COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE

BY

SAMUEL ADEBAYO OLUWADARE
B.Sc. AGRICULTURAL ECONOMICS (IBADAN)

A THESIS IN THE DEPARTMENT OF INDUSTRIAL MATHEMATICS AND COMPUTER SCIENCE, FEDERAL UNIVERSITY OF TECHNOLOGY, AKURE, IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR THE AWARD OF THE DEGREE OF MASTER OF TECHNOLOGY (M.Tech) IN COMPUTER SCIENCE.

JANUARY, 2000

ABSTRACT

Software cost today, represents a significant percentage of computing systems. Inadequate monitoring and evaluation of software development, operations and maintenance could lead to very high cost, failure of projects to meet the targeted or scheduled time, production of low quality software and poor user satisfaction among others. Monitoring and evaluation of software development, operations and maintenance are fraught with a lot of problems among which, is the determination of the appropriate measures of cost, duration, quality and benefits. Unlike in the case of hardware where most of these metrics could be measured directly, most of the metrics associated with software development, operations and maintenance are intangible, that is, they could not be measured directly. This situation probably explains why there has been little efforts at standardising the monitoring and evaluation of software development, operations and maintenance.

In this study, we adopt the use of Project Evaluation and Review Technique (PERT) and the Critical Path Method (CPM) for the estimation of software schedule and cost. The use of PERT/CPM enables us to determine the boundary times like earliest start, latest start, earliest completion, latest completion, total float and free float as well as the critical path of the software life cycle. Piecewise cost-duration curve was used to determine the cost of schedule. McCall's software quality factors and quality metrics are used to derive the data used for software quality performance evaluation. A a software package christened CASMESDOM for monitoring and evaluating the three major phases of software life cycle is developed and implemented in Paradox Relational Database Management System environment. CASMESDOM is menu-driven, user friendly, interactive and intelligent. In CASMESDOM, the practical use of PERT/CPM and the McCall's quality metrics in the estimation of software metrics was demonstrated using the Federal University of Technology, Akure (FUTA) Payroll System.

DEDICATION

This project work is dedicated to the ALMIGHTY GOD.

ACKNOWLEDGEMENT

I want to specially acknowledge the numerous contributions of my project supervisor Prof. O.C. Akinyokun. He was the one that encouraged and nurtured me in the computing profession . But for his love, care and painstaking supervision, this work would not have been a complete success.

My colleagues in the Computer Centre, FUTA: Messrs E.O Olajuyigbe, T.O Olayera, W.O. Fatile, F.O. Sunmola, Robison Okudoh, Miss Esther Fasanmi, Mrs. Toyin Agbonifo, Mrs. Khadijat Wahab, Mrs. A.A. Atoye and Mrs. Tina Okoli, deserves my appreciation for their support and for one or two ideas I have gathered from them. I am grateful to the management and staff of HTRDG Computers Ltd, Akure, for their support in the preparation of the final draft of the thesis.

I want to specially acknowledge the ever constant support and encouragement of my loving parents: Mr. Joshua Ojo Oluwadare and Mrs. Julianah Oja Ibijoke Oluwadare. The following people have also contributed in one way or the other to the success of this work, namely, Prof. S.A. Aluko, Late Major Ayodele Akinola, Mr and Mrs. Osanyingbemi and Mr. Olu Ayeni.

The period of this research was characterised by sleepless nights and very busy days at the expense of the members of my family. I sincerely appreciate my wife Kikelomo Olumuyiwa and my daughter Ifeoluwa for their understanding, love and moral support.

Finally, I acknowledge everyone that has contributed in one way or the other to the success of this work

Samuel Adebayo Oluwadare

CERTIFICATION

This is to certify that this work has been carried out by Samuel Adebayo Oluwadare in the Department of Industrial Mathematics and Computer Science, Federal University of Technology, Akure; and that the work has not been submitted elsewhere for the award of a degree.

Prof. O.C. Akinyokun
Supervisor

05-05-2010
Date

Dr. S.T. Oni
Head of Department

14/8/2010
Date

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

OVERVIEW OF RESEARCH

1.1 Introduction

In recent years, the availability of an efficient software has continued to occupy an increasingly prominent place in matters relating to the success of many businesses and organisations. In [Pressman, 1987], it is reported that, while the first three decades of computing focussed attention on the reduction of cost of computer based systems by reducing hardware cost, the emphasis now is to reduce cost of the development, operations and maintenance of software system.

The challenge today is to harness the tremendous processing and storage capabilities of modern hardware through the development and implementation of an efficient software system. Over the years, the software budget of many organisations has continued to increase and even surpass, the hardware budget while operation and maintenance costs amount to substantially more than half of the entire cost of system development, operations and maintenance [Vic, 1983].

Owing to the increasing importance of software in the success of an organization, software involves art, science and engineering technology. There is now a field of study known as Software Engineering which is aimed at providing sound engineering principles, practice and tools in support of all the phases of software life cycle.

In order to ensure the success of a software development project, there is the need for effective and efficient monitoring. Monitoring helps to discover any shortcoming or flaw in the execution of a project. Monitoring allows corrective measures to be taken while a project is going on.

An important component of the software development process is *ex post* evaluation of its development and implementation. Software evaluation is a form of audit to assess the

performance or success of a software development project. The result of the *ex post* evaluation could be useful in identifying as well as implementing new projects in similar or other applications.

1.2 Motivation for the Project

Software costs today, represents a significant percentage of a computer-based system. In some organisations, it is the most expensive element in computing system. Consequently, a large cost estimation error can make a difference between profit and loss. This calls for a proper monitoring and evaluation of software development projects.

Unfortunately, monitoring and evaluation of software development projects are fraught with a lot of problems, chief among which is the nature of software development projects. Unlike hardware development projects where most elements of cost, benefits and returns are tangible - can be measured directly, most of the cost, benefits and returns to a software development project are intangible, that is, cannot be measured directly.

Many variables - human, technical, environmental, political, and so on, could affect the ultimate cost of a software system and the effort applied to develop it. The opinion is expressed in [Pressman, 1987] that software cost and effort estimation will never be an exact science. That opinion stems from the fact that estimation of resources, cost and schedule of software development effort requires experience, access to good historical information, and courage to commit to quantitative measures when qualitative data are all that exists. This scenario probably, accounts for the reason why there has been little efforts at standardizing the monitoring and evaluation of software development projects. Even where there is monitoring and evaluation, they are somehow based on the experience and sometimes on the convenience of the systems analyst rather than a formal procedure of monitoring and evaluation. Sometimes, different systems analysts disagree on the metrics to be adopted in monitoring and evaluation of software development projects.

In order to find solutions to these problems, the development of a Computer Aided System is been proposed for the monitoring and evaluation of software development, operations and maintenance. A human expert is somebody who possesses knowledge in the form of facts and rules as well as experience in a specific domain. Experience are not normally found in the literature rather it is acquired through many years of practice. Experience consists of heuristic, analogies and judgements. Knowledge and experience enables the human expert to choose, based on strategies and if one strategy fails, he could go back and try another alternative. An Expert System is basically, a computer-based system which possesses a set of facts about a specific domain of human expertise. An Expert System uses rules of inference to draw conclusion or make decisions within a defined domain. They are built to be able to explain why they reject certain paths of reasoning and choose others.

1.3 Objectives of the Research

The objective of the research work are as follows:

a. To provide a user friendly, menu-driven, and iteractive environment for data entry, update, processing and retrieval of data/information relating to the monitoring and evaluation of software development, operations and maintenance.

b. To develop a computer aided system for monitoring and evaluation of software development, operations and maintenance.

c. To implement and carry out the case study of the computer aided system.

1.4 Research Methodology

In order to carry out this research, the following steps were taken:

a. Review of related literature on software metrics and software project cost estimation.

b. Review of related literature on computer aided system.

c. Study of the software life cycle with a view of getting a proper understanding of the

processes involved in the development, operations and maintenance of a software project.

d. Design of computer aided system for monitoring and evaluation of software development, operations and maintenance.

e. Study of Paradox Relational Database Management System, which is adopted for the practical implementation of the computer aided system.

f. Case study of the Federal University of Technology, Akure (FUTA) Payroll System.

1.5 Organisation of Thesis

Chapter two presents the review of expert system. Chapter three reviews related literature on software metrics and software project cost estimation. In chapter four, the software life cycle was discussed. In chapter five the framework for the computer aided system for monitoring and evaluation of software development, operations and maintenance is presented. In chapter six the implementation of the framework is carried out using Federal University of Technology, Akure (FUTA) payroll system. Conclusions and recommendations for further studies are presented in chapter seven.

CHAPTER TWO

REVIEW OF EXPERT SYSTEM

2.1 Introduction

Artificial Intelligence (AI) is the part of computer science concerned with the design of intelligent computer systems, that is, systems that exhibits the characteristics we associate with human behaviour such as understanding language, learning, reasoning and solving problems. It is concerned with programming computers to perform tasks that are presently done by human beings because AI involves such higher mental processes as perceptual learning, memory organization and judgement reasoning. Thus, writing a program to perform complicated statistical calculations would not be seen as an AI activity, while writing a program to design experiments and test hypotheses is an AI activity.

AI is defined as the computer science discipline that is aimed at understanding the nature of human intelligence through the construction of computer programs that imitate intelligent behaviour. In [Gallaire, 1985], it is observed that the field of Artificial Intelligence has been the object of controversial discussions. There have been arguments over the name itself, contents of the field, and results of the field.

Concerning the subject matter of Artificial Intelligence, [Akinyokun and Adeniyi, 1991], observed that, AI deals with the simulation of human behaviour: the discovery of techniques that will allow us to design and program machines which can both emulate and extend our mental capabilities. The discipline is therefore related to a wide range of other academic subjects areas such as computer science, psychology, philosophy, linguistics and engineering.

The controversy and pessimism that greeted earlier works on Artificial Intelligence is giving way to concerted effort aimed at harnessing the gains of AI for the advancement of science, technology and industry. This evolution was due to three events:

a. The breakthroughs in the knowledge engineering component of AI.

b. Advancement in hardware design and implementation thereby making it more efficient and cheaper. This, in turn, makes it feasible to implement ideas relying on more powerful or cheaper technology, and which opens a wide potential market to new and unsophisticated users requiring new ergonomic, behavioural attitudes to which traditional computers cannot give a correct response.

c. The need for industry and society to invent solutions to increasing productivity problems and to satisfy more ambitious social needs [Gallaire, 1985].

Artificial Intelligence centres around problem-solving. It does this by knowledge representation: how to represent what we know about the problem domain, how to extract precisely what is needed for solving the problem. Hence, an intelligent system, basically comprises a perception system, a memory system and a processing system.

The earliest work on AI focussed on the construction of generalized purpose intelligent systems such as game playing, theorem proving and puzzle solving. The emphasis was on powerful inference methods that could function efficiently even when the available domain - specific knowledge was relatively meagre. Today, the emphasis is on the roles of specific and detailed knowledge, rather than on generalized knowledge [Akinyokun and Adeniyi, 1991].

2.2 Basic Concept of Expert System

An Expert System (ES) or Intelligent Knowledge Based Systems (IKBS) is a specialized computer package which can perform the function of a human expert. They have some characteristics which we normally associate with human intelligence. Expert Systems contain the knowledge of human experts in a specific domain along with the reasoning rules the human experts employ to manipulate their knowledge and arrive at conclusions.

An expert system is described as a knowledge-based system which efficiently solve real world problem based on the expertise contained in their knowledge bases elicited from domain

experts. An expert system is considered as a part of Artificial Intelligence that has received the most publicity and as the name implies, these systems contain the knowledge of human expert in specific field, along with the reasoning rules the human experts employ to manipulate their knowledge and arrive at conclusions. From the definition given above we can consider expert systems as a set of computer based programs which store human experts factual and inferential knowledge and use heuristics to draw conclusions from a given case data.

Expert Systems are very useful in the combinatorial problems where straight forward methods of enumeration tends to explosive number of possibilities [Akinyokun and Adeniyi, 1991]. ES deals with combinatorial problems by means of stepwise elimination of the possibilities that are unlikely to prove fruitful. The conventional file system and computer databases, though capable of storing facts and numerical data, allow users to easily retrieve information stored in them, but cannot take decisions about its use and interpretation to the user. Except for a few cases, such systems are unable to answer any question for which there is no stored or derivable answer.

In the disciplines such as medicine, law, management, agriculture, finance, personnel management and so on, the conventional file system and database systems, process a lot of unstructured information which must be put into consideration in decision making. Such unstructured information may not lend itself to mathematical, statistical or routine data processing. Hence, decision making in these areas are still heavily dependent on human experts who possesses the skills for identifying and rating key factors, weighing evidence, evaluating alternatives, predicting outcomes and making complex decisions.

The knowledge of human experts can be built into computer expert system using two methods:

a. Extracting all the necessary and relevant information; and coding it. The system is then built around the coded information. The data in such system is fixed. Building such a system requires a great deal of expertise and work. Once the system is completed it

normally works consistently within the framework of the field of application.

b. Building a general purpose expert system that can handle a large range of information types. Such system, learn all about the area of application. The learning process is by interrogating the expert who will use the system by using the techniques of the knowledge engineer and learning by example (induction).

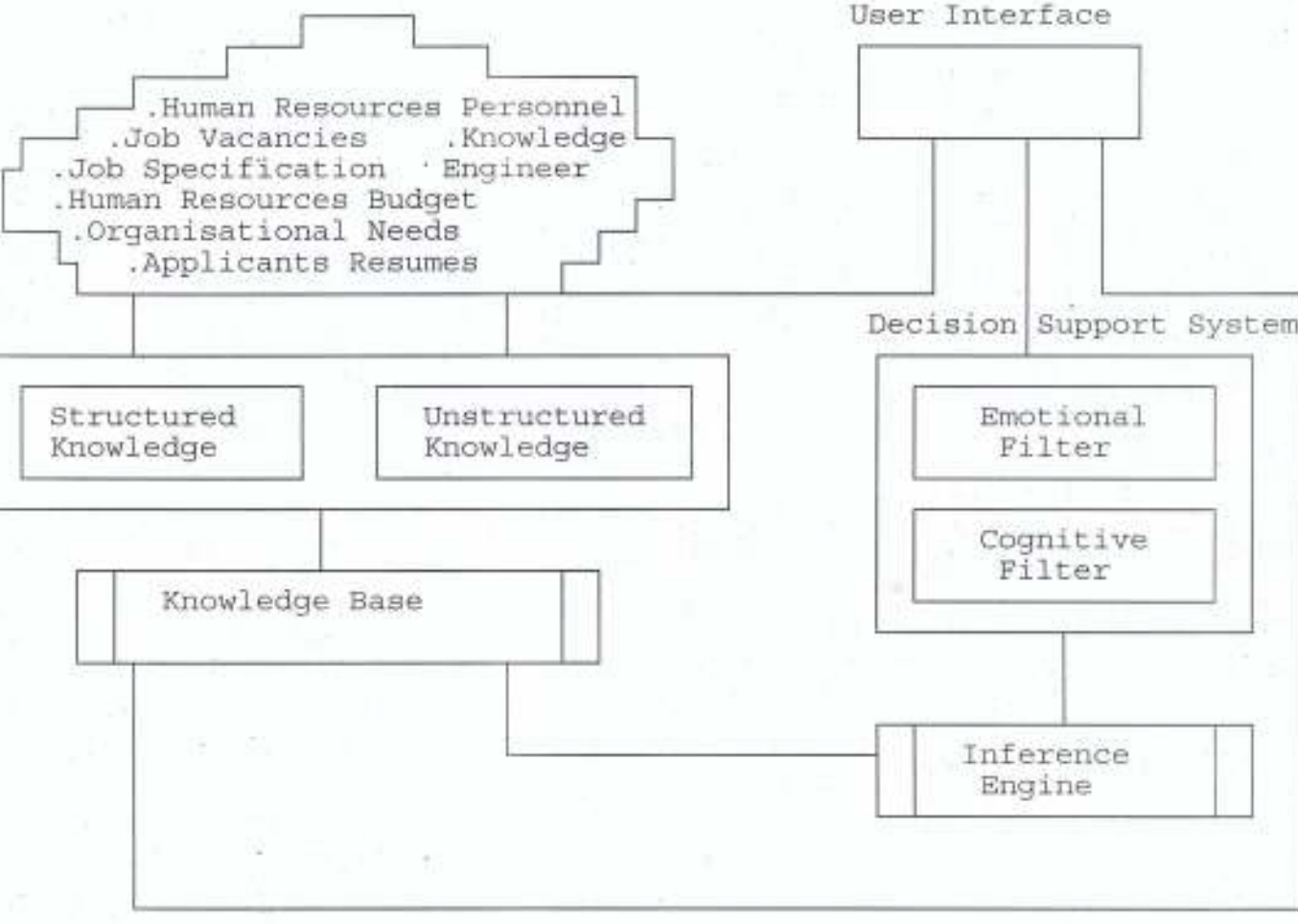
An Expert System is made up of the following components or features:

a. Knowledge base

b. Inference engine

c. Decision support system

The conceptual diagram of expert system for an information technology based personnel procurement is shown in Figure 2.1.



igure 2.1: Conceptual Diagram of Expert System

2.2.1 Knowledge Base

Any expert system must posses some form of knowledge base on which inferences are drawn and decisions taken. There are two types of knowledge:

i. Structured knowledge

ii. Unstructured knowledge

The structured knowledge can be described as the knowledge which includes factual data in a given problem domain as found in text books, technical reports, conference proceedings, journals and so on. They are made up of the laws, facts and decision rules that represent general knowledge of the given subject. An example of a decision rule is:

If CONDITION then DECISION with confidence α.

The CONDITION stands for a list of elementary conditions characterizing a situation or an object to which the rule is applied. The DECISION stands for specific advice or action, which the rule indicates when the CONDITION is satisfied. The parameter α expresses the strength of confidence in the DECISION when the CONDITION is fully satisfied ($0 \leq \alpha \leq 1$).

When $\alpha = 1$, the confidence is maximal. When $\alpha = 0.5$, the rule states that the DECISION may be correct only half of the time. Thus, the rule format permits one to express the conditional knowledge of experts, and also the experts confidence or lack of confidence in this knowledge.

In general, the CONDITION part of the rule may be only partially satisfied. For example, the condition:

temperature = above normal, is only partially true if the temperature was just above average. The degree to which the condition is true is expressed by a parameter $\gamma(0 \leq \gamma \leq 1)$.

If $\gamma < 1$, then the confidence in the DECISION is calculated as a function of both α and γ. The way this function is calculated depends on the so-called evaluation scheme [Akinyokun and Adeniyi, 1991].

The DECISION part of a rule in a knowledge base may be an assignment of the status "TRUE" to some conditions which are in the CONDITION part of another rule. Hence, a satisfaction of one rule may cause a satisfaction of another rule(s) and in this fashion the system can perform a chain of inferences.

Unstructured knowledge comprises experiential knowledge, knowledge of good judgement or good practice in the problem domain. It is also made up of the vague knowledge and the art of guessing which a human expert acquires over the years.

For ES to solve a problem at high level of human expertise, it must have both structured and unstructured of knowledge in its knowledge base. These knowledge is broken into their atomic components as follows:

a. Behaviour description

b. Uncertain facts

c. Vocabulary definition

d. Objects and relationships

e. Decision rules

f. Knowledge base

g. Hypotheses

h. Processes

i. Constraints

j. Unstructured knowledge

k. Disjunctive facts

2.2.2 Inference Engine

One of the unfolding development in the computing science is the importance of information derived from data and the knowledge that can be inferred from data. Knowledge, which forms

a basis for reasoning is not sufficient in itself to discover and use lines of reasoning. It is the inference engine that pieces together a suitable line of reasoning leading to a solution to the problem or the building of a body of consultative advice. This could be accomplished through forward chaining or backward chaining or both [Akinyokun, 1988].

Forward chaining involves starting from a set of available or given information and reasoning forward in order to try to arrive at a conclusion. For example, a plant pathologist may use a set of signs and symptoms observed to come to a set of conclusion about a particular plant disease.

In backward chaining, the plant pathologist having suspected the occurrence of a particular plant disease may use the available statistic to justify the conclusion. The choice of the chaining method to adopt depends on the nature of the problem. While some requires either of forward cahining or backward chaining, others require a combination of the two.

The forward chaining technique can be classified into the parallel approach and serial approach. The parallel approach involves gathering all the relevant information or evidence and then use all the evidence as a whole to come to a conclusion or take a decision.

The serial approach normally requests for one piece of information at a time and uses it to come to a conclusion. This approach has the advantage of not bordering the user about unwieldy information, some of which may not be necessary to the solution of the problem at hand. For instance, if the process of drawing a conclusion about a given problem involves "yes or no" answer, the parallel approach requires a number of "yes or no" comparisons before a decision is made whereas, in serial approach such comparison are made one at a time.

If the learning process involves asking questions whose answers are "yes/no". It means there are only two routes the program can take at any given point in the interrogation. The reasoning pattern can be represented in binary form while the information and the logic path could be represented in form of a "binary tree".

2.2.3 Decision Support System

The decision support system assist in decision making by subjecting the output of the inference engine to inductive and deductive reasoning. The decision support system has two sub-systems, namely: cognitive filter and emotional filter. For instance, in information technology based personnel procurement, [Akinyokun and Uzoka, 1999], used the inference engine in matching the decision variables of the personnel knowledge with the corresponding decision variables of the job requirements knowledge and reports a list of applicants that are appointable for some specific jobs. The cognitive filter and the emotional filter, carries out the inductive and deductive reasoning on the information contents of the list of applicants appointable for a given job produced by the inference engine. The following could form the basis for cognitive filtering of the list of appointable applicants:

a. The decision on whether, say, Institute of Chatered Accountants of Nigeria (ICAN) qualification is more suitable for a job, based on the human resources personnel's judgement of syllabi of the professional body.

b. The decision to employ mostly, people of a particular age bracket based on the nature of the job.

c. The decision to employ a staff who currently works in a similar company, and in the required position.

d. A company may have a bus service for its staff. Based on this, members of staff may be clustered within a particular location to make the bus system effective. Among the new applicants, who are found appointable, there are some who live in the same neighbourhood as the members of staff. Such an applicant may be employed instead of an applicant who lives far away from the neighbourhood because of the inconveniences and extra cost of transportation.

e. The decision to employ somebody who had a second class (lower division) in place of

somebody who had first class because of the former's past experience on the job, despite the fact that both of them had the same aggregate score.

Similarly, the following could form the basis for emotional filtering of the list of appointable applicants:

a. The decision to employ an applicants because he is related to a senior member of staff of the organisation and his referees are credible personalities in the society.

b. The decision to employ somebody because of his state of origin, based on the feelings of the human resources personnel, that applicants from that state have drive and determination.

c. The decision to drop a candidate because he was dismissed from his former place of work, even when he makes a good aggregate score in the matching process.

d. A job may not have sex restriction, but because of the stress involved in the job, the human resource personnel may feel that a male would do the job better than a female; as such, employs a male, rather than a female.

e. An applicant may have indicated that he is a sports man. If the company in question has a sports club, the tendency is there that the applicant may be employed based on the feeling that he would make good contributions to the company's sports programme.

f. From the medical records of an applicant and his or her physical appearance, he or she may be found disabled in a way. But because of the very high intelligent qoutient of the applicant, he may be employed and deployed to an office where his or her physical contact with cilents or customers could be very remote.

