their interfaces to other Modules, sub-systems etc..
6 * Implementation	Frame or sklelton of Software System Developed during the Design Phase.	Detailed programme statements in suitable programming language(s) and its documentations.
7 * Transition	1. System Specifications, information about Functions, real world, etc.. 2. Detailed program statements in a programming language and its documentations.	A working System

APPENDIX H

APPENDIX H

SOFTWARE QUALITIES

These implementation phase attributes are defined by M. A. Hennell [34]. These attributes are defined in the terms of general meanings which are applicable to the implementation phase of the software system development life cycle.

1. Accessibility: Are all items needed for comprehension available e.g. are include or copy statements present. c.f. Comprehensiveness.

2. Communicativity: able to impart information .c.f. readibility.

3. Comprehensiveness: the extent to which all the necessary information is present.

4. Identifiability: the extent to which documents or components of documents can be located.

5. Legibility: capable of being read. Clarity of the objects and their presentation.

6. Readability: the ease with which the software can be read and understood.

7. Self-descriptiveness: the extent to which the entity contains adequate commentary on what the entity is and its purpose.

8. Self-explainability: The extent to which the entity contains adequate commentary on its mode of performance.

9. Style: the presentation (or layout) of the test and information relating to the text.

10. Understandability: the property of being understood by more than one person e.g. uniform style, standard names. c.f. Comprehendability, Readability.

11. Coercion: the extent to which unrelated objects have been made to combine togother.

12. Cohesiveness: classes appear to be contentious. Do the components combine together according to some criteria?

13. Complexity: see simplicity. The extent and number of constituent parts. The degree of complications or intricacy. The extent to which certain attributes overlap. The degree of interdependence.

14. Connectivity: the ease with which components link together.

15. Consistency: with respect to some criteria or some requirement. Are contradictions present?

16. Coupledness: the extent to which various elements are interrelated (see connectivity).

17. Independence: not reliant on external entities, the extent of such reliance, use of overlap or equivalence.

18. Interoperability: the ability of an entity to be linked to another externally.

19. Intraoperability: the linkage between the internal components of a program or system.

20. Modularity: the criteria by which the software is decomposed into smaller units.

21. Simplicity: the extent to which specific characteristics are present without complication.

22. Structuredness: the extent to which the software contains structure of specific types

23. Traceability: the ability to trace

    a) the requirements to the corresponding design or code,

    b) the flow of control,

    c) the flow of data,

d) the consequences of a modification.

24. Accuracy: the ability to produce results within a certain range.

25. Conciseness: minimality with respect to some criteria.

26. Effectiveness: performs its task completely and efficiently.

27. Efficiency: the extent to which a software item consumes resources. Such resources can be temporal, spacial, human, fiscal etc..

28. Error-tolerance: able to recover from:
    a) internal errors,
    b) external errors.

29. Predictability: the characteristics of always yielding the same results for the same inputs.

30. Redundancy: the extent to which there is overlap between processing elements. The amount of duplicated or unnecessary features.

31. Reliability: the perception that a software item can perform a required function under stated conditions for a stated period of time without error.

32. Resilience: the ability to recover from specific events. Containing an error recovery mechanism.

33. Stability: unlikely to change c.f. Resiliant, Robust the ability to perform its tasks correctly even though the environment may be incorect.

34. Adaptability: capable of change for other purposes (see amenability).

35. Amenability: the extent to which a document can be changed (see adaptability).

36. Augmentability: the extent to which additional features can be added. c.f Enhanceability, Modifiability, Maintainability.

37. Changeability: the extent to which components of a document which are subject to change can be identified and altered.

38. Elasticity: capable of being extended by:
    a). application to more general problems, see Generality,
    b). permitting additional features to be added, see Enhanceability, Expandability.

39. Expandability: capable of being expanded:
    a) with respect to entities,
    b) with respect to facilities or functionality.

40. Maintainability: capability for:

    a) correction,

    b) enhancement,

    c) performance improvement

       c.f. Efficiency.

41. Modifiability: ease with which changes can be made c.f. Maintainablity.

42. Repairability: the extent to which faults can be located and corrected.

43. Auditability: the ease of auditing. For example provision of traces and diagnostics. The extent to which knowledge of the program actions can be determined. c.f. Analysability.

44. Analysability: capable of analysis. e.g. is control flow predictable. (c.f. Cobol). Are pointer variables being used? Is arithmetic performed on pointers?

