CHAPTER - 1

INTRODUCTION
CHAPTER 1

INTRODUCTION

1.1 MOTIVATION FOR THIS STUDY

It is a well known fact that costs of development and maintenance of large software systems have become unacceptably high. Also, maintenance cost outweighs the development cost of a typical large software system over its life cycle. Estimates suggest that about 40 to 50 % of annual software expenditure involve maintenance of existing systems. Thus, an increasing importance is being felt for the measurement of various characteristics of software systems [1-2]. It is only through a process of measurement, it will be possible to know whether the new software development techniques are having the desired effect on reducing the problems of software development and maintenance.

Also, if time period increases, then maintenance costs for a typical organization takes a greater share of the total computing budget. Fig. 1.1 highlights the effect of lack of systematic design techniques on variation in expenditure on software development and maintenance [4]. It is clear from this figure that if systematic techniques are not used in the design phase of a software development life cycle, the development costs may be transferred to the maintenance phase. So, in order to reduce the maintenance cost, software engineers must be vigilant throughout the software development life cycle. Otherwise, due to the increased expenditure on software maintenance phase, less resources will be left to be allocated for enhancement of the existing systems or for the development of new systems.

In order to control the rising costs of software development and maintenance, a new discipline called ‘Software Engineering’ has emerged, which provides a scientific life cycle approach for software development. The basic goal of this discipline is to develop a good quality software with minimum cost and within the limited allowed time-period. Software life cycle involves several phases with each phase having its own starting and ending point. These are sequential in the sense that the output of the one phase becomes the input of the following phase. So, these are interdependent because changes made in one phase may significantly affect the activities of other phases. Broadly speaking these are:

(i) Feasibility Analysis Phase
(ii) Requirements and Specifications Phase
(iii) Design Phase
(iv) Coding Phase
(v) Testing Phase
(vi) Acceptance Phase
(vii) Maintenance Phase

The feasibility phase takes place before the start of the actual development of the software product and the maintenance phase after its delivery. Each of these phases is valuable in its own right but the requirements and design phases may have more impact on maintenance phase than others. A poorly designed system can be made working earlier but it may cost more in due course of time.
Fig. 1.1: Effect of lack of systematic design techniques on Variation in Expenditure on Software Development and Maintenance [4]
because of the additional cost of faults repair[3]. So a software engineer should be more careful in these phases.

In software industry, a great deal of effort is now being devoted for developing techniques to produce reliable and cost effective software systems. The basic need for this is to develop programs which are easy to understand and maintain. For this purpose, system design techniques have gained great importance. Examples of such methodologies are Composite Design and Structured Analysis Design techniques. These techniques provide a step by step procedure which guides the system designer. But, these techniques do not serve as measures of the quality of design [4-5]. At the most, the developer can make a qualitative assessment of how judiciously the technique has been applied. Therefore, the decision process about the quality needs to be augmented by quantitative indicators.

It is of no use to address the issue of development effort without addressing the quality of the software at the same time. A poorly designed system may be made operative quickly but it may cost more in the long run because of the additional cost of repairs. Thus, the major goal of research in software engineering is to improve the quality of the delivered software product. From time to time, attempts [6-8] have been made to define and measure software quality in terms of some high level software characteristics. Some of these are:

- Simplicity
- Readability
- Understandability
- Reliability
- Efficiency
- Portability
- Modifiability
- Testability

The quantitative view of these characteristics in terms of quality metrics may be of great importance for software developers, customers and managers as well. On the basis of these, goals to be achieved by a software product, may be set and the decision about its rejection or acceptance can be made. But, it seems very difficult to quantify a particular quality characteristic through a single metric. So, each of high-level software quality characteristic should be decomposed into primitive characteristics [3,9-10]. For example, reliability may be divided into the following primitives:

- Accuracy
- Consistency
- Completeness
- Self-Containedness

Now, by defining a metric for each of these primitives and by combining the measurements of these primitives, we can get a single metric for reliability.
There is close association between software quality and complexity. Software complexity directly affects maintenance activities like software understandability, modifiability and software testability. Predicting software complexity can save millions in maintenance [11-13]. Clearly, if complexities could somehow be identified, then development, maintenance and testing procedures can be adjusted accordingly. This concern has led to various studies directed towards identification of factors contributing to complexity of programs and development of complexity reducing methodologies.

An important prerequisite for activities like program debugging, modification and maintenance is its comprehension. Program understanding is directly associated with its complexity. So, it is now widely felt that the complexity of programs should be significantly reduced to facilitate their understanding. It is not the computational complexity of the program but its psychological complexity. These two types of complexities should be clearly distinguished. The computational complexity refers to the execution time requirement of the program. While, the psychological complexity refers to those characteristics of program which make understanding of the program difficult [3].

In simple terms, psychological complexity is the difficulty of the program development and its understanding. By intuition, one knows it well but finds difficult to formalize. This difficulty is mainly due to absence of formal theory of program understanding. Also, no general model is available to identify various factors of program complexity. However, some features like size of the program, types of control structures, their nesting level, position of a statement in the logic of a program, data structures have been established as complexity affecting features by various researchers [3,5,14-16].

Different specific types of complexity that have been identified include problem complexity, design complexity, process complexity and product complexity [17-19]. Each of such types of complexity refers to the 'difficulty' that a programmer or analyst encounters in performing such functions like problem analysis, designing, coding, testing or maintenance of software systems. It is also natural to assume that a more difficult task requires more skill and effort than a simpler one. Clearly, if complexities could somehow be controlled or measured, then development and maintenance activities may be implemented more effectively. A relationship between software complexity and various attributes of software systems is depicted in Fig. 1.2. Thus, the main concern in the construction and maintenance of software systems is the controlled introduction of complexity. This concern has motivated research efforts to define and validate quantitative metrics of software complexity [15,20-30].