2.3 Knowledge Acquisition

Acquisition of necessary knowledge about the problem domain from the human expert by the knowledge engineer is the first step in building an expert system. Despite the fact that numerous

Expert Systems (ESs) have been constructed, an effective standardised method of acquiring the expert knowledge is still more of an art than science. The process is currently characterised by an expert at one end who is unfamiliar with ES and unable to articulate what knowledge he has and how to use it to solve problems. At the other end, there is the knowledge engineer who may be ignorant of expertise domain but can construct an adequate model of the human expertise of a domain. The knowledge of human experts of a problem domain can be acquired in many ways. Some of the ways are listed below:

a. Conducting interviews with the human expert in the domain.

b. Studying the past and present problems and their solution.

c. Identifying prototypical or idealized problem types and associated solutions.

d. Carrying out literature research.

e. Systematic observation and collection of verbal protocols from the human expert at work in a real environment solving actual,as opposed to hypothetical problems.

In practice, the combinations of two or more of these methods of knowledge acquisition may be taken.

2.4 Knowledge Representation

Knowledge can be represented in many ways. Among the standardized ways are:

a. Production rules

b. Predicate logic

c. Frame

2.4.1 Production Rules

A production system consists of rule set (production memory), a rule interpreter that decides how and when to apply which rule, and a working memory that can hold data, goal and

intermediate results. The basic cycle of a production rule system consists of a select phase and an execute phase. During the execute phase, the system interprets the selected rule to draw inferences that alter the system's dynamic memory. The working memory includes components for long-term static data and short-term dynamic data. The long-term store, which is the knowledge base, contains rules and facts.

Rules specify actions the system should initiate when certain triggering conditions occur. The conditions define important patterns of data that can arise in the working memory. The system presents data in terms of relations, propositions or equivalent logical expressions. Facts define static, true propositions.

In contrast to conventional data processing systems, most production rule systems distribute their logic over numerous independent condition-action rules, monitor dynamic results for triggering patterns of data, determine their sequential behaviour by selecting their next activity from a set of candidate-triggered rules, and store their intermediate results exclusively in a global working memory. The basic features of production rule system is shown in Figure 2.2

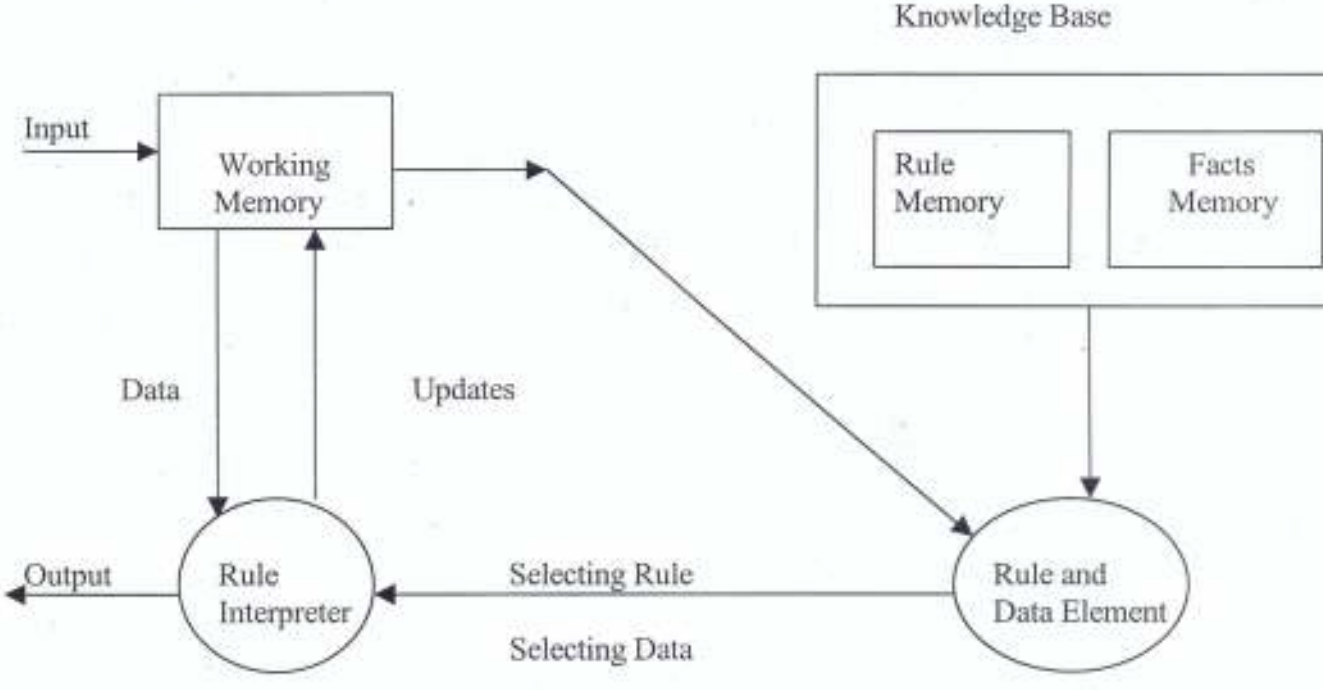


Figure 2.2 - Basic Features of Production Rule System

A production rule is generally expressed as:

If P_1 & P_2 &.....& P_n

Then Q_1 & Q_2 &.....& Q_m

which reads as follows:

If the premise P_1 through P_n are true then perform actions Q_1 through Q_m where P_i(i = 1, 2, ... n) are the conditions and Q_j(j = 1, 2,.....m), the conclusions. The conditions are usually object-attribute-value triples.

Typical production rules are given as follows:

a. IF (student sex is male) &

(student apply for campus accommodation)

THEN (allocate room for him in one of the male halls)

b. IF (student mark not found) &

(student attend exam) &

(student sign attendance sheet) &

(student submitted answer sheet)

THEN (course lecturer suspect)

2.4.2 Frames

Frames can be described as data structures, which provide the mechanism for grouping information in terms of a record of 'slots' and filters. This record can be thought of as a complex node in a semantic network, with a special slot filled by the name of the object that the node is for, and the other slots being filled with the value of the various common attributes associated with such an object.

Frame systems attempt to model classes of object which are related to one another using data abstraction mechanism, such as classification, generalization, aggregation and association [Akinyokun *et al*, 1985]. When these data abstraction mechanisms are applied to a universe of discourse, hierarchies of objects are formed. A network of objects is obtained when two or more

data abstraction mechanisms are integrated in the conceptualization of a universe of discourse.

The conceptual diagram of the aggregation, generalization and association of objects are presented in Figure 2.3a, and Figure 2.3b respectively. The combination of aggregation and generalization of objects is depicted in Figure 2.3c.

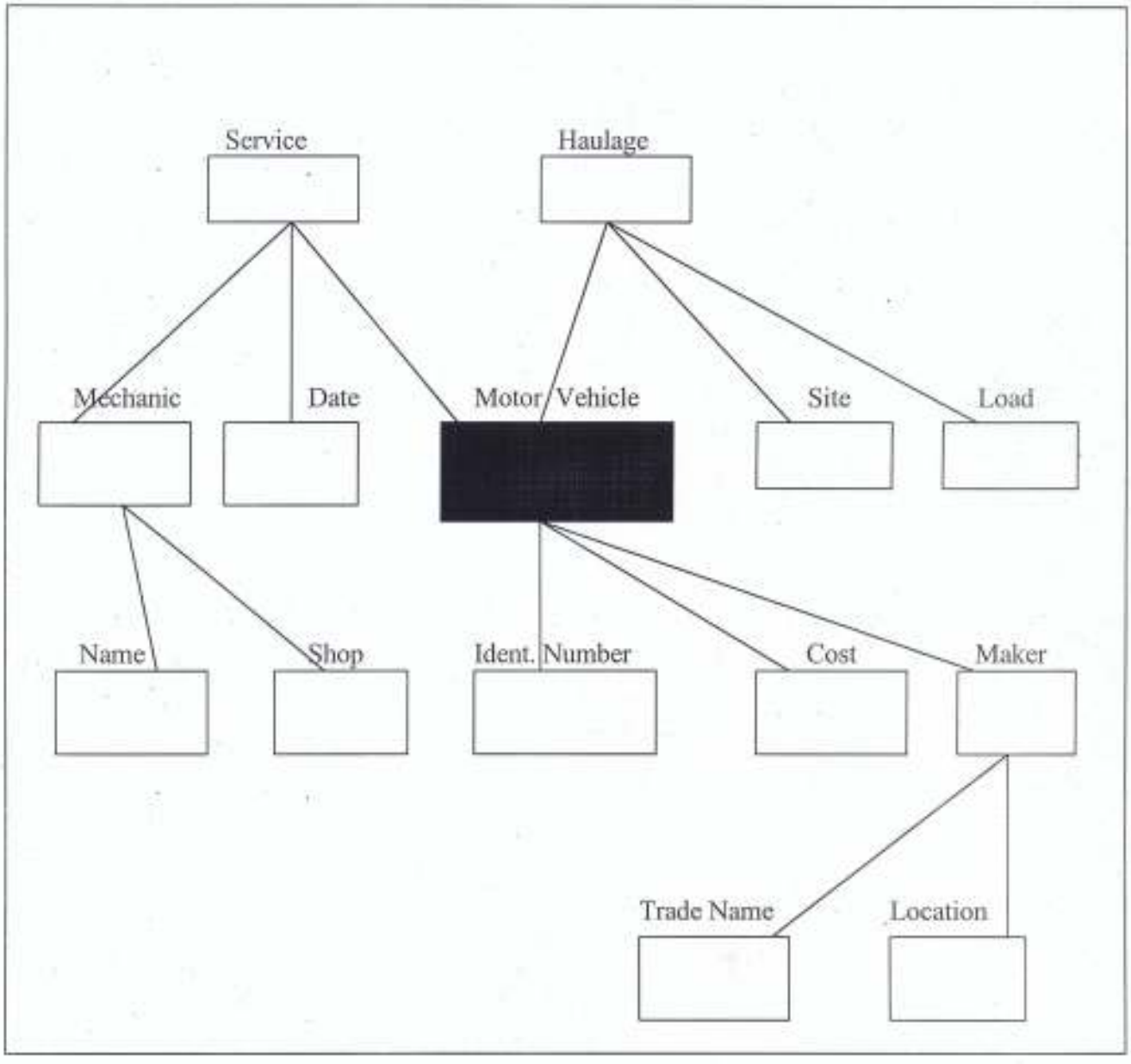


Figure 2.3a: Aggregation Hierarchy of Vehicle

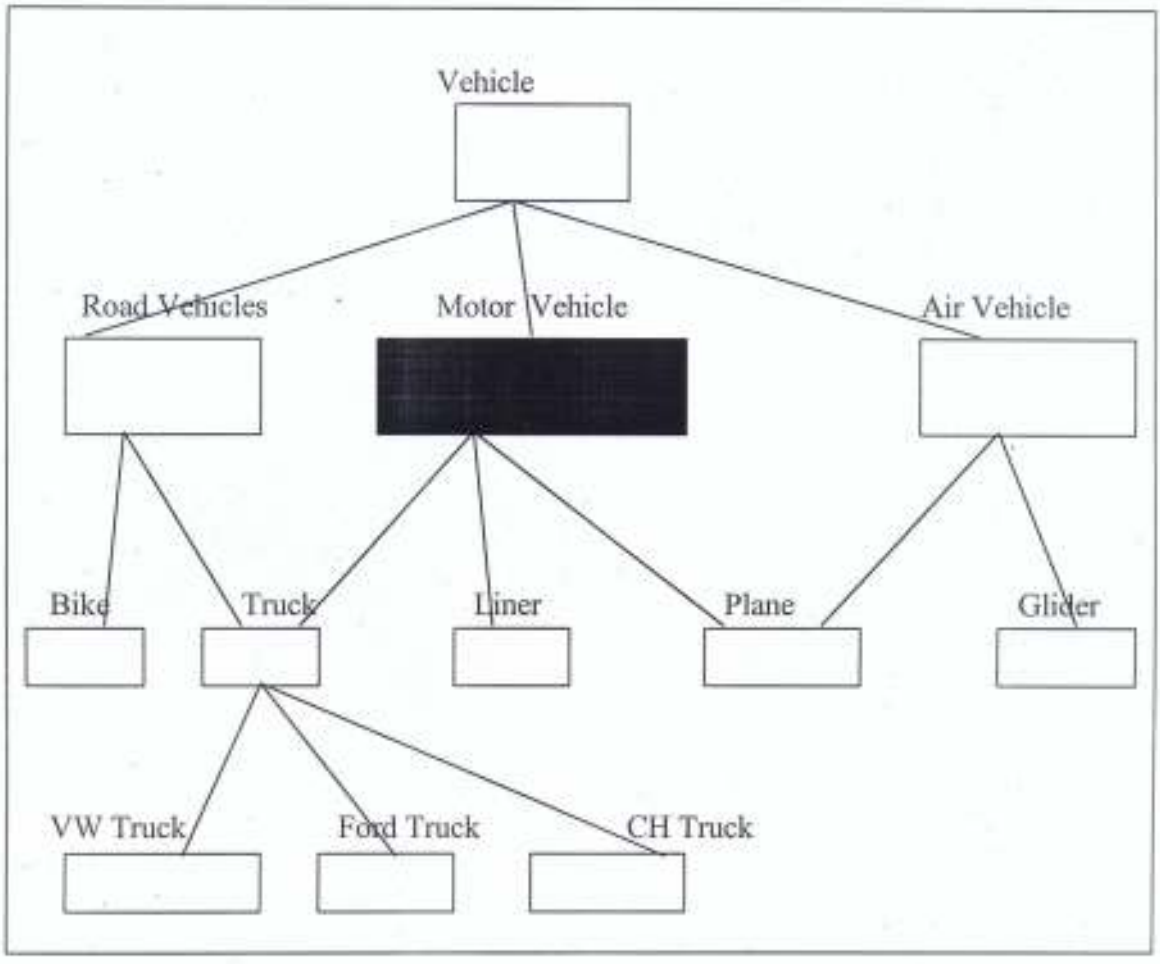


Figure 2.3b: Generalization Hierarchy of Vehicles

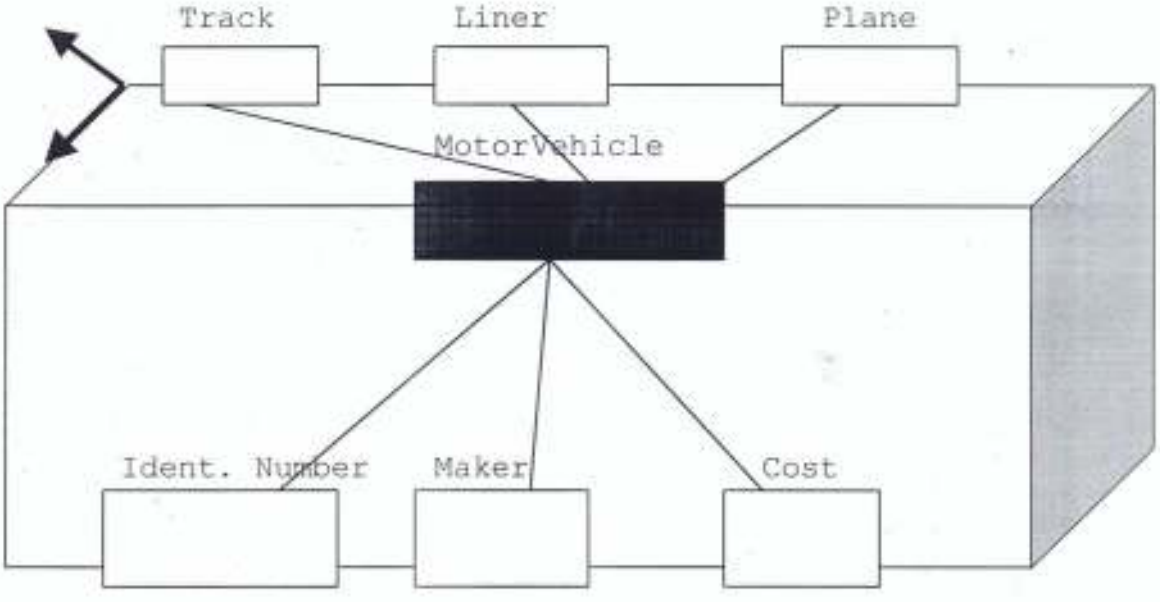


Figure 2.3c: Generalization and Aggregation Hierarchy

2.4.3 Predicate logic

Predicate logic is used as an analyzing tool to represent facts, general statements, vague statements and complex relationships. The use of logic for knowledge representation involves inference. However, logic does not tell one which inferences should be drawn at any given point in the search for a solution; it only tells which inferences one is entitled to draw according to some rules. In automatic theorem proving, logic programming and knowledge representation, attempt is made to normalize predicate logic. The three main syntactic normalization schemes employed are Conjunctive Normal Form (CNF), Full Clausal Form and Horn Clause Subset. A logic program is an arbitrary set of expressions known as clauses. A clause is an expression of the form:

$$Q_1, \ldots\ldots\ldots Q_n \quad :- \quad P_1, \ldots\ldots P_m$$

A sentence of this form says that one of the Q_i must be true if all of the P_i are true.

The expressions to the left and right of the :- operator in a clause must be atoms, that is, expressions of the form:

$$R(T_1, \ldots\ldots T_k)$$

where R is a relation constant and where each T_i is a term. A term is either an object constant, a variable or an expression of the form:

$$F(T_1, \ldots\ldots T_n)$$

where F is a function constant and each T_i is a term.

The schematic diagram of a logic programming system is shown in Figure 2.5. An application-independent inference procedure is at the centre of any logic programming system. When interrogated by the user, the inference procedure replies after drawing conclusions from the facts in the knowledge base. In some instances, the conclusions drawn are stored in the knowledge base for later use.

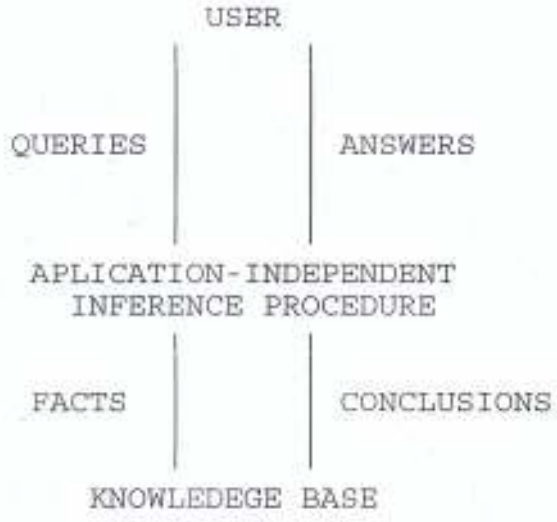


Figure 2.5 – Logic Programming System

2.4.4 Concluding Remarks

The PROLOG programming language gives the knowledge engineer facilities to quantify actions and the pattern matching which are generally more powerful than those afforded by production rule and semantic network interpreters. The fundamental theorem proving method that PROLOG relied upon is called resolution and refutation. It is remarked however that the theorem proving with full unification inherent in logic programming is computationally expensive in terms of both computer memory and central processing unit. For this reason, it has been thought that the resolution and refutation mechanisms were impossible as a conceptual device for reasoning about problems of realistic complexity. Thus the logic base approach is perhaps less tried and tested as a methodology for the construction of a large scale real life system. In practice, existing and popular ESs adopt the combination of production rules and semantic network in an attempt to represent knowledge.

CHAPTER THREE

REVIEW OF SOFTWARE METRICS AND PROJECT COST

3.1 Introduction

A common feature to both monitoring and evaluation of projects whether software or hardware, is that they make use of data relating to costs, benefits, man-time spent, activities completed, materials and machine-time consumed and so on, to determine the extent of progress or success of a project at any point in time. Estimation of cost, efforts and benefits of a software project is fraught with many difficulties due to the subjective nature of the variables involved in the determination of cost, efforts and benefits.

In this chapter, we review some existing and popular approaches to the estimation of software metrics such as cost, efforts and benefits. Software metrics is indeed, a broad range of measures for computer software. In [Pressman, 1987], the following reasons are given for software measurement:

a. To indicate the quality of the software

b. To assess the productivity of the people who produce the software

c. To assess the benefits (in terms of productivity and quality) derived from new software engineering methods and tools.

d. To form a baseline for estimation.

e. To help justify requests for new tools or additional training.

In [Pressman, 1987], the software metrics are classified into the following two categories namely, direct measures and indirect measures. The direct measures include: cost, effort, lines of code, speed, memory size and error rate while the indirect measures include: function, quality, complexity, efficiency, reliability and maintainability.

The direct measures of software are relatively easy to collect as long as specific conventions for measurement are established in advance while indirect measures such as quality,

complexity, functionality, efficiency and maintainability are difficult to assess.

3.2 Size Orientated Metrics

Size-orientated software metrics involve the direct measures of software and the process by which the software is developed. The basic requirement is that an organisation should maintain simple records of size-orientated data. Such data include: number of lines of code, number of person-months used for a particular software development project, cost, number of pages of documentation, number of errors encountered per year, number of people involved in the development and so on.

From these rudimentary data, a set of simple size-orientated productivity and quality metrics can be developed for each project. Averages can be computed for all projects for example we can have the following, among others:

$$\text{Productivity} = \frac{\text{KLOC}}{\text{person-month}}$$

$$\text{Quality} = \frac{\text{error}}{\text{KLOC}}$$

$$\text{Cost} = \frac{\text{Amount of money expended}}{\text{KLOC}}$$

$$\text{Documentation} = \frac{\text{Pages of documentation}}{\text{KLOC}}$$

where one KLOC is one thousand lines of code

In mid 1950's, programmers, who were then very few in number, began to measure software effort in terms of number of "lines of code" (LOC). This was used as the basis for predicting programming costs, testing costs, salaries and so on [Biezer, 1983]. A study by [Weiwurm *et al*, 1965], debunked the use of such simplistic measure. It is observed that cost is largely influenced by such things like number of meetings attended, the number of records in the database, the relative state of hardware (if new hardware), documentation requirements and

other factors that did not relate directly to number of lines of code. The work of [Weiwurm *et al*, 1965] and [Nelson, 1967] are some of the first landmark efforts at cost estimation of software development, as they provide the basis for cost estimation.

In the estimation of number of bugs in a software using number of lines of code, a study carried out by [Thayer, 1976], showed that there is no simple non-linear law that relates number of bugs per line to the number of lines of code. The same lack of correlation was shown by [Rubey, 1975; Curtis,1979].

In another study by [Lipow, 1982], in which 115,000 statements are examined, it reports a non-linear relationship between bug rate per lines of code and program size. Other factors such as language type and modularization, executable lines of code and not data declarations are considered as part of software metrics.

Size-orientated metrics are controversial and are not universally accepted as the best way of measuring the process of software development. It is claimed that LOC measures are programming language - dependent, they penalize well-designed but shorter programs, cannot easily accommodate non-procedural languages, and that their use in estimation requires a level of detail that may be difficult to achieve.

3.3 Function Orientated Metrics

Function-orientated metrics are indirect measures of software and the process by which it is developed. Instead of counting lines of code, function-orientated metrics focus on program "functionality" or "utility". Function-orientated metrics were first proposed by [Albrecht, 1979], which suggests a productivity measurement approach called the "function point" method. Function points are derived using and empirical relationship based on countable measures of software's information domain and subjective assessments of software complexity.