45. Completeness: complete with respect to some criteria or some requirement. Is something missing?

46. Conformance: satisfies standards.

47. Correctness: correct with respect to some criteria c.f. consistency.

48. Feasibility: the extent to which the software can be developed within a specific timescale, within specific resource limits and by the personnel available the extent to which inputs can be supplied to execute particular components., c.f. efficiency, readability, conciseness.

49. Functionality: the extent to which the functions are present. The nature of the functions.

50. Integrity: trustworthy, predictable, correctness, the ability of the system to prevent unwanted access to internal items, see Security.

51. Measureability: capable of being measured.

52. Sensitivity: the stability of programs outputs with respect to internal software errors.

53. Testability: the extent to which the software is capable of being tested.

54. Unambiguity: the extent to which the software is free of interpretation.

55. Uniformity: the extent to which the software is constructed by a common method, technique or style.

56. Veracity: c.f. Correctness, Trustworthy Reliability of outputs.

57. Friendly: easily understandable.

58. Learnability: the difficulty associated with learning how to use the software.

59. Operability: the extent to which the software can be operated.

60. Usability: the effort required to learn, operate, prepare input and interpret output of a program.

61. Safety: the extent to which the software causes danger either to humans or to other systems or data repositories.

62. Security the protection of the code against corruption, the protection of the internal data from unauthorised users, the protection of the code from additions.

63. Trustworthy: capable of performing its tasks without unwanted side-effects.

64. Flexibility: capable of being expanded, elastic, portable, generality.

65. Generality: capable of being used on more complex problems, extensible, expandable, flexible, portable. solving a wider class of problem than the current one.

66. Independent: device independent, language independent, o/s independent, machine independent.

67. Portability: the ability to move between:
    a) dialects (of language or notation),
    b) machines,
    c) environments,
    d) people.

68. Reusability: the extent to which entities can be re-used. These may be; declarations, data structures, operators, procedures, modules, sub-systems, programs, systems.

69. Robustness: the ability to function correctly when supplied with erroneous data or if its operating environment is changed. Planned to cope.

70. Durability: the deterioration of the product with respect to time.

71. Responsive: the ability to cope with external events.

72. Timeliness: the availability of features at a particular time.

73. Up-to-datedness: the extent to which the information in the document reflects the current state. c.f. Timeliness.

APPENDIX I

APPENDIX I

REFERENCES

1 Curtis B.,
"Research on Software Complexity, in Proceedings of The Work Shop on Quantitative Software Models for Reliability, Complexity, and Cost", New-York, IEEE, 1980, PP. 1-12.

2 Halstead M.H.,
"Elements of Software Science",
New-York, Elsevier North-Holland INC, 1977.

3 Curtis B., Sheppard S.B., Milliman P.M.,
Third Time Charm, "Stronger Prediction of Programmer Performance by Software Complexity Metrics", Proceeding of 4th International Conference on Software Engineering, PP. 356-360, 1979.

4 Fitzsimmons A.B. and Love L.T.,
"A Review and Evaluation of Software Science, ACM Computing Surveys, 10, PP. 3-18, 1978.

5 McCabe T.J.,
"A Complexity Measure ",
IEEE Transaction on Software Engineering,
SE-2, PP. 308-320, 1976',

6 Henry S.M. and Kafura D.,
"Structure Metrics Based on Information Flow",
IEEE Transaction on Software Engineering,
Vol. SE-7, No.5, PP.510-518, Sep. 1981.

7 Yourdon E.,
"Structure Design", Fundamentals of
a Discipline of Computer Program
and System Design", 1978.

8 Woodward M.R., Hennel M.A. and Hedly D.,
"A Measure of Control Flow Complexity
in Program Text", IEEE Transaction on
Software Engineering PP. 45-50, 1979.

9 Kafura D. and Canning J.,
"A Validation of Software Metrics Using Many
Metrics and Two Resources",
Proceding of 8th International Conference on
Software Engineering, August, PP. 378-385,1985

10 Jackson M.,
"Principles of Program Design",
Academic Press, London, 1975.

17 Parnas D.L.,
"A Technique on the Criteria to be Used
in Decomposing System into Modules",
Comm. of the ACM, 15, PP. 1053-1058, 1972.

18 Parnas D.L.,
"A Technique for Software Module Specification
with Examples", Comm. of ACM, 15, 5, PP. 330-336, 1972.

19 Myers G.J. Stevens N.P. and Constantine L.L.,
"Structured Design, IBM System Journal,
Vol. 13, PP. 115-139, 1974.

20 Meiler P.J.,
"The Practical Guide to Structured
Systems Design", 1980.

21 Emerson T.J.,
"A Discriminant Metric for Module Cohesion",
Proceeding of 7th International Conference
on Software Engineering",
PP. 294-303,

22 Hamer P. G and Frewin G. D.,
"M. A. Halstead's Software Science
A Critical Examination", Proceeding of
6th International Conference on Software
Engineering, September, 1982, PP, 197-206, 1982.