The notion of software metrics means different to different researchers depending upon the characteristics and the features of the software systems from which metric is derived. Consequently these metrics may be categorized according to the types of characteristics measured by them. Broadly speaking, software metrics may be classified into the following categories [3]:

- Process metrics
- Product metrics

The metrics which quantify the attributes of the development environment and process are termed as process metrics. On the other hand, product metrics are the measures of the software product. In this work, we, in general, are concerned with product metrics and particularly with the software complexity metrics.
Fig. 1.2: Relationship Between Software Complexity Metrics and various attributes of software systems [22]
The concept of software complexity metrics is not new. During the last few decades, various complexity measures have been developed and used [3,31-33]. This activity has led to and been fueled by various encouraging factors for the development of various software models. Study and design of software complexity metrics, faults prediction techniques and establishing relationships among these and software systems characteristics like testability and predicting faults have been the motivation factors for this work.

Advantages of Software Metrics

As stated earlier, software metrics are used for each and every aspect of software systems like size, complexity, testability and so on. Study and determination of software metrics offer a variety of advantages [3,15]. Some of these are:

- In comparative study of various design methodologies of software systems.
- For analysis, comparison and critical study of various programming languages with respect to their characteristics.
- In comparing and evaluating capabilities and productivity of people involved in software development.
- In the preparation of software quality specifications.
- In the verification of compliance of software systems requirements and specifications.
- In making inference about the effort to be put in the design and development of the software systems.
- In getting an idea about the complexity of the code.
- In taking decisions regarding further division of complex module is to be done or not.
- In providing guidance to resource managers for their proper utilization.
- In comparison and making design tradeoffs between software development and maintenance costs.
- In providing feedback to software managers about the progress and quality during various phases of software development life cycle.
- In allocation of testing resources for testing the code.

The above mentioned few number of advantages and other considerations encourage the software engineers, developers, managers and researchers to pursue study and further research in the area of software metrics and software development modeling.
Limitations of Software Metrics

It is well known saying that pros and cons are the two sides of the same coin. Study of software metrics provides a number of uses but at the same time it has some limitations also. These are:

- The application of software metrics is not always easy and in some cases it is very difficult and costly.
- The verification and justification of software metrics is based on historical/empirical data whose validity is difficult to verify.
- These are useful for managing the software products but not for evaluating the performance of the technical staff.
- The definition and derivation of software metrics is generally based on assumptions which are not standardized and may depend upon tools available and working environment.
- Most of the predictive models rely on estimates of certain variables which are often not known exactly.
- Most of the software development models are probabilistic and empirical.

1.2 SUMMARY OF WORK CARRIED OUT

In this work, we have confined ourselves to the study and design of software complexity metrics implemented for modern programming languages, namely, Pascal, C, Ada and Modula - 2. For faults prediction, a technique, namely, bebugging method has been investigated. We have also tried to establish relationships among software complexity metrics, expected number of faults and testability. A brief description of the work carried out by us is given below.

Chapter 2 includes the survey and description of various software metrics which reflect the different characteristics of software systems. Different size metrics like lines of code (LOC), token count and function count have been described. Halstead's theory of software science and McCabe's cyclomatic complexity measure have been explained in detail. Some earlier proposed length estimators have also been described.

Chapter 3 describes the proposed three term length estimator, its experimental study and its comparative analysis with other program length estimators studied in Chapter 2. Various software metrics provided by the theory of software science have been calculated by using the proposed three term length estimator. About 200 programs collected from open literature have been analysed. Counting rules/strategies have been designed for all the above mentioned four languages. It also explains the variations in counting rules used and their effect in resolving the contradiction raised earlier about the level of Ada language. An attempt has, also, been made for predicting size and complexity of a software module before coding.
Chapter four discusses the design of a new program weighted complexity measure. The proposed metric quantifies four prominent complexity factors, namely, size of a software module, types of control structures, their nesting level, and position of a statement in the logic of a program. An attempt has, also, been made to resolve some existing controversial issues of McCabe's metric.

Fifth chapter reports the relationships among the software complexity metrics. It, also, includes the criterion which may be used for the categorization of software systems as "decision bound" and "computation bound" systems.

Chapter six explains the connection between program complexity and faults predicted and it supports the intuition that more the complexity --> more the predicted number of faults. For faults prediction, fault rates for C, Pascal and FORTRAN languages have, also, been reported.

Seventh chapter describes the concept of testability and the conditions for software failure. A method for calculating testability from sensitivity analysis has been explained. It, also, establishes a relationship between program complexity and testability. We observe that more the complexity, more the amount of testing required to reveal expected faults from it.

Chapter eight explains the bebugging technique for faults prediction and its experimental study through a class-room designed experiment to know the effect of complexity over the number of revealed faults. It gives an experimental support to the intuition that the number of faults depends directly on complexity. Comparison between experimentally predicted faults ( through bebugging method ) and theoretically predicted faults ( Lipow’s method ) has also been done and found a positive correlation between the two. However, the results obtained through the experiment on bebugging method are more encouraging than Lipow’s method.

Ninth chapter includes a discussion of the conclusions that can be drawn from this study and the limitations of the present work and also suggests further possibility of research in this area.