Function point measure is originally designed to be applied to business information

system applications. Five information domain characteristics are determined and counts are provided. The information domain covered are the following:

a. Number of user inputs: This refers to the user inputs that provides distinct application-orientated data to the software.

b. Number of user outputs: Each user output that provides application orientated information to the user. Output in this context, refers to reports, screens, error messages and so on.

c. Number of user enquiries: This refers to online input that results in the generation of some immediate software responses in the form of an online output reports.

d. Number of files: This refers to the logical master file, that is, a logical grouping of data that may be one part of a large database or separate file.

e. Number of external interfaces: All machine-readable interfaces, for example, data files on tape or disk, that are used to transmit information to another system.

Once data on the above information domain is collected, a complexity value is associated with each count. Organisations that uses function point methods develop criteria for determining whether a particular entry is simple, average or complex. This sort of determination is somewhat subjective in nature.

Function-points (FP) is computed as follows:

$$FP = \text{Count total} \times [0.65 + 0.01 \times \text{SUM}(F_i)]$$

where count total is the sum of all counts and F_i, $i = 1$ to 14. F_i are known as "complexity adjustment values" based on responses to some questions derived in [Albrecht, 1985]. Such questions bothers on the degree of modularity, conciseness, simplicity, generality, consistency and so on.

Once the function-points have been calculated, they are used in a manner analogous to LOC, to measure software productivity, quality and other attributes. For example, we have

$$\text{Productivity} = \frac{\text{FP}}{\text{Person-month}}$$

$$\text{Quality} = \frac{\text{Error}}{\text{FP}}$$

$$\text{Cost} = \frac{\text{Amount}}{\text{FP}}$$

$$\text{Documentation} = \frac{\text{Number of Pages}}{\text{FP}}$$

Like LOC, function-point metric is controversial. Although, it is programming language independent, making it ideal for applications using conventional and non-procedural languages, however, it is claimed that the computation is based on subjective data, rather than objective data.

Both size-orientated and function-orientated metrics have their strong and weak points. They become more useful when averages of LOC and FP estimates are made over a period of time. Such data can be used for software project planning.

Challenging the usefulness of LOC and FP data, many researchers were of the opinion that many factors influence productivity. For instance, [Basili and Zelkowitz, 1978], identified the following five factors that influence productivity:

a. People factors: the size and expertise of the software development team.

b. Problem factors: the complexity of the problem to be solved and the number of changes in design constraints or requirements.

c. Process factors: analysis and design techniques that are used, languages available, and review procedures.

d. Product factors: reliability and performance of the computer-based system.

e. Resource factors: availability of development tools, hardware, and software resources.

Although function points and LOC have been found to be relatively accurate predictors of software development effort and cost, it becomes more useful when a historical baseline of information is established. This leads to the question of the appropriate methods for metrics data collection.

3.4 McCabes's Complexity Metric

In [McCabe, 1976], a complexity metric is defined as a count of the number of decision statements in a program. In practice, it is a count of the decisions when considering the complexity of an assembly of several modules. Complexity in this context is additive. Thus, if unit "A" has a complexity of seven, that is, six decisions and unit "B" has a complexity of nine, that is, eight decisions, their sum has a complexity of sixteen. Hence, multi-way branches such as ON....GOTO or CASE statements are reckoned at one less than the number of possible branches. For instance, a ten-way CASE statement is counted as equivalent of nine decisions.

The principal virtue of McCabe's metric is its simplicity. It achieves significantly better correlation of complexity to bugs and debugging difficulty than lines-of-code does. Most of the limitations of the Mccabe's metric derive from some of the underlying assumptions of the metric. These assumptions are:

a. Bugs are proportional to control-flow complexity, that is, processing complexity, database structure, typographical and wild card bugs are immaterial.

b. It does not distinguish between different kinds of control-flow complexity, for example, a simple "IF - THEN-ELSE" statement is judged at the same complexity as a loop.

c. A thousand straight-line instructions are judged to be as complex as a single instruction.

Nevertheless, Mccabe's metric is an easy to use and useful rule of thumb for comparing alternative approaches and for estimating debug labour.

3.5 Halstead's Metrics

Halstead's metrics [Halstead, 1977] are based on a combination of arguments derived from

common sense, information theory; and psychology. The set of metrics are based on two parameters of program:

n_1 = the number of distinct operators (keywords) in the program.

n_2 = the number of distinct operands (database objects) in the program.

From this, Halstead defined program length as:

$H = n_1 \log_2 n_2 + n_2 \log_2 n_2$..... equation (1)

Paired operators such as "BEGIN.....END", "DO.....UNTIL", "FOR.....NEXT" are treated as a single operator. For any given program, it is possible to determine the actual operator and operand appearances.

N_1 = program operator count

N_2 = program operand count

The actual Halstead Length is

$N = N_1 + N_2$.....equation (2)

Halstead also defines program's vocabulary as the sum of the number of distinct operators and operands. That is, n = vocabulary = $n_1 + n_2$ .....equation (3)

Equation 1, assumes that program actual Halstead Length (N) can be calculated from the program's vocabulary, even though the program has not been written. Hence; if a program is written using 20 keyword out of a total of 200 in the language, and it references 30 objects in the database, its Halstead Length should be $20 \log_2 20 + 30 \log_2 30 = 233.6$.

Also, the relation appears to hold when a program is subdivided into modules. The bug prediction 'B' formula, is based on the four values, n_1, n_2, N_1 and N_2 and is defined by:

$B = (N_1 + N_2) \log_2(n_1 + n_2)/3000$ ..... equation (4).

Hence, a program which accesses 75 database items a total of 1300 times, and which uses 150 operators a total of 1200 times should have its bug prediction calculated as:

$(1300+1200)\log_2(75+150)/3000 = 6.5$ bugs.

In addition, Halstead also derives relations that predict programming effort and time which seems to correlate with experience. The time prediction is non-linear, showing the following:

a. 24 hours for a 120 - instruction program

b. 230 hours for a 1000 - instruction program

c. 1023 hours for a 2000 - instruction program.

Halstead metrics has been reviewed by many other researchers. For instance, [Lipow, 1982] confirms the suitability of the bug prediction equation. It compares actual bug counts to predicted bug counts to within 8% over a program sizes ranging from 300 to 12,000 executable statements. However, the analysis was of post - compilation bugs, so that syntax errors caught by the compiler are excluded.

In [Curtis, 1979], it is shown that Halstead's metrics are at least twice as good as lines - of - code and are not improved by Mccabe's metric. In [Boris, 1984], while judging from numerous experimental confirmation of Halstead's metrics, it is declared that "if ever a true software science emerges, it will be based in no small part on Halstead's work". However, it also observes the following weaknesses in Halstead's work:

a. Halstead does not distinguish between a programmer's "own" sub-functions and a sub-function provided by another programmer.

b. Non - executable statements such as declarations and data statements are often ignored. Hence, most initialization and data structure bugs are ignored in the computations of Halstead's metrics.

c. It assumes that code is code and data is data, and that the two can never meet. Hence, if Halstead's metrics are rigidly applied to an assembly language program that does extensive modification of its own code, it will predict the same bug count as a routine that performs the same operations on a fixed data area.

d. Problem of data type distinctions arises in languages which permit mixed operations, for

example, integer - floating point addition and which have no user - defined types.

e. It does not take cognisance of call depth, for example, a routine that calls ten different sub - routines as sequence of successive calls would actually be considered more complex than a routine that had ten nested calls.

f. The problem of operator types, for example, IF....THEN....ELSE statement is given the same weight as a FOR....UNTIL statement, even though it is known that loops are far more troublesome.

3.6 Measures of Program Quality

In order to ensure thorough monitoring and evaluation of a software development project there is the need to quantify the quality of a program. Quality tests and measures provide a way of quantifying quality judgements. Several tests and metrics have been proposed to measure objectively, the degree to which certain quality characteristics are present in a software system. No claim can be made as to their completeness or universal applicability.

It is remarked in [James *et al,* 1983], that no single test or metric has been developed thus far to measure overall software quality since many individual quality characteristics are in conflict with one another. It is doubtful whether anybody has since come out with a universally acceptable metric or test of software quality.

Another issue is the fact that it may be difficult to build software quality into software because of change. Software must be changed for it to continue to be useful. Software may be changed in order to correct latent errors and deficiencies not detected during development. It may also be changed in order to adapt it to the changing computing environment and to respond to changing user needs. When software is changed its quality is threatened. New errors are introduced and the conceptual integrity may be destroyed. In an attempt to monitor and evaluate program quality the concept of quality checklists are introduced.

Quality checklist is a list of questions testing for the presence of certain program properties considered essential in a high quality software system. Each question is answered with a "yes" or "no" depending on the qualitative judgement of the human evaluator. Software quality relates to its reliability, testability, modifiability and portability.

3.6.1 **Program Reliability**

Program reliability is defined as the extent to which a program correctly performs its functions in a manner intended by the users as interpreted by its designers. A reliable program is correct, complete, and consistent.

Total reliability remains a goal rather than an actuality in virtually all "real world" software. There is no means of guaranteeing 100% reliability or of measuring exactly how reliable a program is.

Although, there exist some mathematical methods developed to establish program correctness through formal proofs but difficulties in automating these methods have not enable them to be used economically as a viable practice. The more practical and commonly accepted approach for showing correctness is program testing [James *et al*, 1983]. The following checklist or questions may assist the evaluator or the person responsible for monitoring and evaluating program reliability:

a. Does the program contain checks for potentially undefined arithmetic operations (e.g. division by zero)?

b. Are loops termination and multiple transfer index parameter ranges tested before they are used?

c. Are subscript ranges tested before they are used?

d. Are error recovery and re-start procedures included?

e. Are numerical methods sufficiently accurate?

f. Are input data verified and validated?

g. Are test results satisfactory? Do actual output results corresponds exactly to expected results?

h. Do tests shows that most execution paths have been exercised during testing?

i. Do tests concentrate on most complex module interfaces?

j. Do tests cover the normal extreme, and exceptional processing cases?

k. Is the program tested with real as well as contrived data?

l. Does the program make use of standard library routines rather than develop its own code to perform commonly used functions?.

3.6.2 **Program Testability**

Program testability is the ease with which program correctness can be demonstrated. Testability is an extremely important program property in terms of building a high quality software product and in terms of controlling software cost. For medium to large scale software systems, approximately half of the development budget is expended on testing [Jensen *et al*, 1979]. In [Mills, 1976], it is reported that a program that is difficult to test during development carries this property into the maintenance phase. The testability checklist are the following:

a. Is the program modularised and well structured ?.

b. Is the program reliable?

c. Is the program understandable?

d. Can the program display optional intermediate results?

e. Is program output identified in a clear and descriptive manner?

f. Can the program display all inputs upon request?

g. Does the program contain a capability for tracing and displaying logical flow of control?.

h. Does the program contain a check point - restart capability?

i. Does the program provide for display of descriptive error messages?

3.6.3 **Program Modifiability**

Program modifiability is the ease with which a program can be changed. A modifiable program is understandable, flexible and simple.

A programmer has a relatively low probability of success when modifying a program. In [Boehm, 1973], it is reported that, if modification involves fewer than 10 program instructions, the probability of correctly changing the program on the first attempt is approximately 50%, but if the modification involves as many as 50 program instructions, the probability drops to 20%. A major contributor to the high cost of software maintenance is the abundance of fragile programs in the industry. In [James *et al*, 1983], it is observed that since change is the prevalent cause of most maintenance work, improving the modifiability of software is an important factor in reducing maintenance cost. The modifiability checklist are the following:

a. Is the program modular and well structured?

b. Is the program understandable?

c. Is there additional memory capacity available to support program extensions?

d. Is information provided to evaluate the impact of a change and to identify which portions of the program must be modified to accommodate the change?

e Is redundant code avoided by creating common modules/sub-routines?

f. Does the program use standard library routines to provide commonly used functions?

g. Does the program possesses the quality of generality in terms of its ability to:

   i. Execute on different hardware configurations;

   ii. Operate on different input/output formats;

   iii. Function in subset mode performing a selected set of features;

iv. Operate with different data structures or algorithms depending on resource availability?

i. Does the program possess the quality of flexibility in terms of its ability to:

i. Isolate specialised functions that are likely to change in separate modules;

ii. Provide module interfaces that are insensitive to expected changes in individual functions;

iii. Identify a subset of the system that can be made operational as part of contingency planning or for a smaller computer;

iv. Permit each module function to perform one unique function.

v. Define module inter communication based on the function the modules perform, not upon how the modules work internally?.

j. Is the use of each variable localised as much as possible?.

3.6.4 **Program Portability**

Program portability refers to the extent to which a program can be easily and effectively operated in a variety of computing environments. A portable program is well structured, flexible, and independent of features peculiar to a particular computer and/or operating system. The portability checklist are the following:

a. Is the program written in high-level, machine-independent language?

b. Is the program written in a widely used standardised programming language, and does the program use only a standard version and features of that language?

c. Does the program use only standard, universally available library functions and sub-routines?

d. Are the program computation independent of word size for achievement of required precision or memory size restrictions?

e. Does the program initialises memory prior to execution?

f. Does the program isolate and document machine-dependent statements?

g. Is the program structured to allow phased (over lay) operations on a smaller computer?

h. Has dependency on internal bit representation of alphanumeric or special characters been avoided or documented in the program?.

3.7 Collection of Metrics Data

In order to develop accurate estimates, [Pressman, 1987] observes that a historical baseline must be established. The baseline consists of data collected from past software development projects. For the baseline to be an effective aid in cost and effort estimation, baseline data must have the following attributes:

a. Data must be reasonably accurate. Guesses about past projects must be avoided.

b. Data must be collected for as many projects as possible.

c. Measurements must be consistent

d. Applications should be similar to the work that is to be estimated.

For instance, it makes little sense to use a baseline for batch information system work to estimate a real-time microprocessor application.

After data collection and computation of metrics, [Pressman, 1987] proposes that the result should be evaluated in order to critically examine the relevance of the results obtained from the project at hand; and also to look at the project and the circumstances surrounding the process of data collection. It then proposes a spreadsheet model for collection and computation of historical software baseline data. The model includes cost data, size-orientated data, and function-orientated data, enabling computation of both LOC and FP orientated metrics.

3.8 Software Project Scheduling

Scheduling of software development projects can be viewed from two different perspectives

[Pressman, 1987]:

a. Developing software to meet an already established (irrevocable) end-date. In this case, the software development team is constrained to distribute efforts within the prescribed time frame.

b. End-date is set by the software development team after careful analysis of the development process. Effort is then distributed to make the best use of available resources to accomplish the task at hand.

The first perspective is encountered more frequently than the second. Pressman, noted that accuracy in scheduling can sometimes be more important than accuracy in costing because in a product - orientated environment, added cost can be absorbed by repricing or amortization over large number of sales. A missed schedule, however, can reduce market impact, create dissatisfied customers, and raise internal costs by creating additional problems during system integration. Hence, when software project scheduling is being taken, a number of questions must be asked, among which are:

a. How do we correlate chronological time with human effort?

b. What tasks and parallelism are to be expected?

c. What milestones can be used to show progress?

d. How is effort distributed throughout the software engineering process?

e Are scheduling analysis methods available and how do we physically represent a schedule?.

3.8.1 **Task Definition and Parallelism**

When more than one person is involved in a software engineering project, it is likely that development activities will be performed in parallel. The degree of modularity in the design of a software would determine to what extent the jobs can be done in parallel. As components of software are developed, they are integrated and validated before release to the end user.

3.8.2 **Scheduling Method**

Scheduling of a software project is similar to the scheduling of any multitask development effort. Hence, generalized project scheduling tools and techniques can be applied to software with little modification. Project Evaluation and Review Technique (PERT) and Critical Path Method (CPM) are two project scheduling methods that can be applied to software development [Wiest *et al*, 1977]. Both techniques develop a task network description of project, that is, a pictorial or tabular representation of tasks that must be accomplished from the beginning to the end of a project. The network is defined by developing a list of all tasks associated with a specific project and a list of orderings (sometimes called a restriction list) that indicates in what order tasks must be accomplished. Both PERT and CPM provide quantitative tools that allow the software planner to:

a. Determine critical path - the chain of tasks that determines the duration of the project.

b. Establish most likely time estimates for individual tasks by applying statistical models.

c. Calculate boundary times that define a time frame for a particular task.

Boundary time calculations can be very useful in software project scheduling slippage in the design of one function, for example, it can retard the further development of other functions. The important boundary times that may be discerned from a PERT and CPM network are the following:

a. The earliest time that a task can begin when all preceding tasks are completed in the shortest possible time.

b. The latest time for task initiation before the minimum project completion time is delayed.

c. The earliest finish - the sum of the earliest start and task duration.

d. The latest finish - the latest start time added to task duration.

e. The total float - the amount of surplus time or leeway allowed in scheduling tasks so that the network critical path is maintained on schedule. Boundary time calculations lead to

a determination of critical path and provide the manager with a quantitative method for evaluating progress as tasks are completed.

Maintenance effort may not be easy to estimate at the development stage and since, it accounts for a large percentage of project cost, the goal of software engineering should be to help reduce maintenance cost to the barest minimum.

3.9 Software Project Cost Estimation

Software cost estimation is a continuous activity which starts at the proposal stage and continues throughout the lifetime of a project. Projects normally have a budget and continual cost estimation is necessary to ensure that spending is in line with the budget. In [Boehm, 1981], the following seven different techniques of software cost estimation are proposed:

a. Algorithmic cost modelling is developed using historical cost information which relates some software metric (usually its size) to the project cost. An estimate is made of that metric and the model predicts the effort required.

b. Expert judgement on the software development techniques to be used and on the application domain are consulted.

c. Estimation by analogy is employed when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects.

d. Parkinson's Law which states that work expands to fill the time available. In software costing, this means that the cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months.

e. The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customer's budget and not on the software

functionality.

f. A cost estimate is established by considering the overall functionality of the product and how that functionality is provided by interacting sub-functions. Cost estimates are made on the basis of the logical function rather than the components implementing that function.

g. The cost of each component is estimated. All these costs are added to produce a final cost estimate.

Each technique has advantages and disadvantages. The most important point made by Boehm is that no one technique is better or worse than any other technique. For large projects, several cost estimation techniques should be used in parallel and their results compared. If these predict radically different costs, this implies that not enough costing information is available. More information should be sought and the costing process repeated. The process should continue until the estimates converge.

Software project estimation cannot be undertaken without risk and that any software project estimation technique (or set of techniques) should strive to provide the highest degree of reliability, that is, the lowest risk of major estimation error. There are the following options in order to achieve reliable cost and effort estimates:

a. Delay estimation until late in the project

b. Use of decomposition techniques to generate project cost and effort estimates

c. Develop an empirical model from software cost and effort estimate

d. Acquire one or more automated estimation tools

The first option, though attractive, is not practical. This is due to the fact that cost estimates are made "upfront". Hence, the option of delaying estimation until late in the project is not popular among Systems Analysts.

Decomposition techniques involves dividing a complex problem into smaller, manageable

sizes. Each of the subdivisions of the problem is solved and the solutions are combined to solve the original problem.

Empirical estimation model is made up of estimation models and resource models. Empirical estimation models could be used to complement decomposition techniques and also offer a potentially valuable estimation approach. An estimation model for computer software uses empirically derived formulae to predict data that are used for software project planning.

Resource models consists of one or more empirically derived equations that predict effort (in person-month), project duration (in chronological months), or other pertinent project data. There are four classes of resource model: static single variable model, static multivariable models, dynamic multivariable model and theoretical model [Basili, 1980].

Examples of single variable models are found in the work of [Waltson, 1977] and the Constructive Cost Model (COCOMO) introduced by [Boehem, 1981], which presents a hierarchy of software estimation models.

The Putnam Estimation Model [Putnam, 1978], is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project. The model is derived from manpower distributions encountered on large projects (total effort of 30 person-years or more). It is however, possible to extrapolate to a smaller software projects.

The distribution of effort for large software projects has been described in curves and proved analytically by [Norden, 1980]. This curves are often referred to as Raleigh - Norden curve. Other empirical models are Function Point Models,[Albrecht and Gaffney, 1983] and Time Study Model proposed by [Esterling, 1980].

Automated Estimation Tools allow a planner to estimate cost and effort and to perform "what if" analysis for important project variables such as delivery date or staffing. There are many automated estimation tools but all exhibit the same general characteristics and all require one or more of the following data categories:

a. A quantitative estimate of project size (e.g. LOC) or functionality (function point data).

b. Qualitative project characteristics such as complexity, required reliability, or business criticality.

c. Some description of the development staff and/or development environment.

From these data, the model implemented by the automated estimation tool provides estimates of effort required to complete the project, costs, staffing, development schedule, and associated risk.

SLIM applies the Putnam Software Model, linear programming, statistical simulation, and PERT techniques to derive software project estimates. SLIM enables a software planner to perform the following functions:

a. Calibrate the local software development environment by interpreting historical data.

b. Create an information model of the software to be developed by using basic software characteristics, personnel attributes, and environmental considerations.

c. Conduct software sizing.

Once software size has been established, SLIM computes size deviation, a sensitivity profile that indicates potential deviation of cost and effort, and a consistency check with the data collected for software systems of similar size. SLIM also provides a month - by - month distribution of effort and cost so that staffing and cash flow requirements can be projected.

ESTIMACS is a "macro-estimation model" that uses function point estimation method enhanced to accommodate a variety of project and personnel factors. In [Presman, 1987], it is reported that ESTIMACS tools contains a set of models that enable the planner to estimate the following:

a. System development effort

b. Staff and cost

c. Hardware configuration

d. Risk

e. The effects of "development portfolio" (i.e. the effects of the current work).

3.9.1 Maintenance Cost Estimation

Maintenance cost are difficult to estimate. Evidence from existing systems suggests that maintenance cost are, by far, the greatest cost incurred in developing and using a system. Maintenance cost vary widely from one application domain to another. On the average, it is seem to be between 2 times and 4 times of the development cost for large software systems [Sommerville, 1992]. It is usually cost effective to invest time and effort when designing and implementing a system to reduce maintenance cost. For business application systems, a study by [Guimaraes, 1983] shows that the overall lifetime cost of a software system can decrease as more effort is expended during systems development to produce a maintainable system.

Maintenance cost is influenced by non-technical and technical factors and each of these is related to a number of product, process and organization, which are discussed below. The technical factors that influence maintenance cost are as follows:

a. **Application Domain:** If the application is clearly defined and well understood, the system requirements are likely to be complete. Relatively, little adaptive maintenance is necessary. If the application is completely new, it is likely that the initial requirements will be modified frequently as users gain experience with the system.

b. **Staff Stability**: Maintenance cost can be reduced if system developers are responsible for maintaining their own programs. There is no need for other engineers to spend time understanding the system. In practice, it is very unusual for developers to maintain a program throughout its useful life.

c. **The Lifetime of Program**: The useful life of a program depends on its application. Programs become obsolete when the application becomes obsolete or when their original

hardware is replaced and conversion cost exceeds re-writing costs. Maintenance cost usually rise with program age.

d. **External Environment:** A software system must be modified as that environment changes. For example, changes in a taxation system might require payroll and accounting programs to be modified.

e. **Hardware Stability:** If a program is designed for a particular hardware configuration which does not change during the program's lifetime, no maintenance due to hardware changes will be required. Programs must be modified to use new hardware which replaces obsolete equipment.