23 Baker A.L. and Zweben S.H.,
"A Comparison of Measures of Control Flow Complexity"
IEEE Transaction on Software Engineering,

Vol. SE-6, no.6 ,Nov. 1980, PP. 506 -512.

24 Kuck J.,
"The Structure of Computers and Computation",
Vol. 1, New-York, 1978.

25 Born G.,
"Controlling Software Quality",
Special Issue on Controlling Software Project",
"Software Engineering Journal",
Vol. 1 No. 1 Jan. 1986.

26 Frewin G.D. and Hatton B.J.,
"Quality Management Procedures and Practices"
Software Engineering Journal, Vol. 1 PP. 29-38,
Jan. 1986,

27 Jones T.C.,
"Measuring Programming Quality and
Productivity", I.B.M. System Journal,
Vol. 17 No. 1, PP. 9-64, 1978.

28 Boehom B.W.,
"Software Engineering Economics",
IEEE Transaction on Software Engineering, 1984,
SE-10(1) PP. 4-21.

29 Boehom B.W., et al.,
"Characteristics of Software Quality",
TRW Series on Software Technology,
Vol. 1 North-Holland, 1978.

Conference, PP. 326-368, May 1986.

36 Brooks F.P.,
"The Mytical Man Month, Essay in
Software Engin. Reading",
MA: Addison Wesley, 1975.

37 McClure C.L.,
"A Model for Program Complexity Analysis",
Proceding of 3rd International Conference
on Software Engineering, Atlanta, GA, May,
PP. 149-157, 1978.

38 Sinha P.K., Jayaprakash S. and Lakshmanan K.B.,
"A New Look at the Control Flow Complexity
of Computer Programs",
IEE Computing Series 6, Software Engineering,
SE/86, PP. 88-102, 1986.

39 Harrison W.,Mogel K., Kuxzny R. and Dekock A.,
"Applying Software Complexity Metrics
to Program Maintenance",
Computer. 15(9), Pp. 65-79. 1982.

40 Chen E.T.,
"Program Complexity and Program Productivity",
IEEE Transaction on Software Engineering,
SE-4, 1978, PP. 187-194.

41 Myers G.J.,
"A Software Reliability, Principle
and Practices", A Wiley-Interscience Publication,

John Wiley and Sons, 1976.

42 Kafura D. and Henry S.,
"Software Quality Metrics Based on Interconnentivity", the Journal of System and Software, 2, PP. 121-131, 1981.

43 Canning J.T.,
"The Application of Software Metric to Large-Scale System",
Ph.D. Thesis, Computer Science Department, Virginian Polytechnic Institute, April 1985.

44 Myers G.J.,
"An Extension to Cyclomatic Measure of Program Complexity", ACM Sigplan Notices, Vol. 12, No. 10, Oct. PP. 61-64, 1977.

45 Yau S.S. and Collofello J.,
"Some Stability Measure for Software Engineering Maintenance", IEEE Transaction on Software Engineering, Vol. SE-6, NO. 6 , PP. 545-552, NOV. 1980.

46 McCall J.A. , Richard P.K. and Walters G.F.,
"Factors in Software Quality",
Technical Report 77 CISO2, Vol. 1, 2, and 3,
Sunnyvale, CA.,
General Electric Command and Information Systems, 1977.

47 Haney F.M.,
"Module Connection Analysis",
A Tool for Scheduling Software Debugging
Activities, Proceeding Fall Joint Computer
Conference, PP. 173-179, 1972.

48 Chapin N.,
"A Measure of Software Complexity",
AFIPS. Conference Proceedings,
Vol.48, INC 1979, PP. 995-1002.

49 Curtis B.,
"Measurement and Experimentation in Software
Engineering", Proceeding of the IEEE, Vol. 68
No. 9, PP. 1144-1157, september, 1980.

50 Shaikh M.U.,
"Analysis and Comparison of System development
Methodologies in Software Engineering",
Ph. D. Thesis, 1986, University of
Liverpool, U.K.

51 Cramford S. Mcintosh A. and Pregibon D.
"An Analysis of Static Metrics and
Faults in C. Software",
The Journal of system and Software
Vol. 5 No. 1, Feb. 1985.