The technical factors that affects maintenance cost are:

a. **Module Independence:** It should be possible to modify one component of a system without affecting the other system components.

b. **Programming Language:** Programs written in a high-level programming language are usually easier to understand (and maintained) than programs written in a low-level language.

c. **Programming Style:** The way in which a program is written contributes to its understandability and hence, the ease with which it can be modified.

d. **Program Validation and Testing**: Generally, the more time and effort spent on design validation and program testing, the fewer errors in the program. Consequently, corrective maintenance cost are minimised.

e. **Quality of Program Documentation**: Program maintenance costs tends to be less for a well documented systems than for systems supplied with poor or incomplete documentation.

In [Pressman, 1987], it is observed that the cost of software maintenance has increased steadily during the past twenty years. It is submitted that 35 - 40% and 40 - 60% of overall

software budget were expended on the maintenance of existing software in the 1970s and 1980s respectively. It is also, projected that between 70% - 80% would be expended in the 1990s. Although industry averages are difficult to ascertain and open to broad interpretation, the typical software development organisation spends between 40% and 70% of expenses in conducting corrective, adaptive, perfective and preventive maintenance.

3.10 Effort Distribution

In [Pressman, 1987), it is observed that in systems development, distribution of efforts on system analysis and design, coding and testing are 40%, 20% and 40% respectively. This is reffered to as the 40-20-40 rule. The 40-20-40 rule is, however, a guideline as the characteristics of each project usually dictate the distribution of effort. Effort expended on project planning rarely accounts for more than 2%-3% of effort unless the plan commits an organisation to large expenditures with high risks. Requirement analysis may account for 10%-20% of project effort. Effort expended on analysis or prototyping should increase in direct proportion to project size and complexity. A range of 20%-30% percent of effort is normally applied to software design. Time expended for design review and subsequent integration must also be considered, because of the effort applied to software design, coding should follow with relatively little difficulty. A range of 10% - 20% of overall effort can be achieved. Testing and subsequent debugging can account for 30% - 50% of software development effort. The nature of the task that a software system performs often dictates the amount of testing that is required. If software is human-rated, that is, software failure can result in loss of life, even higher percentage may be considered.

CHAPTER FOUR

SOFTWARE DEVELOPMENT CYCLE

Software development project could be broadly categorized into three major phases namely, development, operations and maintenance. Each of these three major phases could be further sub-divided into activities [Sommerville, 1992].

4.1 Development Phase

Some of the activities that made up the development phase are:

a. Definition of the problem

b. System design

c. Program coding

d. Program walkthrough

e Program compilation

4.1.1 Definition of the Problem

The need to develop a new software is usually motivated by a felt need of an organisation. That is, a software project idea is aimed at achieving certain objectives which are stated in qualitative terms. It is the duty of the System Analyst to quantify the objectives so that performance could be measured against pre - determined target.

During this phase, an otherwise complex problem is broken down into sub-problems which are manageable. The requirements for solving the problem such as programming language, output reports, input data, processes, hardware and other inputs are carefully analysed.

Usually, there are existing solutions to a problem. For example, there may be a manual method of solving the problem or even an existing computer software in operation. The development of a new software may arise as a way of overcoming the limitations of the existing solution.

The end product of the system analysis should be the identification of alternative solutions to a problem, the cost implication of each alternative and the careful selection of a good solution to the problem. The criteria for selecting a software development project includes:

a. Potential return on investment;

b. Management desire;

c. Requirement for integration with other systems;

d. Technical feasibility of the proposed project;

e. Critical company need.

f. Capacity of the organisation to implement the project. This is in terms of technical know-how of the personnel, ability to absorb the additional work-load in an additional project.

4.1.2 System Design

The main activities which are undertaken at this phase include:

a. Defining output requirements, volume, frequency, format and distribution.

b. Specifying the input layouts, frequency and so on.

c. Developing the overall system logic.

d. Determining the control and audit procedures.

e. Determining the data elements, output requirements, data relationships and information flow.

f. Identifying the master files, transaction files, data volumes, frequency of updating, length of retention, speed of response required from files.

g. Deciding on which storage devices to use.

h. Determining the file organisation and access techniques.

i. Identifying computer programs and manual procedures required.

j. Deciding on division of the computer-based parts of the software development project

into individual computer runs.

k. Preparing program specifications.

l. Developing general test requirements (type of data source, control checks and so on).

m. Producing detailed plan of implementation.

n. Revising estimates of operational costs of the program.

o. Documenting the design phase in a report for user and system management.

In designing a program, a number of tools such as flowcharts, block diagrams, decision tables and IPO charts are used.

Program flowchart is a symbolic representation of the processing that should take place in a program. Flowchart is a pictorial or symbolic representation of a program and it is independent of programming languages. It can be coded using any programming language. It uses symbols to show the detailed steps within the program and the order in which the steps are performed. It indicates the "logic" of a program. Program flowchart serves the following purposes:

a. Specifies the logic of a program.

b. Helps to sort out the procedural steps in a program.

c. Serves as an aid to program construction and coding.

d. Serves as a reference when testing and debugging the program.

e. Serves as a documentation of the program, either for future modifications or for use by other personnel.

Flowchart is a pictorial or symbolic representation of a program and it is independent of programming languages. It can be coded using any programming language.

General principles of the preparation of flowchart include:

a. The flowchart should be clear, neat and easy to follow.

b. The flowchart should have logical start and finish.

c. As much as possible, crossed flow lines should be avoided.

d. Comparison instructions should be made simple so that choice or selection is based on "Yes" or "No" answers.

e. Keep the direction of flow of the chart down the page and left to right. Indicate by arrows if the flow is against any direction.

f. Check that the flowchart is logically correct and complete. Check that no set of activities stops dead without returning to enter the flow at a logical point before END. This can be achieved by passing simple test data through the flowchart.

g. There should be consistency in the level of detail illustrated on the flowchart. Parts of the flowchart that needs to be shown in greater detail could be drawn on a separate sheet and referenced. Such modular approach to flowchart design helps to simplify program development.

An Input-Processing-Output (IPO) chart is a textual description of the input, output and the logic of a program. In IPO chart, emphasis is on the logic of the program component, routines and sub-routines rather than the fine details of the algorithm as in the case of program flowchart. The statements of the logic of the IPO chart of a program are described using a natural language and modifications to system can be effected without a total restructuring typical of program flowchart.

Decision tables are used as a method of defining the logic of a process, that is, the processing operations required in a compact manner. They are more convenient to use than program flowcharts in situations where a large number of logical alternatives exist. The basic format consists of four quadrants as follows:

Condition stub	Condition entry
Action stub	Action entry

The purpose of condition stub is to specify the values of data we wish to test for. The condition entry specifies what those values might be. Between them, the condition stub and condition entry show what values an item of data might have that the program should test for. Establishing conditions will be done within a program by means of comparison checks. The condition entry quadrants are divided into columns, each column representing a distinct 'rule' or unique result of all the conditions. The action entry quadrants shows the action or actions that will be performed for each rule. The important values of a decision table are:

a. Compact notation.

b. Easy program modification.

c. Standard form which facilitate automatic generation of machine language program.

d. Compatibilty with flowcharts.

e. It is possible to check that all combinations have been considered.

f. They show a cause and effect from actions to conditions.

g. It is easy to trace from actions to conditions.

h. They are easy to understand and copy as they use a standard format.

i. Alternatives can be grouped to facilitate analysis.

4.1.3 **Programs Coding**

Program coding refers to the act of writing of the computer instructions using a language or package. Before embarking on the coding of the programs, the programmer need to study the flowcharts, decision tables and/or IPO charts in order to be sufficiently familiar with the logic of the program to be written. In coding the program, the programmer should take cognisance of the problem to be solved and the end result.

He needs to consider the appropriate programming language or software package to be used in coding the program. The nature of data, volume of data, the type and complexity of

computations and so on, would determine the final choice of the programming language.

For instance, it is better to use a database language in coding a business orientated programs (e.g. payroll, personnel records, stock control, Inventory etc), which involves large volume of data but simple calculations. On the other hand in coding scientific or engineering problem which involves complex mathematical formulae but small volume of data, it is better to use a programming language like FORTRAN language, BASIC language or C - language.

Also, in coding programs, the programmer should put the optimal performance of the program into consideration. In a situation where there are two or more alternative algorithms for accomplishing a task, the cost analysis of each algorithm has to be carried out. Also, the trade-off analysis of the alternative algorithms should be carried out. The algorithm which is less costly in terms of storage space and/or input-output processing time should be coded.

4.1.4 **Program Walkthrough**

Program walkthrough is undertaken in order to detect errors committed by the programmer during coding. Usually, computer programs are checked by another programmer or a supervisor for possible errors in coding. There are two major types of errors that can be found in a typical program, namely, syntax error and semantic error.

Syntax error is often referred to as 'computer grammatical error', syntax error may occur as a result of any of the following:

a. Assigning the same symbolic name to two different sets of data.

b. Omission of an instruction or an operator between two operators.

c. Misuse of an operation symbol.

d. Incorrect spelling of a keyword .

e. Omission of a special character such as comma.

Whenever there is a syntax error, the program will terminate at the compilation stage thus

preventing the production of the object code.

Semantic error may be due to inability of certain conditions in the program to adequately represents the data processing operation required within the program. An example is division of a real number by zero. Semantic error may also be caused by error of logic. That is, when the solution to a particular problem is based on a wrong logic. Semantic errors are more difficult to detect than syntax error.

An important benefit of program walkthrough is that it saves time during compilation and test running of a program. In order to ensure a thorough program walkthrough a list of questions have been suggested in [Collins *et al*, 1982]. Such questions are known as program walkthough checklist. The supervisor or the person performing the walkthrough should ensure that the program meets the requirements stated in the checklist. Such walkthrough checklist include:

a. Is there a transaction profile for each input?

b. Have all the correct process structures been identified?

c. Do the processes occur in the correct sequence?

d. For each process, are all the data required specified?

e. For each process, are all the files required specified?

f. Are there dependencies which have not been annotated?

4.1.5 **Program Compilation**

Each programming language has a language translator which translates the source code into the object code. Depending on the type of language, the translator can be an assembler, an interpreter or a compiler. The object program generated in the compilation process is a machine language program which performs the instruction originally presented in the coding.

The translator checks the source code line by line for any syntax error. Usually, the running of the program is halted whenever a syntax error is encountered. Appropriate corrections are made and the corrected version of the source code is again re-compiled until it

is free of syntax errors when an object code or object program is produced and tested using a sample data.

4.2 System Operations

Some of the activities invloved in system operations phase are:

a. Data survey and collection

b. System test run

c. Parallel run

d. Full implementation

e. Documentation

4.2.1 Data Survey and Collection

The computer programs written are meant to process a set of input data to generate the desired output reports. Data survey and collection are very important steps most especially when programming a problem that has as input large volume of data. The sources of input data to a program have to be carefully defined and the source document should be carefully studied in order to ensure that all relevant data are available. Input data could be collected from any of the following sources:

a. Conducting interview and meetings;

b. Administering questionnaire;

c. Sampling users opinions;

d. Searching existing records (output reports, input data and the documentation of operations).

Data may be collected by applying the combinations of two or more sources outlined above. The data collected should be analysed in order to formulate their logical relationships, degree of association, estimated population and layout of standard input forms. Data collected

are coded (written) into standard input forms. Depending on the level of sophistication or complexity of the data processing department, different categories of personnel may be involved in the data coding process. Such personnel may include a data coder, data supervisor, data analyst and so on. The coded data values should be manually verified and validated by the data analyst before they are passed to the data entry staff for input into the computer.

Data cleaning is carried out by printing out the data entered into the computer for manual checking by the Data Analyst. A computer program can even be coded and implemented to perform data cleaning operations. Such program may be designed to check for duplicate records, validate each field of every record of a file and generate error messages. An edit program can also be used to validate the value set of data.

4.2.2 **Program Test Run**

Testing is defined as the controlled execution of a program in order to reveal errors [Janson *et al*, 1979]. The end result of program compilation produces the object code or object program. The object program though free of syntax errors may contain semantic errors. Hence, the object program need to be tested using a sample data. The sample data should be selected such that every branch of the program would have been checked against every combination and sequence of data conceivable under actual operating condition. In order to ensure a thorough test run, the correct output of the sample transaction should be pre-determined so that the result of the program test run could be carefully compared with it. Sometimes it might be helpful to test for intermediate results in order to detect where the suspected error emanates from.

Sometimes it may even be necessary for the programmer to develop a debugging routine in order to eliminate costly computer time. The print out of the areas that are causing error should be produced and the line numbers of the statements causing the errors are displayed.

4.2.3 **Parallel Run**

Parallel run of systems are carried out in order to investigate and compare the performance of the new system with the old system using real life data. Investigation must be carried out by using the same set of data. The result generated by the two systems are then compared for any possible disagreement. If any disagreement is discovered, it should be carefully studied and the cause of such disagreement identified and solutions should be found.

During the parallel run of the existing and new systems, the cost of the two systems in terms of computer time, storage space, computer consumables and human resources can be easily quantified thereby permitting the comparison and quantification of the benefits of the new system over the existing one.

4.2.4 **Full Implementation**

Once it is established that a new system is better than the existing one, the old system can be discontinued. The full implementation of the system should be handled with care because of psychological, social and technical effect it may have on the employee. It is often difficult to adjust what one has been used to; hence, the employees may find it difficult or even be unwilling to jettison their 'old way' of doing things with the apparently new one. Such factors like the fear of re-training, re-deployment, redundancy, and even the fear of loss of job may make the employees not be willing to adopt the new system.

Hence, the personnel involved in the existing operations need to be assured of the security of their jobs. An evolutionary approach which permits a smooth take off of computerization exercise and guarantees job security of the existing employees of the user departments should be suggested.

The approach does not render the participation of the existing human labour obsolete,

rather it should enforce the retraining or re-orientation of the human resources and an efficient method of doing things.

4.2.5 **Documentation of System**

Documentation is an integral part of all the phases of system development. It is a document that should provide all facts and figures about the system. It should describe all inputs, processes and reports generated. A typical system documentation should consist the following items, among others:

a. Problem description

b. System design

c. System flowcharts, decision tables, etc

d. Input and output

e. Records and file formats

f. Printed copy of the programs

g. Copy of the test data used during the debugging phase and so on.

The major documents normally produced in system documentation are contained in the user guide and system reference manual. The user guide should contain the following documents:

a. A functional document which explains what the system can do.

b. An installation document which explains how to install the system and tailor it to a particular hardware configuration.

c. An introductory manual which explains, in simple terms, how to get started with the system.

d. A reference manual which describes in detail the facilities available to the user and how to use these facilities.

e. An operator guide which explains how the operator should react to situations which arise whilst the system is being run.

The system reference manual should contain the system specification, input, processes and outputs. The system reference manual is essential for the proper understanding and maintenance of the system. The system reference manual should include:

a. The system requirements specification.

b. An overall system specification showing how the system is decomposed into a set of interacting modules.

c. A description of each system module with emphasis on its functions, input data and output reports.

d. A test plan showing how the integrated unit of the system is tested.

e. An acceptable test plan, devised in conjunction with the user.

4.3 System Maintenance

System maintenance may involve simple changes to correct coding errors, more extensive changes to correct design errors or a total review to correct specification errors or accommodate new requirements. System maintenance is carried out after the system has been delivered and is in use by the user organisation. System maintenance falls into three categories, viz: perfective maintenance, adaptive maintenance and corrective maintenance.

Perfective Maintenance is aimed at addressing changes demanded by the user of the system. For instance, users may request for new reports to be generated from the system or the programmer may see the need for some facilities which might support end user more effectively and efficiently. Also the programmer may decide to replace an existing algorithm with a newly developed and more efficient one. Perfective maintenance attempts to improve the system without changing its functionality.

Adaptive maintenance may be carried out due to changes in the environment of the system. For example, adaptive maintenance may be carried out due to:

a. The need to change from batch processing to on-line processing;

b. The need to replace the existing peripherals with some new ones;

c. The need to change the existing implementation technique.

Corrective maintenance involves the correction of undiscovered system errors. The errors may occur when the test data are not representative enough to test all the various branches of the system and for a period of time during the full implementation of the system, the real life data to test such branches are not used. For example, it may not have occurred to the designer of a payroll system at the onset that an employee may have negative monthly pay. It may happen that during the implementation that an employee have obtained several loans to the extent that his total monthly earnings is less than the total monthly deduction.

The cost of maintenance of a system far out weighs the cost of system development. Maintenance cost vary from one application to another, but, on the average, maintenance cost is about four times the development cost for a large system [Sommerville, 1992]. Also, maintenance cost increases with the age of the system.

Since, it is difficult to estimate, in advance, the cost of maintaining a system, it is better to invest time and efforts when designing and implementing a system. There are the short term and term goals of the system. Emphasis, should however, be on the long term goal thereby leading to a term benefit and reduction of maintenance cost. Coding errors are usually relatively cheap to correct, d errors are more expensive as they may involve the re-writing of several system components. Require errors are the most expensive to repair because of re-design which is involved.

In [Arthur, 1988] a model of maintenance process which is shown in Figure 4.2 below is prop

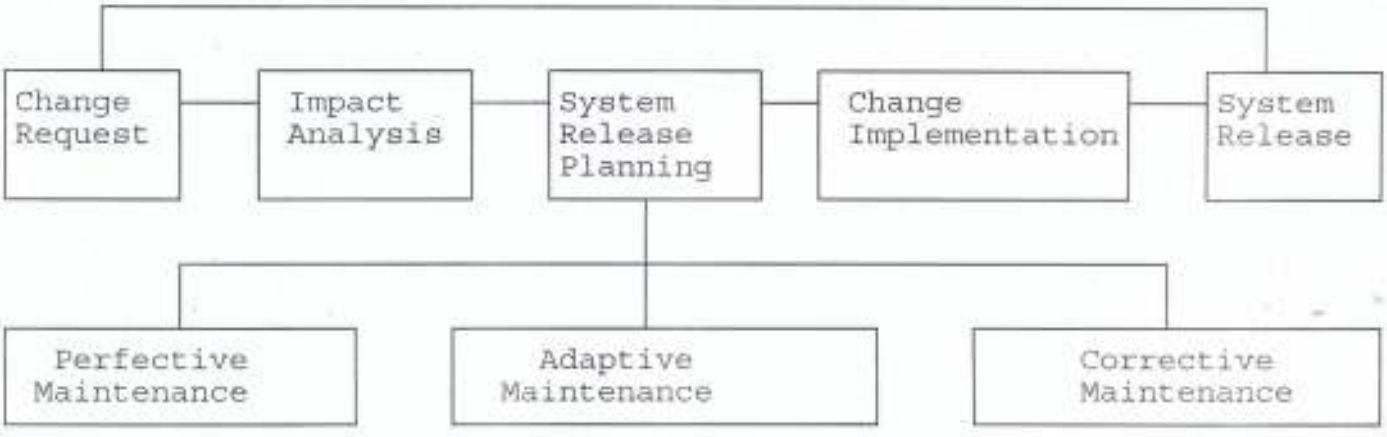


Figure 4.2: Maintenance process

4.4 System Monitoring

A project is any distinct economic activity within the domain of economic and social development. As a unit of investment decision, it may be simple or complex. It has the following characteristics:

a. It has a specific time dimension and it is geared towards accomplishing a certain set of objectives.

b. It has a latent stream of benefits and a latent stream of cost.

c. It may or may not have a spatial dimension.

Since resources are limited and there are many alternative ways of investing them, it follows that, there are alternative ways of achieving a given objective. It then means that, there has to be a proper monitoring of projects in order to ensure judicious allocation of scarce resources.

Regardless of the type of project under consideration, a project is normally divided into phases. Before the end of a project, there is the need for proper monitoring of every phase and/or activity of the project in order to ensure that achievement tally with the predetermined objectives of the project.

Project monitoring involves checking performance against targets. Monitoring helps to

discover any shortcoming or flaw in the execution of the project. Once, the flaw is detected, corrective and/or preventive measures are taken promptly. Monitoring allows corrective measures to be taken while the project is on. It improves project management. The method of monitoring adopted and the steps involved in checking achievement against target must be well articulated and strictly adhered to unless there is a genuine need for adjustment or changes in the method used based on past experience.

The method and steps involved depend largely on the type and nature of the enterprise being monitored. It has been suggested that monitoring is better carried out by individuals or agencies/organizations external to the project [Kayode, 1985]. This would lend credibility to the exercise. Whether monitors are external or within the project, the following must be noted, among others:

a. Good knowledge of the environment in which the project is being implemented.

b. Sound knowledge of the technical aspects of the project. They must know the different operations involved.

c. Know the procedures for the employment of labour and their wages.

d. Have accurate record of every input committed to the project, the source, quantity and cost.

For a computer programming project, monitoring should centre around the following:

a. Total costs incurred.

b. Man-time spent.

c. Activities completed.

d. Materials and machine time consumed.

e. The quality of work done at every phase of the project.

Within these broad categories, the project monitoring system should relate the progress to the budgeted time, cost and so on, at an activity, and if appropriate, individual level, for

example, number of system instructions completed, number of terminals delivered and installed and so on.

4.5 System Performance Evaluation

The performance evaluation of a system is undertaken after the system has been completed and has been running error free for several cycles. *Ex post* evaluation is done for the following three main reasons:

a. To examine the system and see whether or where improvement can be made.

b. To compare the achievements of the system against the set objectives.

c. To provide feed back to the Systems Analyst so that he can draw lessons for the good and bad points to be used for future projects.

For the purpose of evaluation, the performance of the system should be assessed from the following aspects:

a. Actual Cost: The cost analysis of all the resources used in the system development should be carried out. The analysis should cover the cost of computer consumables, hardware, personnel cost, operational cost and so on. Also, the human and machine time spent on system analysis, design, implementation, documentation and maintenance can be quantified in terms of money.

b. Realized Benefits: The benefits realized should be compared with the projection made during the feasibility and design phase. If the benefits are higher than the cost of the system development then the new system is a success in the sense that it has generated considerable returns on investment.

c. Timing: An important attribute of any project is that it must have a starting point and a definite end. That is, it must be completed within a time frame. In order to achieve this objective, each phase of the project is also allotted specific time frame within which they must be completed. The estimated time taken to complete each phase of the project

should be carefully compared with the actual time taken to complete them. Any discrepancy in the forecast time and the actual time spent should be noted and the reasons for it noted for future references.

d. Problem Areas: All problem areas identified in the analysis, design, implementation, documentation and maintenance of the system should be carefully noted and solutions should be sought. As the system evolves, effort should be made to cut down cost. This may be achieved through a low-level or high-level approach.

i. Low-level approach may be achieved by using the following strategies:

- Improving the efficiency of the system through the use of better algorithm..
- Restructuring the logical connections of the operational data.
- Periodic re-organization of the stored data with a view to reducing the retrieval and update cost.

ii. High-level approach involves the constant training of the personnel responsible for the operation of the system. Such training should be both on short term and long term basis. This will enable the personnel involved to keep abreast of the developments in the computer industry. This may be achieved through regular attendance of seminars, conferences, workshops, in-service training and so on.

e. User satisfaction: In order to realize the full benefits of a system, user satisfaction with the operations and functions as well as the output of the system should be evaluated. Users opinion should be sampled from time to time in order to find solutions to their observations. Sometimes, the user may need to be given proper re-orientation on the use and benefits of the system.

f. Error Rates: The rate at which errors are generated should be evaluated. The nature of such errors and their causes as well as remedies should be evaluated and documented.

CHAPTER FIVE

DESIGN AND IMPLEMENTATION OF COMPUTER AIDED SYSTEM

In this chapter, we present the framework for the estimation and implementation of software metrics using Project Evaluation and Review Technique (PERT) and Critical Path Method (CPM). The metrics derived from these project scheduling methods are used for the design and implementation of the Computer Aided System christened CASMESDOM.

5.1 Software Metrics

Two categories of software metrics considered in this project are the following:

a. Direct metrics

b. Indirect metrics

The direct metrics include those measures of software which can be determined directly. Such direct measures include:

a. Cost

b. Duration

The indirect metrics refers to measures of software quality whose determination are somewhat subjective in nature. The software quality factors focus on three important aspects of a software product namely:

a. Its operational characteristics (product operations)

b. Its ability to undergo change (product revision)

c. Its adaptability to new environments (product transition)

This is represented in Figure 5.1.

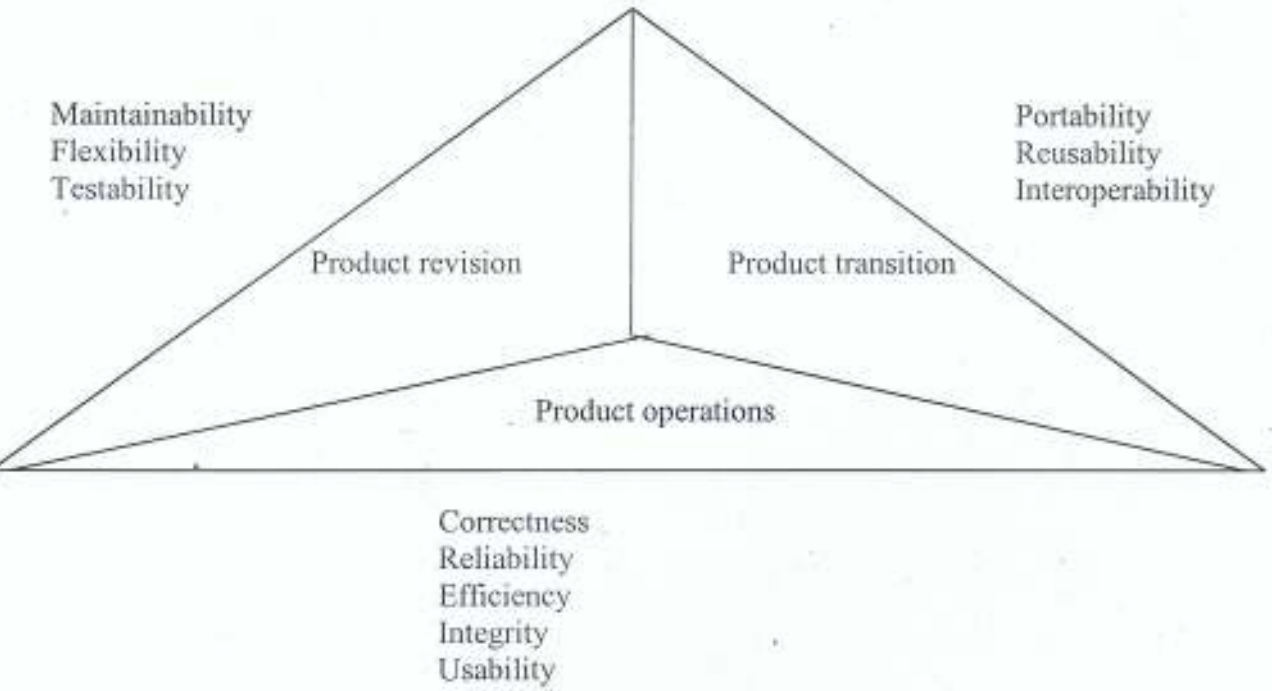


Figure 5.1: Software Quality Factors

The description of the software quality factors are presented as follows:

a. Correctness: The extent to which a program satisfies its specification and fulfils the end users mission.

b. Reliability: The extent to which a program can be expected to perform its intended function with required precision.

c. Efficiency: The amount of computing resources and code required by a program to perform its function.

d. Integrity: The extent to which access to software or data by unauthorized persons can be controlled.

e. Usability: The effort required to learn, operate, prepare input, and interpret the output of a program.

f. Maintanability: The ease with which software can be understood, corrected, adapted and/or enhanced.

g. Flexibility: The effort required to modify an operational program.

h. Testability: The effort required to test a program to ensure that it performs its intended function.

i. Portability: The effort required to transfer the program from one hardware and /or software system environment to another.

j. Reusability: The extent to which a program (or part of a program) can be reused in other applications. This is related to the packaging and the scope of the functions that the program performs.

k. Interoperability: The effort required to couple one system to another.

It is difficult and in some cases impossible to develop direct measures of the above factors. Therefore, a set of metrics are defined and used to develop expressions for each of the factors according to the following relationship:

$$F_q = C_1 * M_1 + C_2 * M_2 + \ldots + C_n M_n$$

where F_q is a software quality factor, C_n are regression coefficients, and M_n are the metrics that affect the quality factor. Unfortunately, many of the metrics defined by McCall can only be measured subjectively. The metrics were then assessed by checklist to "grade" each of the quality factors. The metrics proposed in [McCall , 1977] are the following:

a. Auditability: The ease with which conformity to standards can be checked.

b. Accuracy: The precision of computations and control.

c. Communication commonality: The degree to which standard interfaces, protocols, and bandwidth are used.

d. Completeness: The degree to which full implementation of required function has been achieved.

e. Conciseness: The compactness of the program in terms of lines of code.

f. Consistency: The use of uniform design and documentation techniques throughout the software development project.

g. Data commonality: The use of standard data structures and types throughout the program.

h. Error tolerance: The damage that occurs when the program encounters an error.

i. Execution efficiency: The runtime performance of a program.

j. Expandability: The degree to which architectural, data, or procedural design can be extended.

k. Generality: The breadth of the potential application of program components.

l. Hardware independence: The degree to which the software is decoupled from the hardware on which it operates.

m. Instrumentation: The degree to which the program monitors its own operation and identifies errors that do occur .

n. Modularity: The functional independence of program components.

o. Operability: The ease of operation of a program.

p. Security: The availability of mechanisms that control or protect programs and data.

q. Self-documentation: The degree to which the source code provides meaningful documentation.

r. Simplicity: The degree to which a program can be understood without difficulty.

s. Software system independence: The degree to which the program is independent of non-standard language features, operating system characteristics, and other environmental constraints.

t. Traceability: The ability to trace a design representation or actual program component back to requirements.

u. Training: The degree to which the software assists in enabling new users to apply the system.

The relationship between software quality factors and metrics listed above is shown in Table 5.1. In the final analysis the metrics that affect each of the software quality factor is graded on a five-point scale as follows:

Poor = 1

Average = 2
Good = 3
Very Good = 4
Excellent = 5

Table 5.1: Relationship between software quality factors and metrics (adapted from Pressman, 1987).

QUALITY FACTOR	METRICS
1. Correctness	Completeness Consistency Traceability
2. Reliability	Accuracy Consistency Error tolerance Modularity
3. Efficiency	Conciseness Execution efficiency Operability
4. Integrity	Auditability Instrumentation Security
5. Maintainability	Conciseness Consistency Instrumentation Modularity Self-documentation
6. Flexibility	Conciseness Consistency Expandability Modularity Self-documentation Simplicity
7. Testability	Auditability Instrumentation Modularity Self-documentation Simplicity
8. Portability	Hardware independence Modularity Self-documentation Software system independence
9. Reusability	Generality Hardware independence Modularity Software system independence
10. Interoperability	Communication commonality Data commonality Modularity
11. Usability	Operability Training

5.2 Estimation of Software Project Schedule.

One of the important attributes of a project is that it has a specific time dimension and is geared towards accomplishing certain objectives. In other words, a project has a starting point and an end point. Between the starting point and the end point, there are activities or jobs which must be accomplished. Hence, for a project to conform to the time frame and the objectives, there is the need for a proper scheduling of the activities involved in the execution of the project. One of the earliest and popular methods of project scheduling is the Project Evaluation and Review Technique (PERT) and Critical Path Method (CPM).

5.2.1 Software Project Scheduling Using PERT/CPM

PERT/CPM were developed almost simultaneously by two different groups between 1956 and 1958. CPM was first developed by E.I.du Pont de Neumours and Company as an application ro construction projects and was later extended to a more advanced status by Mauchly Associates. PERT, on the other hand was developed for the US Navy by a consulting firm for scheduling the research and development activities for Polaris missile program [Taha, 1976].

PERT and CPM are basically time-orientated methods in the sense that they both lead to the determination of a time schedule. Although, PERT and CPM were developed almost independently, today, they comprise one technique and the differences if any, are perhaps, only historical. Consequently, both PERT and CPM are referred to as "project scheduling techniques". In this research, we employ the PERT/CPM for the estimation of time schedule and cost estimation of software development, operations and maintenance.

5.2.2 Network Diagram Representation

The first step in PERT/CPM project scheduling technique involves the breaking down of the project into a set of individual jobs or events and arranging them into a logical network. This is followed by estimating the duration and resource requirements of each job, deducing a schedule and finding which jobs controls the completion of the project. The arrow diagram represents the inter-dependencies and precedence relationships among the activities of the project. An arrow is

commonly used to represent an activity with its head indicating the direction of progress in the project. The precedence relationship between the activities is specified by using events. An event represents a point in time signifying the completion of some activities and the beginning of new ones. The beginning and end points of an activity are thus described by two events usually known as the tail event and the head event. Activities originating from a certain event cannot start until the activities terminating at the same event have been completed.

In the network theory terminology, each activity is represented by a directed arc and each event is represented by a node. The length of the arc need not be proportional to the duration of the activity nor does it have to be drawn as a straight line. The direction of progress in each activity is specified by assigning a smaller number to the tail event compared with the number of its head event. Conventionally, letter "i" is used to depict the tail event and letter "j" depicts the head event. Hence, a typical activity is represented by the symbol (i,j).

The rules for constructing the network diagram are summarised as follows:

a. Each activity is represented by one and only one arrow in the network. That is, no single activity can be represented twice in the network.

b. No two activities can be identified by the same head and tail events. In case of concurrent activities, the use of dummy or logical links may be introduced in order to avoid two events having the same head and the same tail events. A dummy is a logical link or constraint which represents no specific operation.

c. In order to ensure the correct precedence relationship in the network diagram the following questions must be answered as every activity is added to the network:

   i. What activities must be completed immediately before this activity can start?

   ii. What activities must follow this activity?

   iii. What activities must occur concurrently with this activity?

   iv. What controls the start?

   v. What controls the finish?

5.2.3 Activities on Software Life Cycle

The Activities in a typical software life cycle revolves around software development, operations and maintenance. For the purpose of this research, the following major activities have been identified:

Data Requirement Analysis (DR)

Physical Requirement Analysis (PR) - building, furniture, electrical installation and so on.

Hardware Requirement Analysis (HR)

Hardware Procurement (HP)

Hardware Installation (HI)

Hardware Testing (HT)

Software Requirement Analysis (SR)

File Design (FD)

Report Design (RD)

Personnel Requirement Analysis (PER)

Program Design (PD)

Program Coding (C)

Program Walkthrough (W)

Program Compilation (CM)

Data Creation (DC)

Data Update (DU)

System Documentation (DOC)

Test Run (TE)

Life Run (LR)

Performance Evaluation (PE)

Maintenance (MT)

It is remarked that, each of the above core activities may be further broken down into components. For instance, system maintenance, may be further broken down into perfective maintenance, corrective maintenance and adaptive maintenance. The network diagram of the software life cycle is depicted in Appendix I

5.2.4 Duration of Activities on the Network

Finding the duration of a job is one of the important aspects of network analysis which must be handled with utmost care. A realistic estimate of jobs duration will enhance the determination of job schedule and will also help in project planning and control which is the ultimate goal of project scheduling. In well established organisations, a lot of facts could be derived from work study and records of past projects and similar projects in other organisations. Another useful method for getting realistic jobs duration estimates is the analytic estimate.

An analytic estimator breaks down a job into its elements and estimates the "work content" and hence the duration of each. An analytic estimator possesses a basic skill in the type of work to be done. This may be supplemented by training in time study. The skill enables him to assess a future job realistically. Time studies taken later, on the job itself, provide a check on his estimate and also give information which can be classified into standard data for future use. Scheduling for software development projects can be viewed from two perspectives.

a. An end date for the release of a computer-based system has already been set. The software developing team is constrained to distribute effort within the prescribed time frame.

b. Rough chronogical bounds have been discussed, but the end-date is set by the software developing team. Effort is distributed to make best use of the resources and an end-date is defined after careful analysis of the software element.

However, the first perspective whereby the end-date has already been pre-determined by the client organization is encountered more frequently [Pressman, 1987].

In order to make a realistic estimate of effort and schedule in software development, operations and maintenance, we took a study of three earlier studies by [Pressman, 1987; Veryard, 1991; and Sommerville, 1992]. They are presented below.

Veryard [1991], reports the result of effort and schedule distributions by phase for medium-sized organic project in Constructive Cost Model (COCOMO) model depicted in Table 5.2.

Table 5.2: Effort and schedule distribution by phase in COCOMO.

Phase	Effort %	Schedule %
Plans and requirement	6	12
Product design	16	19
Detail design Code and Unit test	24 38	55
Integration and test	22	26
	106	110

Commenting on the above estimates (Veryard, 1991) noted that:

a. The above estimates covers the development phases of COCOMO.

b. Plans and requirements are outside the phases because they have to be completed in most cases before reliable estimates can be made.

c. The distribution of both time and effort by phase depends on both application type and project size. The figures above are for medium sized organic project.

Pressman [1987], also presented theoretical estimates of distribution of efforts in software development as follows:

Planning – 2-3%

Requirement Analysis - 10-20%

System design - 20-30%

Coding - 10-20%

Testing/compilation - 30-50%

System maintenance 3-4 times of the system development effort

In another study by [Sommerville, 1992], the following estimate of effort distribution in software development, operations and maintenance were reported:

Table 5.3: Percentage of effort distribution in software project..

ACTIVITY	PERCENTAGE
Problem definition System analysis Software design	40%
Program coding Program walkthrough Program compilation	20%
Data survey and collection Program test run Parallel run of new program with old program	40%

Maintenance was estimated to be 3 to 4 times of effort expended on the above named activities. The network diagram for software development, operations and maintenance presented in Appendix I was drawn with the following assumptions:

a. The time and effort required for the software life cycle from the beginning up to and including performance evaluation is 100 units.

b. The value of each activity represents the estimated percentage of time and effort required to accomplish such activity.

c. Software maintenance is assumed to take 4 times (400%) of the time and effort expended on activities from the beginning of the software life cycle up to and including performance evaluation. Thus, if 100 time units is allocated to activities from the beginning of software life cycle up to and including performance evaluation, then 400 time units would be allocated to software maintenance.

d. Depending on the time available for the development, operations and maintenance of a given software, the time units could be converted to the actual time like hours, days, weeks, months and years.

e. In the use of actual time units, consistency in the definitions must be ensured. For instance, the definition of a week must state clearly whether it includes Saturday, Sunday, Overtime and how many hours makes a working day and so on.

The duration of each activity is labelled on the arc in the network diagram depicted in Appendix I.

5.2.5 Critical Path Calculations

The application of PERT/CPM should ultimately yield a schedule specifying the start and completion dates of each activity. The network diagram represents the first step toward achieving this goal. Due to the interaction among the different activities, the determination of the start and completion times requires special computations. These calculations are performed directly on the network diagram using simple arithmetic. The end result is to classify the activities of the project as critical or non-critical.

An activity is said to be critical if a delay in its start will cause a delay in the completion date of the entire project. A non-critical activity is such that the time between its earliest start and its latest completion dates is longer than its actual duration. In this case, the non-critical activity is said to have a slack or float time.

5.2.6 Determination of the critical path

A critical path defines a chain of critical activities which connect the start and end events of the arrow diagram. In other words, the critical path identifies all the critical activities of the project. The critical path calculations are in two phases. The first phase is called the forward pass where calculations begin from the 'start' node and move to the 'end' node. At each node, a number is computed representing the corresponding event. These numbers are shown in Appendix I in squares (□). The second phase, called the backward pass, begins calculations from 'end' node and moves to the 'start' node (shown in triangles Δ) represent the latest occurrence time of the corresponding event.

Let ES_i be the earliest start time of all the activities emanating from event i. Thus, ES_i represents the earliest occurrence time of event i. If i=1, is the "start" event, then conventionally,

for the critical path calculations, $ES_1=0$. Let $D_{i,j}$ be the duration of activitiy (i,j). The forward pass calculations are thus obtained from the formula

$$ES_j = \max \{ES_i + D_{i,j}\} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{equation (1)}$$

for all defined (i,j) activities where $ES_1=0$, thus, in order to compute ES_j for event j, ES_i for the tail events of all the incoming activity (i,j) must be computed first.

5.2.6.1 Forward Pass

$ES_1 = 0$

$ES_2 = ES_1 + D_{12} = 0 + 6 = 6$

$ES_3 = \text{Max } \{ES_i + D_{i3}) = \text{Max } \{0 + 3, 6 + 6\} = 12 , j = 1,2$

$ES_4 = ES_3 + D_{i4} = 12 + 1 = 13$

$ES_5 = ES_4 + D_{i5} = 13 + 1.5 = 14.5$

$ES_6 = ES_5 + D_{i6} = 14.5 + 0.5 = 15$

$ES_7 = ES_6 + D_{i7} = 15 + 3 = 18$

$ES_8 = ES_7 + D_{i8} = 18 + 5 = 23$

$ES_9 = \text{Max } \{ES_i + D_{i9}) = \{ \text{Max } \{23+6, 18+2\} = 29 \; i=7,8$

$ES_{10} = ES_9 + D_{i10} = 29 + 8 = 37$

$ES_{11} = ES_{10} + D_{i11} = 37 + 12 = 49$

$ES_{12} = ES_{11} + D_{i12} = 49 + 3 = 52$

$ES_{13} = ES_{12} + D_{i13} = 52 + 8 = 60$

$ES_{14} = ES_{13} + D_{i14} = 60 + 15 = 75$

$ES_{15} = \text{Max } \{ES_i + D_{i15}\} = \text{Max } \{60+0, 75+5\} = 80, \; i=13,14$

$ES_{16} = ES_{15} + D_{i16} = 80 + 1 = 81$

$ES_{17} = ES_{16} + D_{i17} = 81 + 10 = 91$

$ES_{18} = \text{Max } \{ES_i + D_{i18}\} = \text{Max } \{81+0, 91+3\} = 94$

$ES_{19} = ES_{18} + D_{i19} = 94 + 2 = 96$

$ES_{20} = ES_{19} + D_{i20} = = 96 + 3 = 99$

$ES_{21} = ES_{20} + D_{i21} = 99 + 400 = 499$

The calculations of the forward pass for all the activities defined in Appendix I are derived by substituting into the equation 1.

5.2.6.2 Backward pass

The backward pass starts from the "end" event. The objective of this phase is to compute LC_i, the latest completion time for all the activities coming into event i. Thus, if i=n is the "end" event, $LC_n = ES_n$ initiates the backward pass. In general, for any node i,

$LC_i = \min \{LC_j - D_{ij}\}$ .....equation (2)

for all defined (i,j) activities. The values of LC_i (entered in the triangles) are determined as follows:

$LC_{21} = ES_{21} = 499$

$LC_{20} = LC_{21} - D_{2021} = 499 - 400 = 99$

$LC_{18} = LC_{20} - D_{1920} = 9 - 3 = 96$

$LC_{18} = LC_{19} - D_{1819} = 96 - 2 = 94$

$LC_{17} = LC_{18} - D_{1718} = 94 - 3 = 91$

$LC_{16} = \text{Max } \{LCj = D_{16j}\} = \text{Min } \{91\text{-}10, 94\text{-}0\} = 81\}$ j=17,18

$LC_{15} = LC_{16} - D_{1516} = 81\text{-}1 = 80$

$LC_{14} = LC_{15} - D_{1415} = 80\text{-}5 = 75$

$LC_{13} = \text{Min } \{LCj - D_{13j}\} = \text{Min } \{80\text{-}0, 75\text{-}15\} = 60$

$LC_{12} = LC_{13} - D1213 = 60\text{-}8 = 52$

$LC_{11} = LC_{12} - D_{1112} = 52\text{-}3 = 49$

$LC_{10} = LC_{11} - D_{1011} = 49 - 12 = 37$

$LC_9 = LC_{10} = D_{910} = 37\text{-}8 = 29$

$LC_8 = LC_9 - D_{89} = 29\text{-}6 = 23$

$LC_7 = \text{Min } \{29\text{-}2, 23\text{-}5\} = 18, \text{Min } \{LCj - D_{7j}\}$ j=8,9

$LC_6 = LC_7 - D_{67} = 18 - 3 = 15$

$LC_5 = LC_6 - D_{56} = 15 - 0.5 = 14.5$

$LC_4 = LC_5 - D_{45} = 14.5 - 1.5 = 13$

$LC_3 = LC_4 - D_{34} = 13-1 = 12$

$LC_2 = LC_3 - D_{23} = 12-6 = 6$

$LC_1 = \text{Min } \{6-6, 8-3\} = 0\ j=2,3\ \text{Min } \{LCj - D_{1j}\}\ j-2,3$

5.2.6.3 Identification of Critical Path activities

The critical path activities is identified by using the results of the forward and backward passes. An activity (i,j) lies on the critical path if it satisfies the following three conditions,

a. $ES_i = LC_i$ .....equation (3)

b. $ES_j = LC_j$.....equation (4)

c. $ES_j - ES_i = LC_j - LC_i = D_{ij}$.....equation (5)

These conditions actually indicate that there is no float or slack time between the earliest start (completion) and the latest start (completion) of the activity. Thus, this activity must be critical. In the arrow diagram these activities are characterized by the numbers in square (□) and triangle (Δ) being the same at each of the head and the tail events and that the difference between the number in square and triangle at the head and tail event is equal to the duration of the activity. The critical activities; and the critical path for the software life cycle depicted by thick bold lines in Appendix I are determined by substituting into the equation as shown below:

$ES_j - ES_i = LC_j - LC_i = D_{ij}$

(1,2) = 6-0 = 6-0 = 6

(1,3) 12=0 = 12-0 = 12

(3,4) 13-12 = 13-12 = 1

(4,5) 14.5-13 = 14.5-13 = 1.5

(5,6) 15-14.5 = 15-14.5 = 0.5

(6,7) 18-15= 18-15 = 3

(7,8) 23-18 = 23-18 = 5

(7,9) 23-18 = 23-18 = 5

(8,9) 29-23 = 29-23 = 6

(9,10) 37-29 = 37-29 = 8

(10,11) 49-37 = 49-37 = 12

(11,12) 52-49 = 52-49 = 3

(12,13) 60-52 = 60-52 = 8

(13,14) 75-60 = 75-60 = 15

(13,15) 80-80 = 80 - 80 = 0

(15,16) 81-80 = 81-80 = 1

(16,17) 91-81 = 91 - 81= 10

(16,18) 91-91 = 91-91 = 0

(17,18) 90-87 = 90-87 = 3

(18,19) 96-94 = 96=94 = 2

(19,20) 99-96 = 99-96 = 3

(20,21) 499-99 = 499 - 99 = 400

It is observed that while all the activities satisfy conditions (a) and (b) for critical activities (1,3), (7,9) (13,15) and (16,18) did not satisfy condition (c). Hence, they are not critical. Note that the critical path must form a chain of connected activities which spans the network from "start" to "end". The critical activities is depicted (⟶) in Appendix II. Thus, the critical path in Appendix II is defined by the following activities.:

((1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10), (10,11), (11,12), (12,13, (13,14), (14,15), (15,16), 16,17), (17,18), (18,19) (19,20), (20,21).