52 Lehman M.M.,
"Programs, Life Cycle, and Laws of
Software Evolution"

Proceeding of IEEE, Vol. 68, No. 9, September, PP.1060-1076, 1980.

53 Halstead M.H., "Natural Laws Controling Algorithm Structure", ACM Sigplan Notice Vol. 7 No. 2,Feb. 1972.

54 Elshof J.L., "An Investigation Into the Effects of the Counting Method on Software Science Measurement", ACM. SIGPLAN Notice Vol. 13, No.2, PP. 30-45, Feb. 1978.

55 Baker A.C. and Zweben S.H." "The Use of Software Science in Evaluating Modularity", Concepts", IEEE Transaction on Software Enginnering Vol. SE-5, No.2 March 1979, PP. 110-120.

56 Oviedo E.I., "Control Flow, Data Flow and Program Complexity", Proceeding of Compsac 80, Chicago, ILL, PP. 146-152.

57 Boffey T.B., "The Linearisation of Flow Charts", BIT, Vol.15 PP. 341-350, 1975.

58 Curtis B., Sheppard S.B., Milliman P.M., Borts M.A. and and T. Love, "Measuring the PsychologicaL Complexity of Software, Maintenance Tasks with Halstead and MCcabe Metrics" IEEE. Trans. Software Engineering 5(2), 1979, PP. 95-104.

59 Prather R.E., "An Axiomatic Theory of Software Complexity Measure", The Computer Journal, Vol. 27, No. 4 , PP. 40-347, 1984.

60 Oulsnam G., "Cyclomatic Number do not Measure Complexity of Unstructured Program", Information Processing Letters, Vol. 9, No. 5, PP. 207-211, Dec. 1979.

61 Tausworthe R.C., "Standarized Development of Computer Software", Prentice-Hall, Englewood Cliffs, NI, 1977.

62 Lunderberg M., Goran Goldkunl, and Nilsson, N., " Information systems development a Systematic Approach", University of Stockholm, Prentice-Hall, INC., Englewood Cliffs, New-Jersey 07632, 1981.

63 Gilb T.G., "Software Metrics", Cambridge, MA:, Winthrop, 1977.

64 Yourdon E.,
"Structured Design",
Workshop, Edition 2.1, Yourdon Inc.,
N.Y., 1980.

65 Enger N.L.,
"Classical and Structured Systems Life
and Documentation",
Systems Analysis and Design a Formulation
for 1986, PP. 1-24, Elsevier North Holland INC.,
U.S.A, 1981.

66 Hennel M.A.,
"Metric Evaluation",
Syntax-Oriented Programme Handling and
Instrumentation for ADA"
Document: LDRA/009, Version 2,
October, 1986.

67 Kearney J.K., Sedlmeyer R.L., Thompson W.B.,
Gray M.A. and Adler M.A.
"Software Complexity Measurement"
Communication of ACM Vol. 29,
No. 11, November, 1986.

68 Boehom B.W.
"Software Engineering Economics",
Prentice Hall,EngleWood Cliffs, NJ, 1981.

69 Ishikawa K.,
Translate by Lu J.D.,

"What is Total Quality Control?",
the Japanese Way, Prentice-Hall, INC.
Englewood Cliffs, N.J. 1985.

70 Kitchenham B.A. and Walker J.G.,
"The Meaning of Quality"
Proceeding of the BCS/IEE,
Software Engineering 1986, PP. 393-406, 1986.

71 Naur P.,
"A Model and the World",
Proceeding of the BCS / IEE,
Conference SE 86, September, 1986.

72 Drummond S.,
"Measuring Applications Development
Performance",
Datamation Feb. 1985, PP. 102-108.

73 Albrecht A.J.,
"Function Points Helps Managers, Assess Applications,
Maintainence Value"
Computer World (special report),
August 26, 1985, PP. 20-21.

74 Behrens C.A.,
"Measuring the Productivity of Computer Systems
Development Activities With Function Points".
IEEE Transaction on Software Engineering,
Vol. SE-9, No.6, Nov. 1983, PP. 648-652.

75 Albrecht A.J., John E., and Gaffeny JR.,
"Software Function, Source Line of Code and Development Effort Prediction: A Source Science Validation", IEEE Transaction on Software Engineering, Vol. SE-9, No.6, Nov. 1983, PP. 639-648.

76 Albrecht A.J.,
"Measuring Application Development Productivity " in Proc. IBM Application Develop. Sympsium, Montery, CA, 14-17th Oct.1979., PP. 83

77 Hennel M.A.,
"Software Qualities",
Project, Test Specification and Quality Management, UOL/023, Version 1, August, 1986.