5.2.7 Determination of Float

All critical activities have zero float or slack. In order to determine the float of non-critical activities, we need to define latest start time of activity (i,j) LS_{ij} and the earliest completion time of activity (i,j) EC_{ij} by:

$LS_{ij} = LC_j - D_{ij}$ .....equation (6)

$EC_{ij} = ES_i + D_{ij}$ .....equation (7)

5.2.7.1 Latest Start

The latest start time (LS_{ij}) is calculated by substituting into equation (6):

$LS_{ij} = LC_j - D_{ij}$

$LS_{12} = 6\text{-}6 = 0$

$LS_{13} = 12 - 3 = 9$

$LS_{23} = 12 - 6 = 6$

$LS_{34} = 13 - 1 = 12$

$LS_{45} = 14.4 - 1.5 = 13$

$LS_{56} = 15 - 0.5 = 14.5$

$LS_{67} = 18 - 3 = 15$

$LS_{78} = 23 - 5 = 18$

$LS_{79} = 23 - 2 = 21$

$LS_{89} = 29 - 6 = 23$

$LS_{910} = 37 - 8 = 29$

$LS_{1011} = 49 - 12 = 37$

$LS_{1112} = 52 - 3 = 49$

$LS_{1213} = 60 - 8 = 52$

$LS_{1314} = 60 - 15 = 45$

$LS_{1315} = 80 - 0 = 80$

$LS_{1415} = 80 - 5 = 75$

$LS_{1516} = 81 - 1 = 80$

$LS_{1617} = 91 - 10 = 81$

$LS_{1718} = 94 - 3 = 91$

$LS_{1819} = 96 - 2 = 94$

$LS_{1920} = 99 - 3 = 96$

$LS_{2021} = 499 - 400 = 99$

5.2.7.2 Earliest Completion

The earliest completion (EC_{ij}) is calculated by substituting into equation (7):

$EC_{ij} = ES_i + D_{ij}$

$EC_{12} = 0 + 6 = 6$

$EC_{13} = 0 + 3 = 3$

$EC_{23} = 6 + 6 = 10$

$EC_{34} = 12 + 1 = 13$

$EC_{45} = 13 + 1.5 = 14.5$

$EC_{56} = 14.5 + 0.5 = 15$

$EC_{67} = 15 + 3 = 18$

$EC_{78} = 18 + 5 = 23$

$EC_{79} = 18 + 2 = 20$

$EC_{89} = 23 + 6 = 29$

$EC_{910} = 29 + 8 = 37$

$EC_{1011} = 37 + 12 = 49$

$EC_{1112} = 49 + 3 = 52$

$EC_{1213} = 52 + 8 = 60$

$EC_{1314} = 60 + 15 = 75$

$EC_{1315} = 60 + 0 = 60$

$EC_{1415} = 75 + 5 = 80$

$EC_{1516} = 80 + 1 = 81$

$EC_{1617} = 81 + 10 = 91$

$EC_{1618} = 81 + 0 = 81$

$EC_{1718} = 91 + 3 = 94$

$EC_{1819} = 94 + 2 = 96$

$EC_{1920} = 96 + 3 = 99$

$EC_{2021} = 99 + 400 = 499$

There are two important types of floats: Total Float (TF) and Free Float (FF). The total float TF_{ij} for activity (i,j) is the difference between the maximum time available to perform the activity (LC_j - - ES_i) and its duration (D_{ij}); that is

$$TF_{ij} = LC_j - ES_i - D_{ij} = LC_j - EC_{ij} = LS_{ij} - ES_i$$

The free float is defined by assuming that all the activities start as early as possible. In this case, FF_{ij} for activity (i,j) is the excess of available time (ES_j - ES_i) over its duration (D_{ij}); that is

$$FF_{ij} = ES_j - ES_i - D_{ij}$$

The results obtained from the critical paths calculations together with the floats for the non-critical activities is presented in Appendix III.

5.2.8 **Estimating the Cost of Software Schedule**

Costs are generated by the consumption of resources, and it follows that the best estimate of the cost of a job will be derived from work measurement described in analytical estimate. In a typical organisation using PERT/CPM, special forms are designed for cost estimation. The cost aspect is included in project scheduling by defining the cost-duration relationship for each activity in the project. Costs are defined to include direct elements only. Indirect costs such as administrative or supervision costs are not included. Figure 5.2 shows a typical straight-line relationship between cost and duration used with most projects.

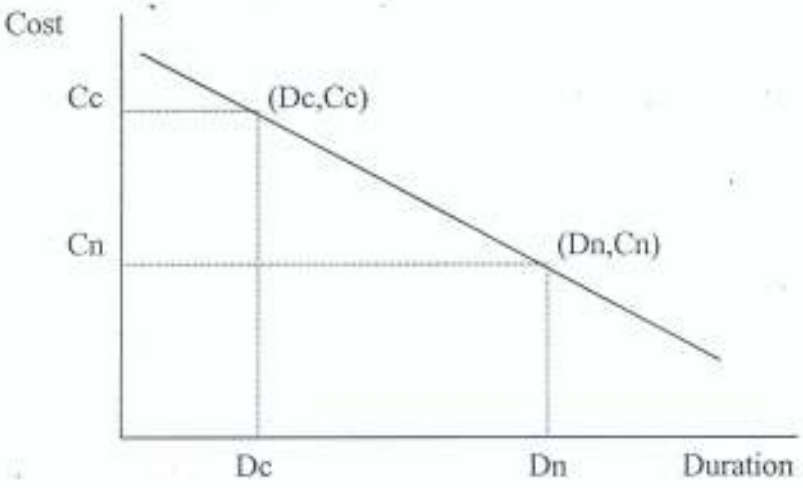


Figure 5.2: Cost-Duration curve.

The point (Dn,Cn) represents the duration 'Dn' and its associated cost 'Cn' if the activity is executed under normal conditions. The duration 'Dn' can be reduced by increasing the allocated resources and hence by increasing the direct costs. There is a limit, called crash time, beyond which no further reduction in the duration can be effected. At this point any increase in resources will only increase the costs without reducing the duration. The crash point is indicated in Figure 5.2 by the point (Dc, Cc). The linear relationship between cost and duration depicted by Figure 5.2 is a simplifcation of the real life cost-duration relationship as a non-linear relationship would complicate the calculations. The straight-line relationship could be improved upon by a piecewise linear curve as shown in Figure 5.3.

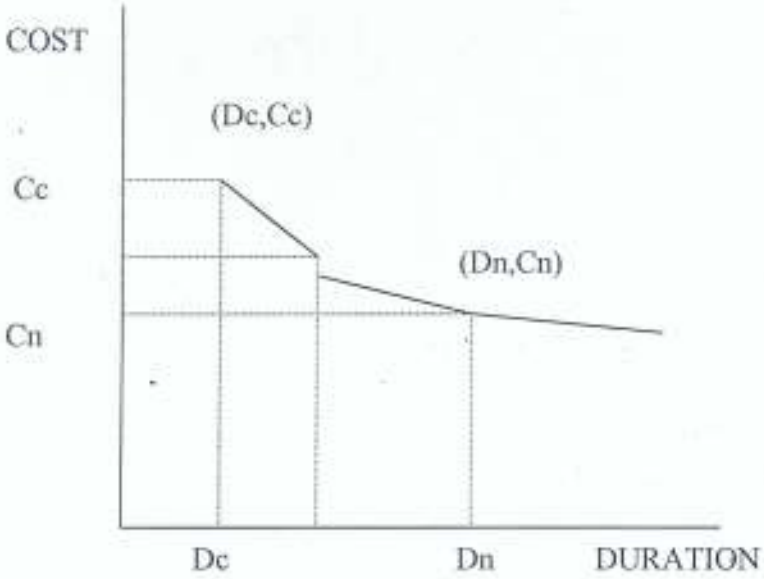


Figure 5.3: Piecewise Cost-Duration curve

The piecewise cost-duration curve in Figure 5.3 is derived by breaking down each activity into a number of sub-activities each corresponding to one of the line segments. The slopes of the line segments increases as one moves from the normal point to the crash point. This condition must be satisfied in order to ensure the validity of the approximation.

After defining the cost-time relationship, the activities of the project are assigned their normal durations. The corresponding critical path is then computed and the associated (direct) costs are recorded. The next step is to consider reducing the duration of the project. Since such a reduction can be effected only if the duration of a critical activity is reduced, attention must be paid to such activities alone. In order to achieve a reduction in the duration at the least possible cost,

one must compress as much as possible the critical activity having the smallest cost-time slope. The amount by which an activity can be compressed is limited by its crash time.

When an activity is compressed, a new schedule with perhaps a new critical path is obtained. The cost associated with the new schedule must be greater than with the immediately preceding one. The new schedule must now be considered for compression by selecting the (uncrashed) critical activity with the least slope.

The procedure is repeated until all critical activities are at their crash times. The final result of the above calculations is a cost-time curve for the different schedules and their corresponding costs. A typical curve is shown by a solid line in Figure 5.4. This represents the direct cost only. The indirect cost is depicted by dotted line in Figure 5.4 and increases as the project duration increases. The sum of the direct and indirect costs gives the total cost of the project. The optimum schedule corresponds to the minimum total cost.

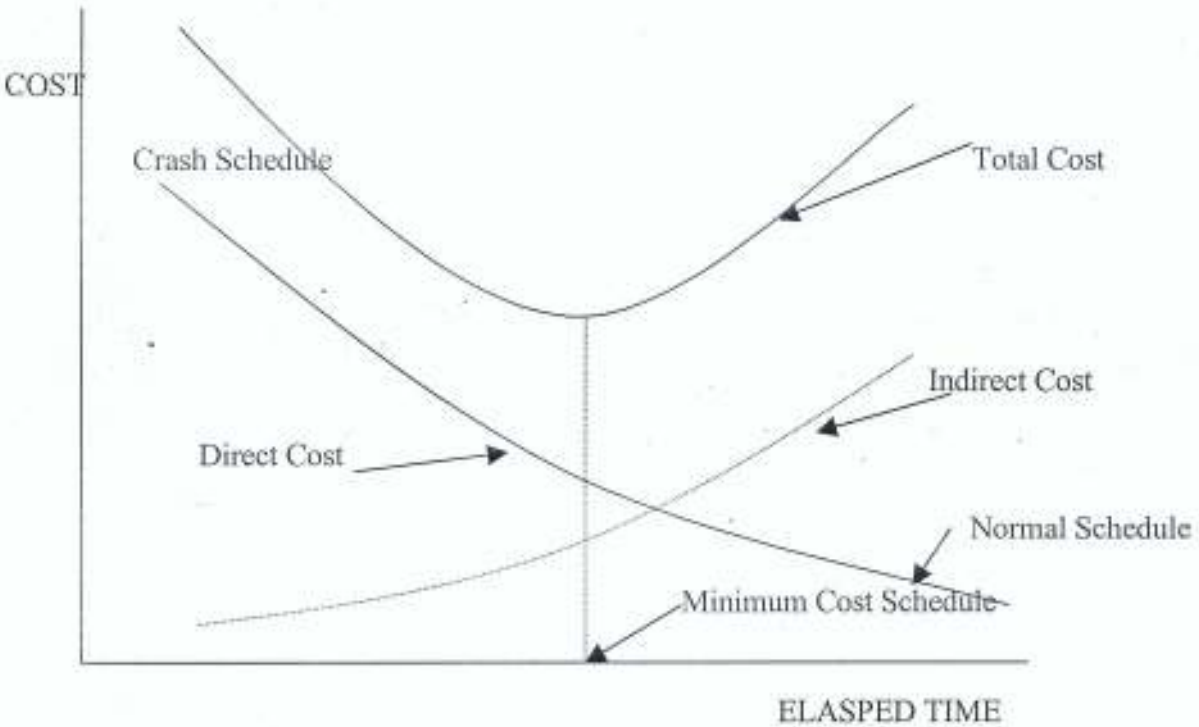


Figure 5.4: The relationship between Total-Cost, Direct Cost, Indirect Cost and Minimum Cost.

5.3 Monitoring and Evaluation of Software Life Cycle

For the purpose of monitoring and evaluation of software development, operations and maintenance, the metrics described earlier in this chapter are modelled into a relationship form. The general form of a relation is given by $R[a_1, a_2, a_3 \ldots a_k, a_{k+1} \ldots a_n]$

where R represents the name of the relation or file, $[a_j]$, $j=1, 2 \ldots n$ represents the attributes, properties or decision variables. The set of relations that are considered in this research are the following:

1. Activity-file [code-of-activity , description-of-activity]
2. Cost-file [code -of–activity, estimated-cost, actual-cost]
3. Duration-file[code-of-activity, estimated-duration, actual-duration]
4. Quality-code-file[code-of-quality-factor, description-of-quality-factor]
5. Software-quality-file[code-of-quality-factor, metric, score-obtainable, actual-score-obtained]

The procedure for evaluation of software life cycle involves:

a. The comparison of the estimated duration (schedule) of development, operations and maintenance activities with the actual duration of these activities.

b. Comparison of the estimated cost of software development, operations and maintenance activities with their actual cost.

a. Drawing of inferences based on the outcome of the comparisons made in steps (a) and step (b).

For instance, in case of the monitoring and evaluation of cost, the inference will indicate whether there is cost overrun, cost under-run or break-even. It will also give the actual difference and percentage difference between estimated cost and actual cost.

The monitoring and evaluation of the duration and cost of activities in the software life cycle is conceptualized in Figure 5.5 and Figure 5.6 respectively.

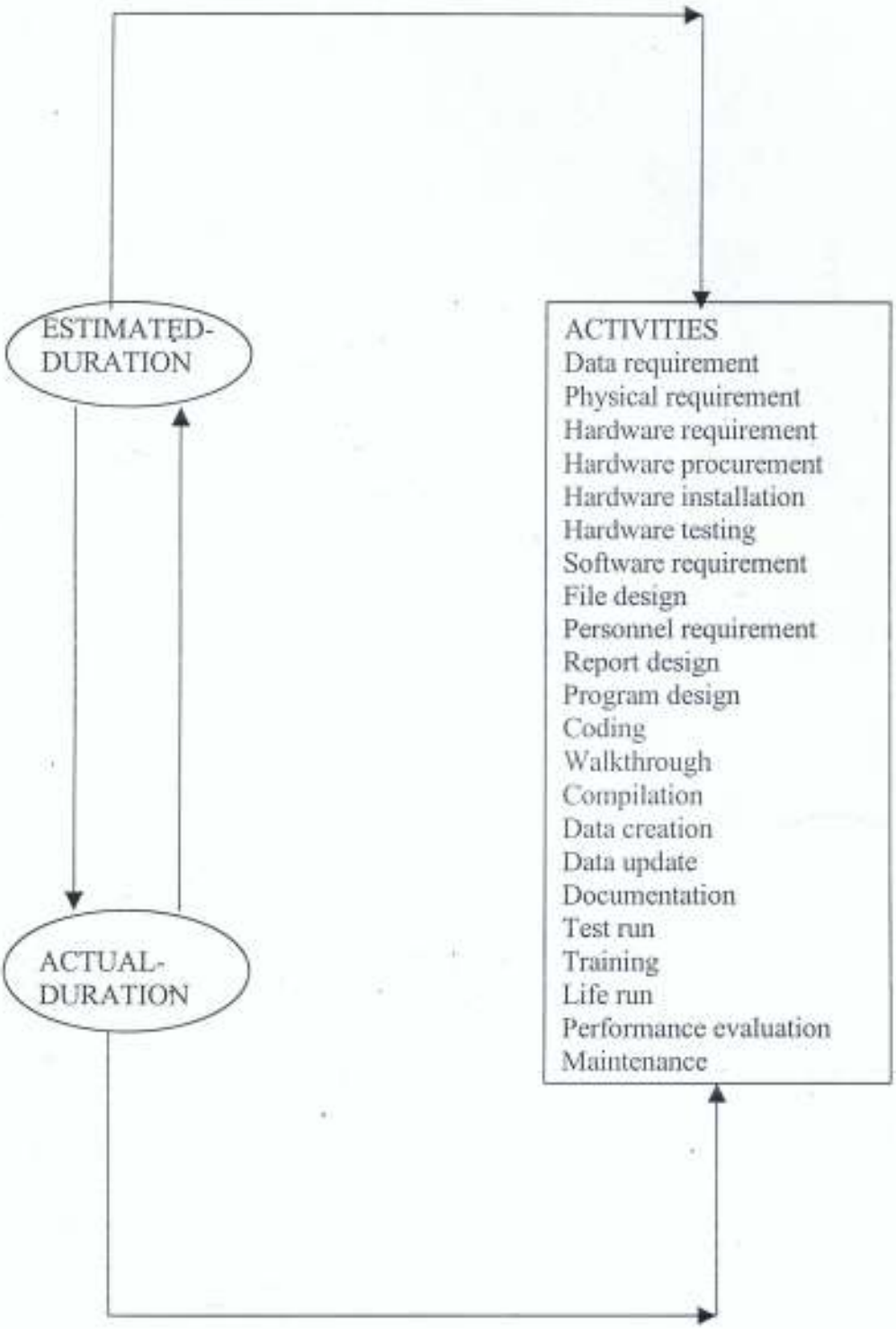


Figure 5.5: Matching of Estimated-duration with Actual -duration

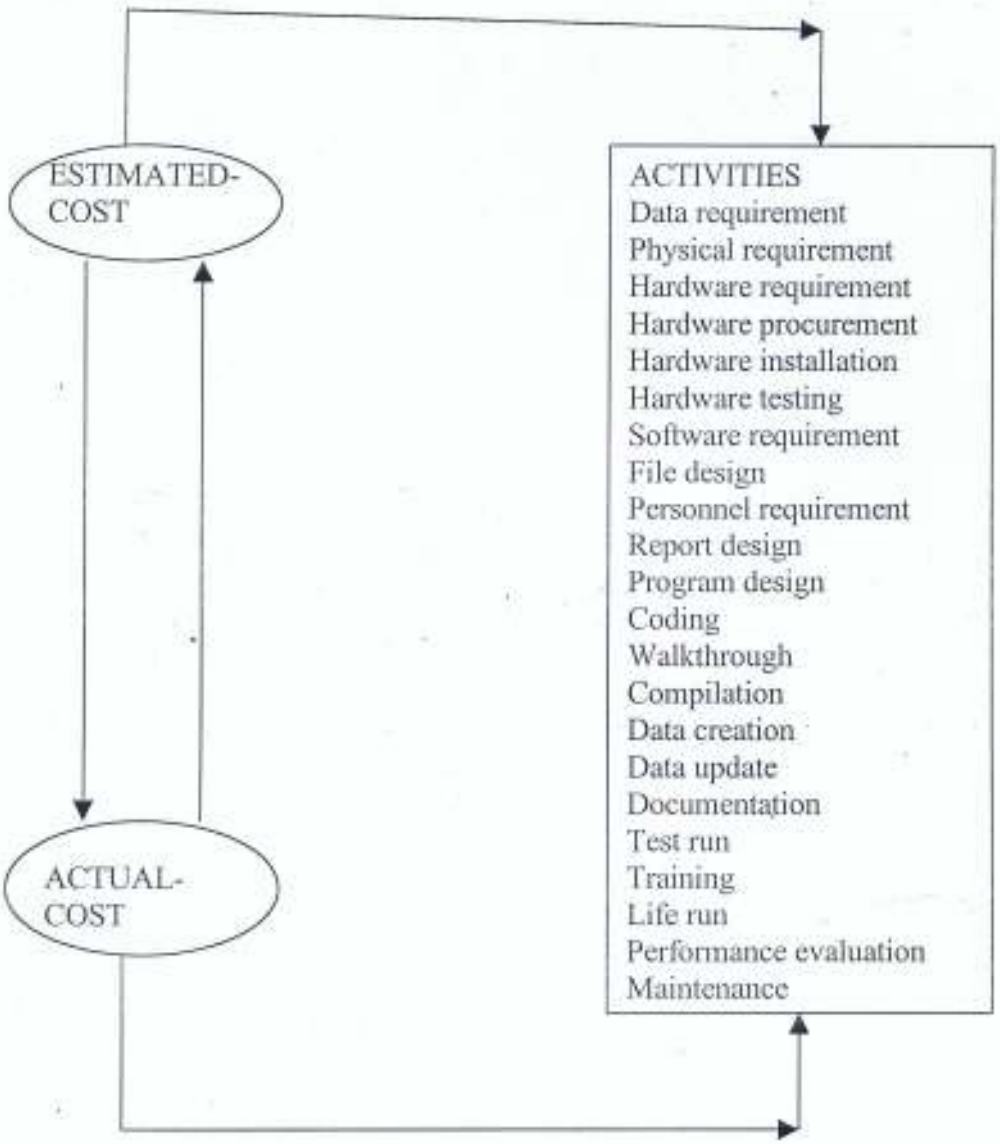


Figure 5.6: Matching of Estimated-cost with Actual-cost.

5.4 Evaluation of Software Quality Factors

Performance evaluation of software quality involves making a quantitative assessment of software quality factors using software quality metrics. This is achieved by grading each quality metric on a "scale" on which the maximum score (or score obtainable) is 5 and the minimum score is 1. A score is awarded based on the judgement of the human evaluator on how a given software meets or possesses the quality attribute under consideration. The overall evaluation of software quality is determined by taking the sum of the scores of all software quality factors as depicted in the following equation:

$$SQ = \sum_{i=1}^{n} x_i$$

where SQ is software quality, x_i refers to the various software quality factors and n is the total number of the software quality factors.

Finally, based on the result of the evaluation procedure, an inference of the extent to which the software under consideration meets the quality of an ideal software is made. The system rates the software quality either as poor, average, good, very good or excellent. The evaluation of a software quality factor is conceptualized in Figure 5.7

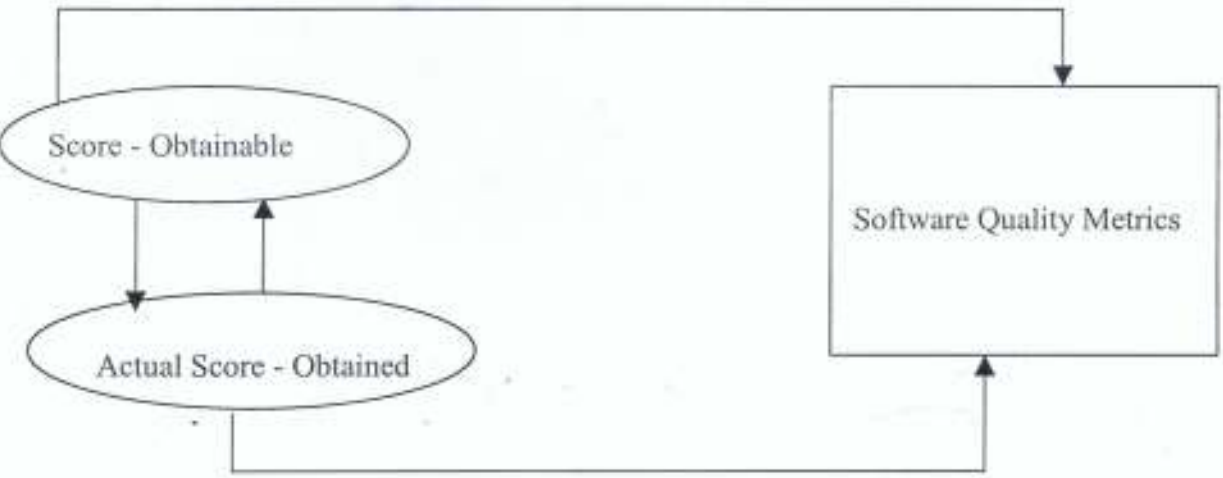


Figure 5.7: Evaluation of a software quality factor.

The results derived from the evaluation of each software quality factor serve as an input to the final or terminal module. The results are designated as $\{P_k\}$, k=1,2,3,4,5,n+1, and represents the actual score for each software quality factor like correctness, reliability, efficiency, integrity, maintainability, flexibility, testability and so on. The $\{P_k\}$ serve as input parameters to the terminal module, which produces the difference and percentage difference between expected total score and total actual score of the metrics for each software quality of the software being evaluated. An inference, of whether the quality of the software is poor, average, good, very good or excellent is also made. The logical relationships of the systems modules is presented in Figure 5.8

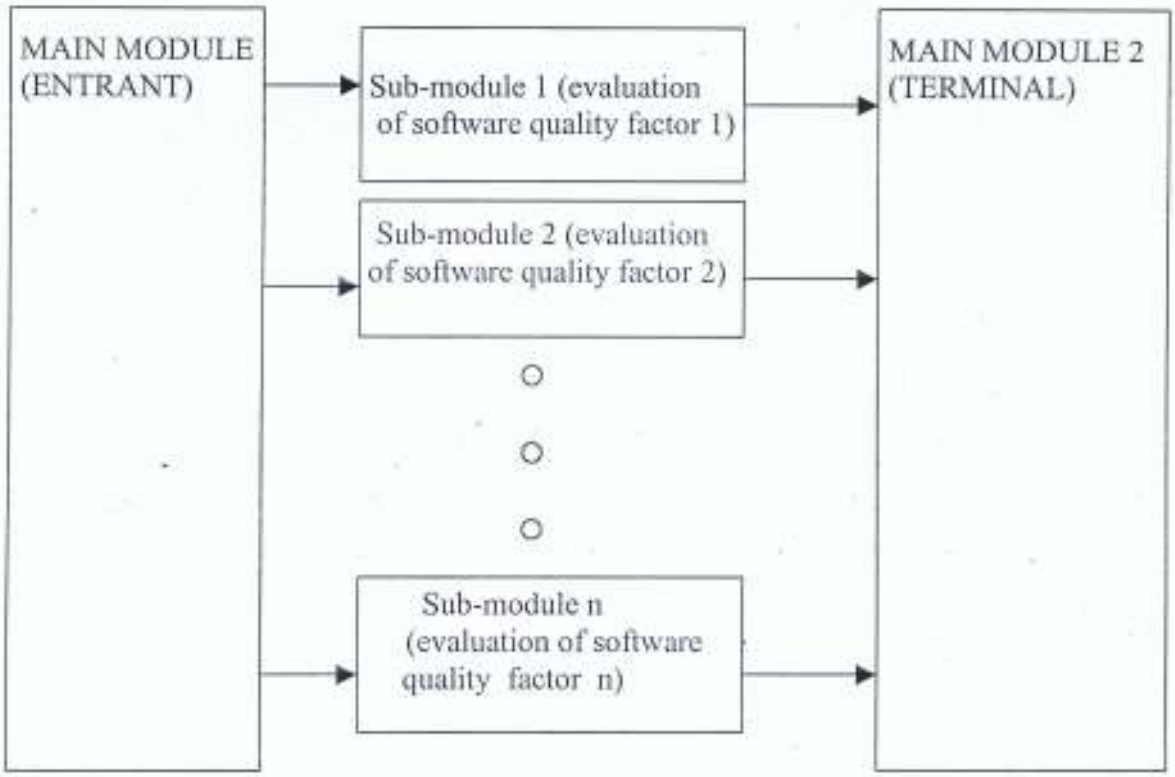


Figure 5.8: Logical relationships of the evaluation of software quality factors

5.5 System Implementation

The Computer Aided System (CASMESDOM) is implemented using Paradox 4.5 Relational Database Management System (RDBMS). The Paradox 4.5 RDBMS environment allows the development of a user friendly, interactive and intelligent software package. The top-down design adopted in the development and implementation of CASMEDOM enables the system through menu sessions and dialogue sessions.

5.5.1 Dialogue and Menu Sessions

The first dialogue session begins when user types "CASMESDM" at the C:\> prompt of the computer. This opens the login screen depicted in Figure 5.9. The user name and Password are entered, subject to the verification and validation, authorization is granted and the system main menu depicted in Figure 5.10 is displayed on the screen.

YOU ARE PLEASE WELCOME TO CASMESDOM
Please, enter the following: User 's Name: Password:

Figure 5.9: Login screen.

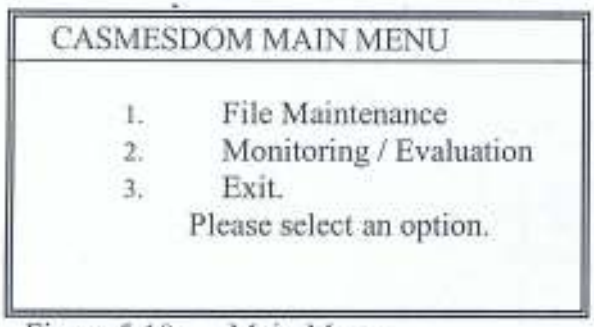


Figure 5.10 Main Menu.

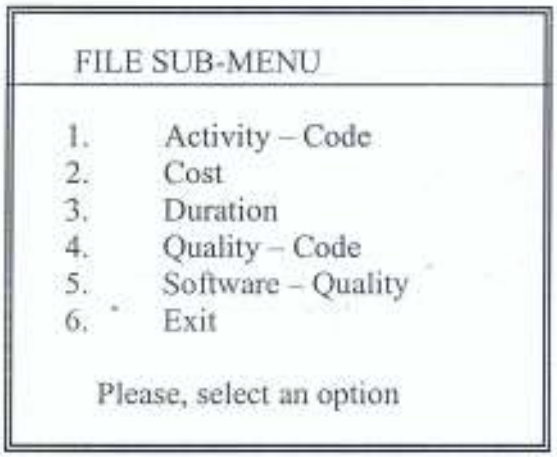


Figure 5.11: File Sub – menu

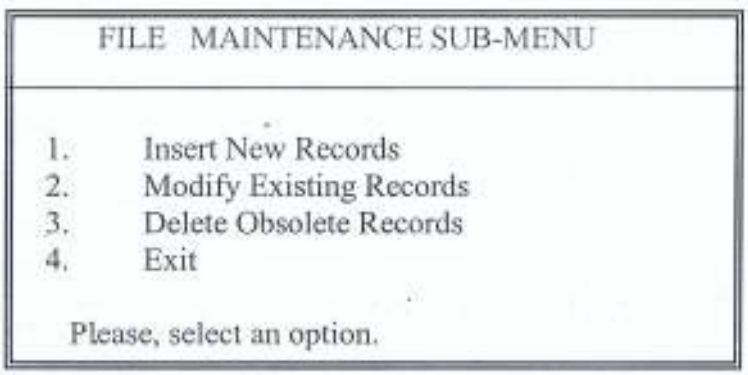


Figure 5.12: File Maintenance Sub – menu

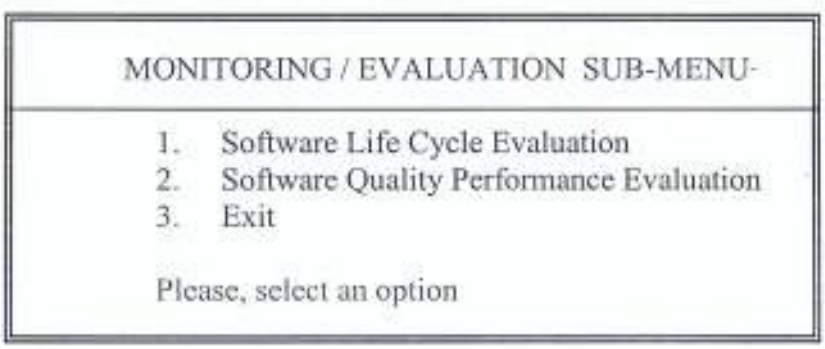


Figure 5.13: Monitoring / Evaluation Menu.

The first menu session begins when the option "File Maintenance" is selected in Figure 5.10. A file maintenance can be carried out by an authorized human evaluator or monitor, hence, the access right of the human evaluator or monitor has to be verified and validated. If access right is granted, the file sub–menu depicted in Figure 5.11 is displayed on the screen. The selection of a file will cause the sub – menu depicted in Figure 5.12 to be displayed on the screen. The selection of any of the options in Figure 5.12 will cause the form or screen view of the file selected in Figure 5.11 to be displayed on the screen.

The second menu session begins when the option "Monitoring /Evaluation" is selected in Figure 5.10. Immediately the selection is made, the submenu depicted in Figure 5.13 is displayed on the screen. Suppose the option "Software Life Cycle" is selected in Figure 5.13, the software life cycle duration and software life cycle cost monitoring evaluation forms are displayed one after the other for the human monitor to supply the information concerning the actual duration or actual cost of the activities involved in the software development, operations and maintenance. Each activity and its estimated duration or cost is displayed one after the other until all the activities that make up the software life cycle is displayed. As soon as the human monitor finishes the dialogue session, an inference procedure is activated which produces the report of the monitoring or evaluation of all the activities. The system allows the human monitor or evaluator to send the report to the screen for the purpose of viewing, file for future use or printer for hard copy production. The relational form of the output reports are given as follows:

Duration [activity-description, estimated-duration, actual-duration, absolute-difference, % - difference, inference].

Cost [activity-description, estimated–cost, absolute-difference, %-difference, inference].

Each report is followed by a key giving the basis or explanation for the inferences made.

Suppose the option "Software Quality Performance Evaluation" is selected from Figure 5.13, a dialogue session is invoked during which the human evaluator uses his own judgement to give the actual score of the software quality metrics. The screen view of software quality evaluation form is presented in Figure 5.14.

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION

OF SOF TWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE

SOFTWARE QUALITY EVALUATION FORM

RECORD#: [ ][ ][ ]

ATTRIBUTE-DESCRIPTION: [ ]

SCORE-OBTAINABLE: [ ]   ACTUAL-SCORE: [ ]

TYPE THE NUMBER CORRESPONDING TO YOUR ASSESSMENT OF THE SOFTWARE QUALITY UNDER CONSIDERATION

1. POOR
2. AVERAGE
3. GOOD
4. VERY GOOD
5. EXCELLENT

Figure 5.14: Software Quality Evaluation Form

Once the assessment of all metrics associated with each quality factor is completed the report is automatically "fed" into the final or terminal module, where the total-actual-score of each software quality factor is compared with its total-score-obtainable. The relational form of the output report is presented below:

Software-quality [code-of-quality-factor, total-score-obtainable, total-actual-score-obtained, total-absolute-difference, %-difference, inference].

The report is followed by a key giving the basis or explanation for the inferences made. The report could be viewed on the screen, stored in a file for future use or sent directly to the printer for hard copy production.

CHAPTER SIX

CASE STUDY

6.1 Introduction

The Federal University of Technology, Akure (FUTA), Payroll System is an in-house software package developed for the processing of FUTA staff salaries and wages. The payroll system was developed between December 1997 and January, 1998 The payroll system was developed by the FUTA Computer Centre for the Bursary Department. Hence, the project development team consists of the staff from both departments. Systems analysis, design, coding, compilation, test-running, debugging were carried out by the staff of the Computer Centre. Data collection, data entry, data cleaning, data verification and validation were jointly carried out by the staff of both departments. Several systems develpoment meetings were held by the staff of the two departments in the course of the project.

Table 6.1: Composition of the project development team

Category	Number
Project Director	1
Operation Manager	1
Systems Programmers	4
Data Entry Supervisors	2
Data Entry Clerks	5
Data Analysts	3

Before the advent of the new payroll system, a payroll system developed and coded in COBOL had been in operation since 1993. The need to develop a new payroll system arose due to some problems associated with the COBOL based payroll system. Some of these problems include:

a. The breakdown of the COBOL based package due to virus attack and the inability to get the appropriate COBOL compiler to compile the backup source code.

b. The package is not interactive, user-friendly and menu-driven.

c. The package could not be easily maintained.

d. The Package was based on conventional file processing which is now becoming obsolete.

6. 2 Objectives of FUTA Payroll System

The objectives of the FUTA new payroll system is as follows:

a. To develop a menu-driven, user-friendly, interactive and intelligent software package that would be easy to maintain.

b. To develop a package that would be easy to maintain.

c. To develop a package that would produce report at a faster rate.

d To produce neater output reports.

e. To change from the conventional file processing system to a relational database management system.

6.3 Systems Development

6.3.1 Systems Analysis and Design

The analysis and design of the FUTA Payroll was carried out in 1997. In systems analysis and design the objectives of the FUTA Payroll system as stated in section 6.2 were taken into consideration. The issue of programming language, output reports, input data, processes, hardware and other inputs were carefully analysed. Alternative approach to the problem were considered before a final choice was made. In selecting the option, factors such as management desire, ability to integrate with existing systems, technical feasibility, critical organizational need and capacity for the organisation to implement the project, were taken into consideration.

In designing the new payroll system the following activities were carried out:

a. Definition of output requirements.

b. Specification of input layouts

c. Development of the overall system logic

d. Identification of files, data volumes, frequency of updating and so on.

e. Identification of the relationships between input and output files.

f. Identification of computer programs and manual procedures required.

g. Production of detailed plan of implementation.

h. Identification of different modules and sub-modules and the relationship between them.

6.3.2 Program Coding

The program was coded in Paradox Relational Database Management System environment using Paradox Application Language (PAL) version 4.0. PAL have facilities for developing an interactive, user friendly and menu driven software package. The query-by-example facility of paradox enhances the processing of query transactions which involve several tables.

6.3.3 Program Compilation

The PAL has a powerful tool called Debugger. This tool can be used to test scripts (programs) and find errors. With the debugger you can:

a. Find out the cause of an error.

b. Evaluate expressions interactively

c. Check values of variables and arrays and save them in a file.

d. Insert additional commands during script play.

e. Determine where you are, within nested levels of scripts and procedures.

f. Return from nested scripts or procedures.

g. Step through command execution, one command at a time.

h. Skip individual commands in a script.

i. Instantly enter the script editor or your own editor to change the script being tested.

The debugger is part of PAL's integrated environment for prototyping, developing, and testing scripts and applications [Borland, 1988]. With the aid of the PAL debugger, the compilation of the programs was made easier and faster.

6.4 Systems Operation

6.4.1 Data Collection and Data Entry

The University Personnel and Bursary departments maintains an accurate record of staff. Records of staff personal data, staff emolument and other financial records like loans, advances, taxes and so on were collected from these departments. Data entry, data cleaning, verification and validation were

jointly carried out by the staff of the Computer Centre and the Busary department. Since, they were familiar with the manual and the COBOL based processing of the Payroll, the task of validation and verification was accomplished with better confidence and accuracy.

6.4.2 System Test Run

With the real data of employees the programs were test run. Sample reports were generated to ascertain the accuracy and consistency of the programs.

6.4.3 Parallel Run and Full Implementation

After several trial runs the program was put into full operation by the end of January, 1998.

6.5 Systems Maintenance

Most of the maintenance work on FUTA Payroll were perfective. From time to time there were requests from the Bursary department to generate a new set of reports which were not in the original design. Also, the Harmonised Tertiary Institutions Salary Structure (HATISS) introduced by the Federal Government in September, 1998, led to the introduction of new set of allowances which had to be included in the output report. In January, 1999, there was an amendment to the HATISS. These changes were also effected on the payroll system.

The maintenance work was made easy by the modular nature of the design of the payroll system. One module could be amended without significantly disrupting other modules. Also, the fact that the software maintenance is being carried out by the personnel that were involved in the system development and operations helped in effecting necessary changes with relative ease.

6.6 Monitoring and Evaluation of Software Life Cycle

The software life cycle activities depicted in Appendix I using PERT/CPM project scheduling method was used for the estimation of the duration of each software life cycle activity. The total number of days available for the project was fifty-two days. The duration of each activity was derived by using the percentage of the duration of each activity presented in Appendix I. The estimated duration (D_{ij}), earliest start (ES_i), earliest completion (EC_{ij}), latest start (LS_{ij}), latest completion (LC_j), total float (TF_{ij}) and t he free float (FF_{ij}) are presented in Appendix IV. The

actual values of the duration of the activities (D_{ij}) observed during the development , operations and maintenance of FUTA Payroll System is presented in Appendix V. The estimated duration and actual duration of activities presented in Appendix IV and Appendix V respectively were evaluated using CASMESDOM. The transcript of the output report is presented in Appendix VII.

6.7 Software Quality Performance Evaluation

In order to evaluate the quality of the FUTA Payroll System, the metrics that affect each software quality factor were "graded" on a scale where the maximum score or score obtainable is "5" and the minimum score is "1". The actual score of the metrics of quality factors of FUTA Payroll System is presented in Appendix VI. The actual score is subject to the assessment of the human evaluator. The report of the evaluation of each quality factor is "fed" into the terminal module where the overall assessment of the software quality is made. The transcript of the output report of the performance evaluation of FUTA Payroll System is presented in Appendix VIII and Appendix IX.

CHAPTER SEVEN

CONCLUSION AND RECOMMENDATION

7.1 Conclusion

Software costs today, represents a significant percentage of computer-based system. Consequently, a large cost estimation error or cost escalation can make a difference between profit and loss. Also, maintenance cost now constitutes up to 70% of software project cost [Pressman, 1987; Sommerville, 1992]. This calls for proper monitoring and evaluation of all the phases and activities involved in software projects development, operations and maintenance. Monitoring help to detect any flaw or shortcoming at every phase of the software development, operations and maintenance with a view to introducing corrective measures. Evaluation on the other hand, is carried out at the end of the project. A good software evaluation exercise would assist in planning, and implementation of future software projects.

In this thesis, we adopt the use of Project Evaluation and Review Technique (PERT) and the Critical Path Method (CPM) for the estimation of software metrics such as duration and cost. Also, McCall's software quality factors and quality metrics (McCall, 1977) were used to derive the data used for software quality performance evaluation.

A Computer Aided System (CASMESDOM) was designed and implemented. The main objective is to build a system that would assist software developers, operators and maintenance crew to properly monitor and evaluate software development, operations and maintenance. This would enable developers to correct flaws or discrepancies. This helps in reducing escalation of cost and enhances conformity of software to expected qualities. CASMESDOM was implemented using the data derived from PERT/CPM and McCall's software quality metrics.

CASMESDOM is developed and implemented on an IBM compatible micro computer running on MS-DOS, version 6.22. The program coding was done using Paradox Application Language (PAL) version 4.0, with its powerful user friendly and interactive features. The design of CASMESDOM is such that most selections are made from menu selections and when users need to

type values, such values are in most cases, a single character or a few key strokes. The data entry dialogue screens have been made attractive so as to reduce the boredom associated with data entry and editing. CASMESDOM is capable of doing the following:

a. Monitoring every phase and activity in software development, operations and maintenance and generating reports and using forward chaining to match achievement with targets, and giving assessment of how far the targets were met:

b. Evaluation of the entire process of software development, operations and maintenance and using forward chaining to match achievements with targets, and giving an assessment of how far the targets were met.

b. The use of PERT/CPM for software project scheduling and estimation of cost of software development, operations and maintenance.

c. The use of McCall's software metrics for software quality performance evaluation.

7.2 Recommendation

This research has a number of limitations which could be a subject for further research.

a. The issue of a universally acceptable method of estimating software project cost, time schedule and other metrics is still a matter of controversy. Further research is recommended in the area of evolving a universally acceptable technique for estimating software metrics like cost, duration and so on.

b. The case study of FUTA Payroll System does not fully reflect the full potentials and capabilities of the system. It is recommended that CASMESDOM be implemented on larger software packages such as FUTA Personnel System, FUTA Asset Valuation System or FUTA Library System.

REFERENCES

Akinyokun et al [1985]. Bidirectional Mapping Between a User Orientated Conceptual Schema and a Target Schema: The ACS. Proceedings of Fourth British National Conference on Database System. Published by Cambridge University Press.

Akinyokun, O.C. [1988]. A Framework for Computer Aided Investigation of Crime in Developing Countries. Journal of the Information Technology for Development Vol.3 No.2 Oxford University Press.

Akinyokun, O.C. and Adeniji, O.A. [1991]. Experimental Study of Intelligent Computer Aided Medical Diagnosis and Therapy of Tropical Diseases. Journal of the Institute of Mathematics and Computer Sciences, India, Vol. 2. No. 2.

Akinyokun, O.C. [1993]. Experimental Study of Value Added Tax (VAT) Database Administration. Proceedings of 23rd Annual Conference of Senior Staff of Federal Inland Revenue Service, Lagos.

Akinyokun, O.C. and Arekete, S.A. [1996]. Case Study of Knowledge Bases Server for Valuation of Assets. Ife Journal of Technology, Vol. 6, No. 1; Obafemi Awolowo University Press, Ile - Ife.

Akinyokun; O.C. and Uzoka, F.M.E. [1999]. A prototype of Information Technology Based Human Resources System. International Journal of the Computer, Internet and Management, Volume 7, No. 3, December, 1999 . Published by Computer Society, Internet Society and Engineering Society of Thailand.

Albrecht, A.J. [1979]. Measuring Application Development Productivity. Proceedings of IBM Application Development Symposium,Montrey, CA.

Albrecht, A.J.and Gaffney, J.E.[1983]. Software Function, Some Lines of Code and Development Effort Prediction: A Software Science Validation. IEEE Transcations in Software Engineering.

Arthur, L.J. [1985]. Measuring Programmer Productivity and Software Quality, Wiley–Interscience.

Arthur, J.R. [1988]. Software Evolution.. New York, John Wiley and Sons.

Basili, V.,and Zelkowitz, M. [1978]. Analysing Scale Software Development. Proceedings 3rd International Conference on Software Engineering. IEEE, pp. 116 - 123.

Basili,V. [1980]. Models and Metrics for Software Management and Engineering. IEEE Computer Society Press.

Biezer, B. [1983]. Software Testing Techniques, Van Nostrand Reinhold.

Boehm, B.W. [1981]. Software Engineering Economics. Englewood Cliffs NJ, Prentice-Hall.

Borland [1992]. PAL Users Guide and Reference Manual. Borland International, Scotts Valley, U.S.A.