78 Kitchenham B.A. and Walker J.G.,
"The Place of Checking and Testing in the Software Life Cycle",
Working Paper for Project Review and Comments, ICL 021, Version 1, Feb. 1987.

79 Bowen, Wigle, and Tsai,
"Specification of Software Quality Attribute Final report"
D182-11678-1,
D182-11678-2,
D182-11678-3,
Final report, Boeing Aerospace Co.
Seattle, WA. October, 1984.

80 Hedley D. and Hennel M.A.,
"Definition and Meaning of Quality",
Project, Test Specification and Quality
Management, UOL/005, Version 2, August, 1986.

81 Woodward M.R.,
"The Application of Halstead's Software Science
Theory to ALGOL 68 Program", Software-Practice
and experience, Vol. 14(3), PP. 263-276, March,
1984.

82 Lloyed S.M.,
"Roget's Thesaurus of English Words and
Phrases", 1982.

83 G and C Merriam Company,
Longman New Universal Dictionary, 1982.

84 Kitchenham B.A. and Walker J.G.,
"Test Specification and Quality Management, the
Meaning of Quality ", Deliverable A21, Alvey
Project SE/031, Report to the Alvey Directorate,
27th, May, 1986.

85 Fagan M.E.,
"Design and Code Inspections to
Reduce Errors in Program Development",
IBM System Journal, 15(3), PP.182-212, 1976.

86 Yeh R.T.,
"Guest Editorial on Software Science",
IEEE Transactions on Software Engineering,
Vol. SE-5, No. 2, March, PP. 74-75, 1979. .

87 Kitchenham B.A.,
"Measuring of Programming Complexity",
ICL Technical Journal, PP. 298-316, May, 1981.

88 Myers G.J.,
"Reliable Software Through Composite Design",
N.V. Van-Nostrand, R. Heinhold 1975.

89 Troy D.A. and Zweben S.H.,
"Measuring the Quality of Structured Designs",
the Journal of Systems and Software, 2 ,
PP. 113-120, 1981.

90 Ho˜cker H., Itzfeldt M. and Timm M.,
"Comparative Description of Software Quality
Measurement", GMD-Studien, March, 1984.

91 Freigenbaum A. V.,
"Total Quality Control", Third Edition, 1983,
McGraw-Hill Book Company.

92 Garvin D.A.,
"What Does Product Quality Realy Means",
Sloan Management Review Fall 1984, PP. 25-43,
1984.

93 Christesen K., Fitsos G.P. and Smith C.P,
A perspectiv on Software Science", I.B.M. System Journal, Volume 20, Number 4, PP. 372-387, 1981.

94 Stroud J.M.,
"The Fine Structure of Psychological Time", Information Theory in Psychology, the Free Press, Chicago, ILL., 1956.

95 Shaikh M.U.,
" Private Communication at Special Meeting of the ALVEY Directorate, in Malvern, U.K., february, 1987.

96 Monro D. M.,
" FORTRAN 77", Imperial College of Science and Technology University of London, Donald M. Monro, 1982.

97 Hecht, M.,
"Flow Analysis of Computer Program", New York, North-Holland, 1978.

98 Allen F.E.,
"Interprocedural Analysis and the Information Derived by it", in Lecture Notes in Computer Science, Vol 23, Wildbad, Germany: Springer, PP. 291-321, 1974.

99 Ejiogu, L.O.,
"The Critical Issues of Software Metrics", Sigplan Notice, Volume 22,Number 3, PP. 59-64, March, 1987.

100 Ryder B. G.,
"The PFORT Verifier", Software-Practice and Experience, Volume 4, PP. 359-374, 1974.

101 Van Verth P. B.,
"A System for Automation Grading Program Quality, Suny (Buffalo), Technical Report Number 85-05 (1985).

102 Scheidewind N. F,
"Software Metrics for Aiding Program Development and Debugging", National Computer Conference, PP. 989-994, 1979.

103 Gilb T.,
"Design by Objective", North-Holand, Planned 1985.

104 ISO/TC 176(1984),
Resolution adopted at March 1984, meeting of TC 176 for inclusion in Submission DP 8402 as a Draft International Standard.

105 Kitchenham B.,
"Personal Communication".

106 Hennell M. A.,
"Personal Communication".

107 Woodward P. M., Wetherall, P. R. and Gorman B.,
Official Definition of Coral 66, HMSO. London, 1970.