Collins, G. and Blay, G. [1982]. Structured Systems Development Techniques: Strategic Planning to Systems

Curtis, W. [1979]. Measuring The Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics, IEEE Transaction in Software Engineering, Vol. 5. March, 1979.

Curtis, B., Sheppard, S.B. and Charm, P.T. [1979]. Stronger Predictions of Programmer Performance by Software Complexity Metrics. IEEE Proceedings of Fourth International Conference on Software Engineering, Munich.

Esterling, R. [1980]. Software Manpower Costs: A Model. Datamation.

Gallaire, H. [1985]. Artificial Intelligence and Industry. European Computing Research Centre. Ellis Horwood Series in Artificial Intelligence ed. Campbell, J.

Collings, G. and Blay, G. [1982]. Structured Systems Development Techniques: Strategic Planning to System Testing.

Halstead, M. [1977]. Elements of Software Science, North Holland. Harris [1986]

James, M. and Carma, M. [1983]. Software Maintenance: The Problem and Its Solution. John Wiley

and Sons, New York.

Jensen, R. and Tonies, C. [1979]. Software Engineering. Englewood Cliffs, NJ: Prentice-Hall Inc.

Kayode, M.O. [1985]. The Art of Project Evaluation. Ibadan University Press.

Lientz, B.P. and Swamson, E.B. [1980]. Software Maintenance Management. Reading MA: Addison - Wesley.

Lipow, M. [1979]. On Software Reliability. IEEE Transactions on Reliability R-28, August, 1979, Special Issue on Software Reliability.

Lipow, M. [1982]. Number of Faults Per Line of Code. IEEE Transactions on Software SE – 8.

McCall, J.P. and Walters, G. [1977]. Factors in Software Quality 3 vols., NTS AD-A049-014, 015,055.

McCabe, T. [1976]. A Software Complexity Measure. IEEE Transaction in Software Engineering.

McKee, J.R. [1984]. Maintenance as a Function of Design; In Proceedings 1984 AFIPS National Computer Conference.

Mills, H. [1976]. Software Development. IEEE Transactions on Software Engineering, Vol. SE-2, No. 4.

Nelson, E.A. [1967]. A Management Handbook for the Estimation of Computer Programming Costs. System Development Corporation, TM-3225/000/01, Santa Monica California

Norden, P. [1980]. Useful Tools for Project Management. Software Cost Estimating and Life Cycle Control. IEEE Computer Society Press.

Pressman, R.S. [1987]. Software Egineering: A Practioner's Approach. McGraw-Hill International Editions. Computer Science Series.

Putnam, L. [1978]. A General Empirical Solution to the Macro Software Sizing and Estimating Problem. IEEE Transaction in Software Engineering Vol. 4. No. 4.

Riggs, J. [1981]. Production Systems Planning, Analysis and Control, 3rd Edition, Wiley.

Rubin, H.A. [1983]. Macro-Estimation of Software Development Parameters: The Estimacs System, Softfair Proceedingss, IEEE July, 1983.

Rubey, R.J., Dana, J.A. and Biche, P.W. [1975]. Quantitative Aspects of Software Validation. IEEE Transactions on Software Engineering SE-1.

Sommerville, I. [1992]. Software Engineering. 4th Edition, Addison Wesley Publishing Company.

Taha, H.A., [1976]. Operations Research an Introduction. Macmillan Publishing Co. Inc.

Thayer, T.A. [1976]. Software Reliability Study. Rome Air Development Center, Report No. RADC - TR .

Veryard, R. [1991]. The Economics of Information Systems and Software. Butterworth – Heinemann Ltd.

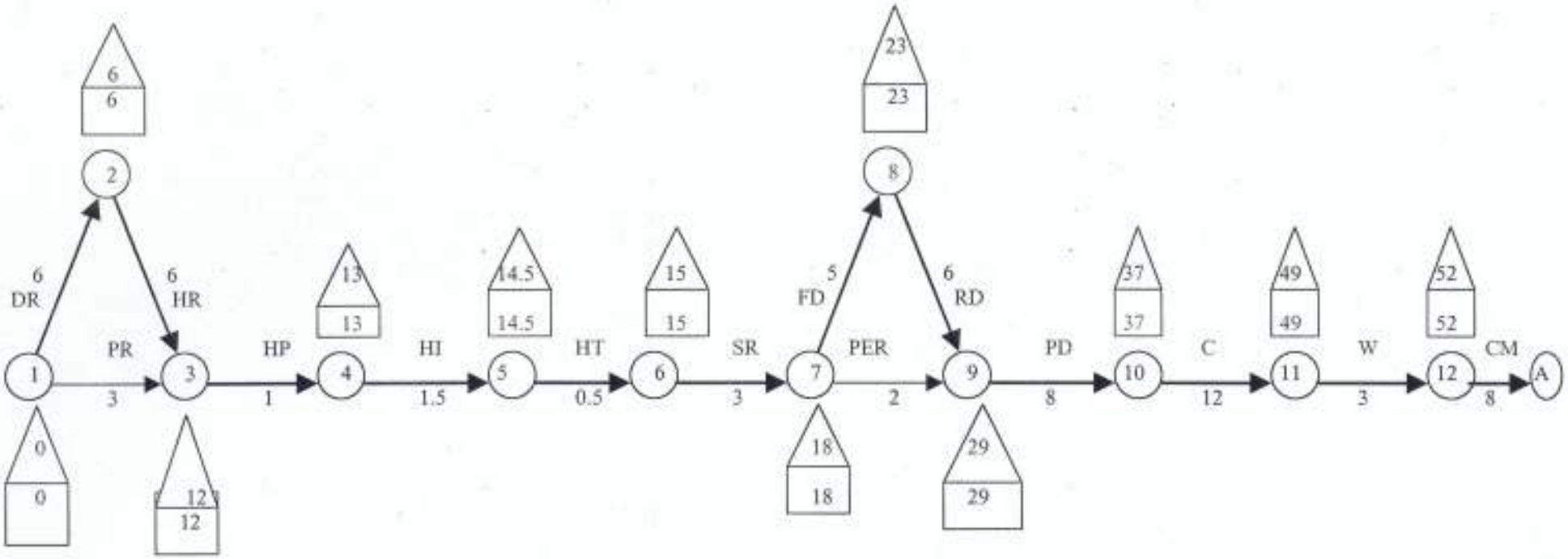
Vic, C.R. [1983]. Handbook of Software Engineering. Edited by C.R. Vic and C.V. Ramamoorthy.

Waltson, C. and Felix, C. [1977]. A Method for Programming Measurement and Estimation, IBM Systems Journal Vol. 16 No. 1, 1977.

Weiwurm, G.F. and Zagorski, H.J. [1965]. Research into Management of Computer Programming: A Transitional Analysis of Cost Estimation Techniques.

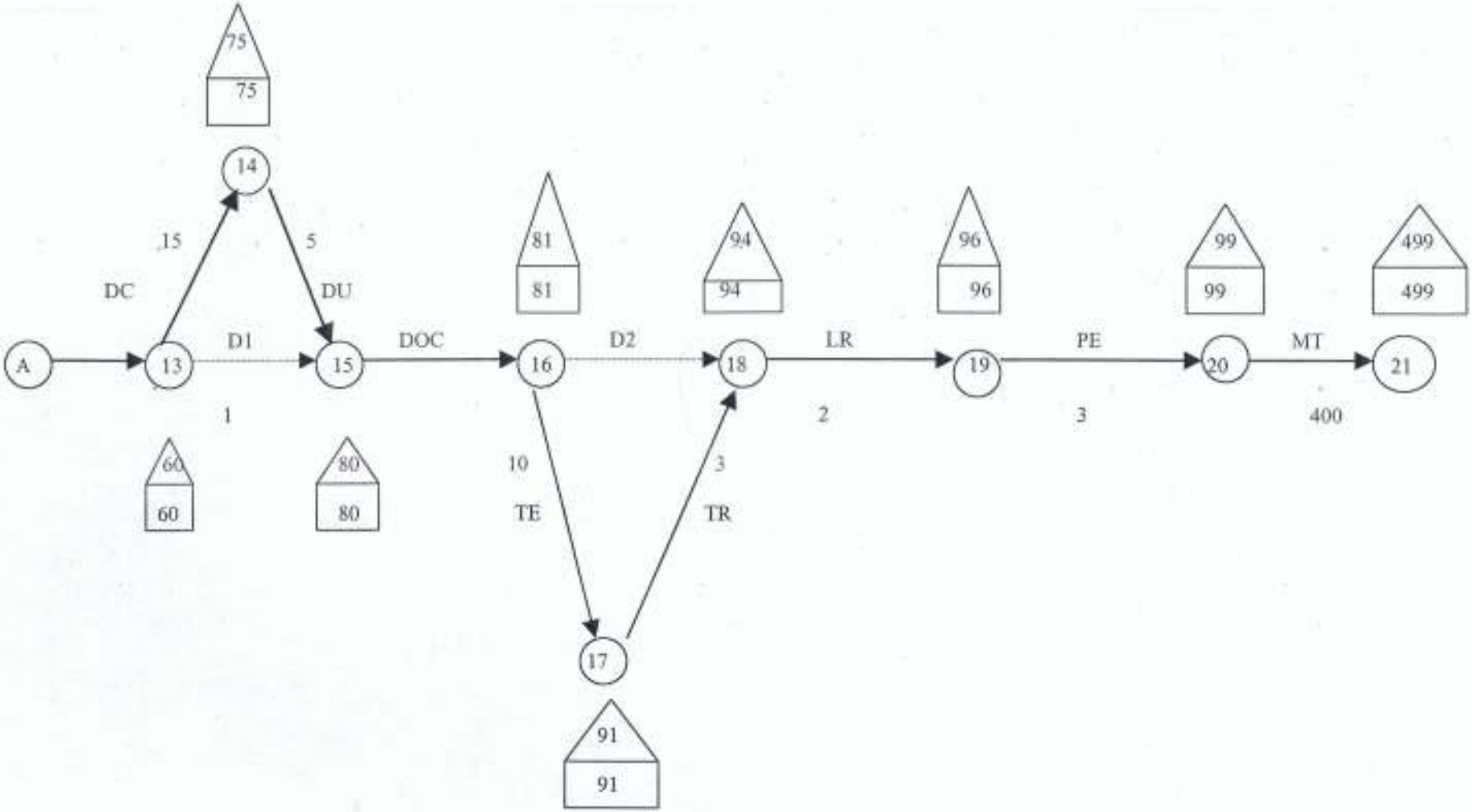
Wiest, J. and Levy, F. [1977]. A Management Guide to PERT/CPM, 2nd Edition, Prentice - Hall.

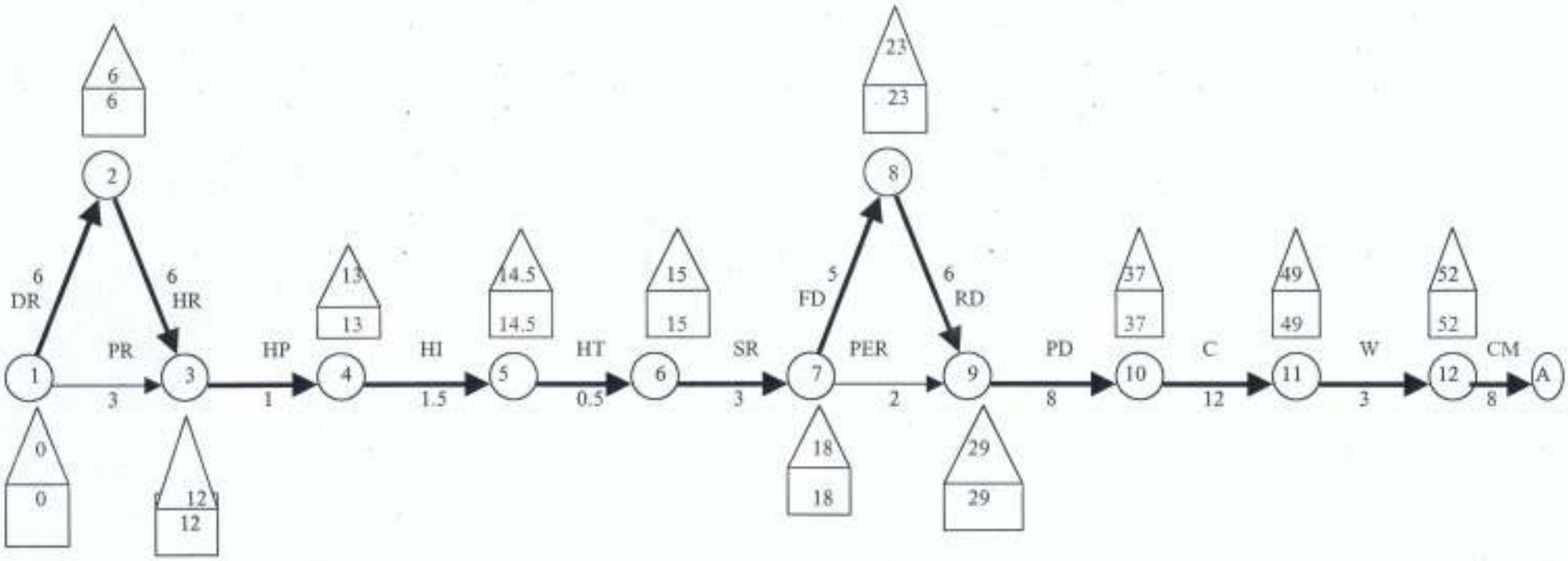
Williams, R.D. [1983]. Management of Software Development. Handbook of Software Engineering, Van Nostrand Reinhold, New York.



KEY

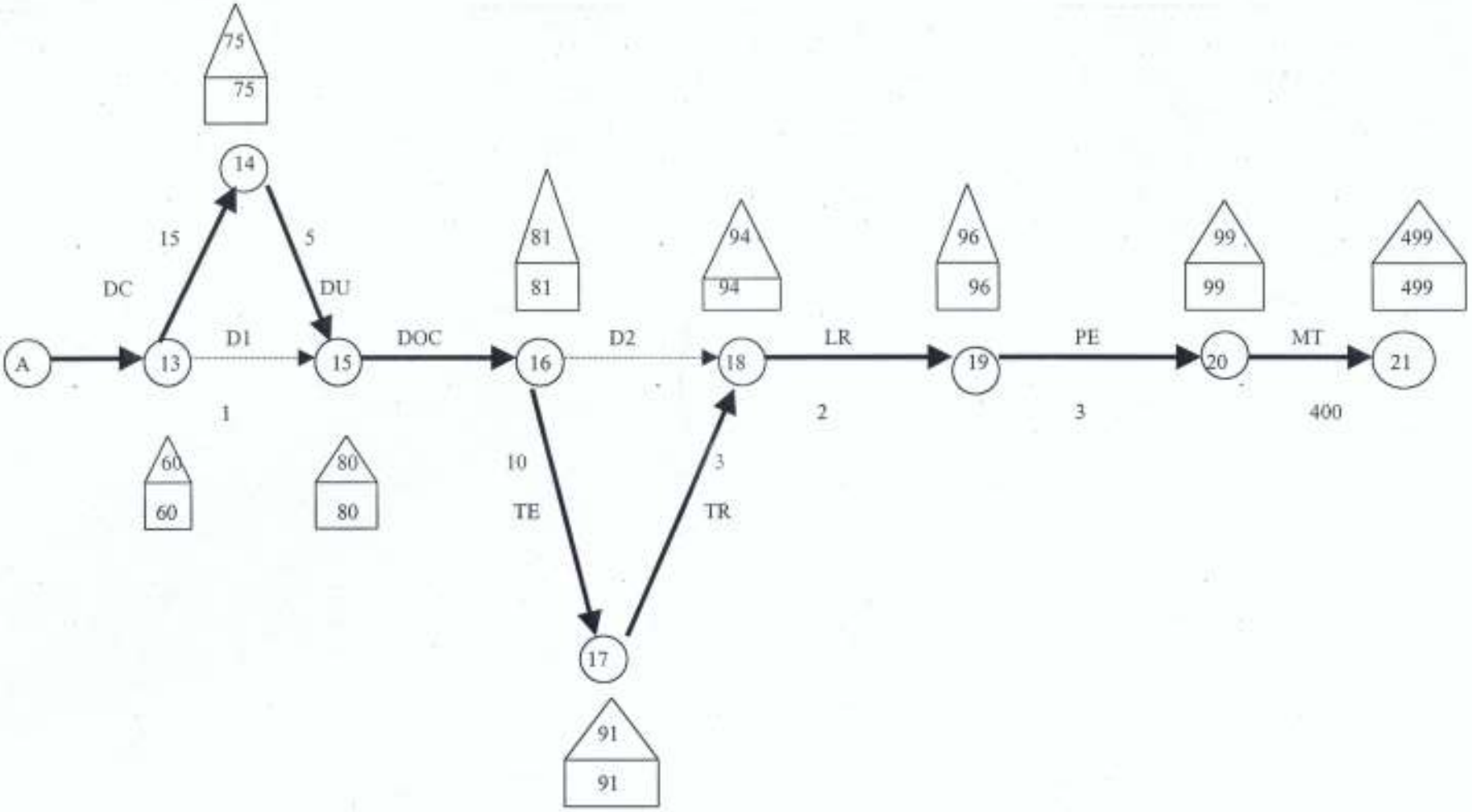
DR = Data Requirement
PR = Physical Requirement
HR = Hardware Reqiurement
HP = Hardware Procurement
HI = Hardware Installation
HT = Hardware Testing
SR = Software Requirement
FD = File Design
RD = Report Design
PE R = Personnel Requirement
PD = Program Design
C = Program Coding
W = Program Walkthrough
CM = Program Compilation
DC = Data Creation
DU = Data Update
DOC = Documentation
TE = Test Run
TR = Training
LR = Life Run
PE = Performance Evaluation
MT = Maintenance


75
75
14
15
DC
5
DU
81
81
94
94
96
96
99
99
499
499
A
13
D1
15
DOC
16
D2
18
LR
19
PE
20
MT
21
1
2
3
400
60
60
80
80
10
TE
3
TR
17
91
91



KEY

DR = Data Requirement
PR = Physical Requirement
HR = Hardware Reqiurement
HP = Hardware Procurement
HI = Hardware Installation
HT = Hardware Testing
SR = Software Requirement
FD = File Design
RD = Report Design
PE R = Personnel Requirement
PD = Program Design
C = Program Coding
W = Program Walkthrough
CM = Program Compilation
DC = Data Creation
DU = Data Update
DOC = Documentation
TE = Test Run
TR = Training
LR = Life Run
PE = Performance Evaluation
MT = Maintenance


75
75
14
15
DC
5
DU
81
81
94
94
96
96
99
99
499
499
A
13
D1
15
DOC
16
D2
18
LR
19
PE
20
MT
21
1
2
3
400
60
60
80
80
10
TE
3
TR
17
91
91

APPENDIX III

Schedule of Software Development, Operations and Maintenance

ACTIVITY (i,j)	DESCRIPTION	DURATION (Dij)	EARLIEST START □ ESi	EARLIEST COMPLETION ECij	LATEST START LSij	LATEST COM0PLETION △ LCj	TOTAL FLOAT TFij	FREE FLOAT FFij
(1,2)	Data Requirement	6	0	6	0	6	0	0
(1,3)	Physical Requirement	3	0	3	9	12	9	9
(2,3)	Hardware Requirement	6	6	12	8	6	0	0
(3,4)	Hardware Procurement	1	12	13	12	13	0	0
(4,5)	Hardware Installation	1.5	13	14.5	13	14.5	0	0
(5,6)	Hardware Testing	0.5	14.5	15	14.5	15	0	0
(6,7)	Software Requirement	3	15	18	15	18	0	0
(7,8)	File Design	5	18	23	18	23	0	0
(7,9)	Personnel Requirement	2	18	20	21	23	3	3
(8,9)	Report Design	6	23	29	23	29	0	0
(9,10)	Program Design	8	29	37	29	37	0	0
(10,11)	Coding	12	37	49	25	49	0	0
(11,12)	Walkthrough	3	49	52	49	52	0	0
(12,13)	Compilation	8	52	60	52	60	0	0
(13,14)	Data Creation	15	60	75	45	60	0	0
(13,15)	Dummy	0	60	60	80	80	20	20
(14,15)	Data Update	5	75	80	75	80	0	0
(15,16)	Documentation	1	80	81	80	81	0	0
(16,17)	Test Run	10	81	91	81	91	0	0
(16,18)	Dummy	0	81	81	94	94	4	4
(17,18)	Training	3	91	94	91	94	0	0
(18,19)	Life Run	2	94	96	94	96	0	0
(19,20)	Performance Evaluation	3	96	99	96	99	0	0
(20,21)	Maintenance	400	99	499	99	499	0	0

APPENDIX IV

Estimated Schedule of Activities in the Development, Operations and Maintenance of FUTA Payroll System.

ACTIVITY (i,j)	DESCRIPTION	DURATION (Dij)	EARLIEST START □ ESi	EARLIEST COMPLETION ECij	LATEST START LSij	LATEST COM0PLETION △ LCj	TOTAL FLOAT TFij	FREE FLOAT FFij
(1,2)	Data Requirement	3.12	0	3.12	0	3.12	0	0
(1,3)	Physical Requirement	1.56	0	1.56	2.6	4.16	4.16	4.16
(2,3)	Hardware Requirement	1.04	3.12	4.16	3.12	4.16	0	0
(3,4)	Hardware Procurement	0.52	4.16	4.68	4.16	4.68	0	0
(4,5)	Hardware Installation	0.78	4.68	5.46	4.68	5.46	0	0
(5,6)	Hardware Testing	0.26	5.46	4.72	5.46	5.72	0	0
(6,7)	Software Requirement	1.56	5.72	7.28	5.72	7.28	0	0
(7,8)	File Design	2.6	7.28	9.88	7.28	9.88	0	0
(7,9)	Personnel Requirement	1.04	7.28	8.32	11.96	13	4.68	4.68
(8,9)	Report Design	3.12	9.88	13	9.88	13	0	0
(9,10)	Program Design	4.16	13	17.16	13	17.6	0	0
(10,11)	Coding	6.24	17.6	23.4	17.6	23.84	0	0
(11,12)	Walkthrough	1.56	23.84	25.4	23.84	25.4	0	0
(12,13)	Compilation	4.16	25.4	29.56	25.4	29.56	0	0
(13,14)	Data Creation	7.8	29.56	37.36	29.56	37.36	0	0
(13,15)	Dummy	0	29.56	29.56	39.96	39.96	10.4	10.4
(14,15)	Data Update	2.6	37.36	39.96	37.36	39.96	0	0
(15,16)	Documentation	0.52	39.96	40.48	39.96	40.48	0	0
(16,17)	Test Run	5.2	40.48	45.68	40.48	45.68	0	0
(16,18)	Dummy	0	40.48	40.48	47.24	47.24	6.76	6.76
(17,18)	Training	1.56	45.68	47.24	45.68	47.24	0	0
(18,19)	Life Run	1.04	47.24	48.28	47.24	48.28	0	0
(19,20)	Performance Evaluation	1.56	48.28	49.84	48.28	49.84	0	0
(20,21)	Maintenance	208	49.84	257.84	49.84	257.84	0	0

APPENDIX V

Actual duration of activities observed during the development, operation and maintenance of FUTA Payroll System.

ACTIVITY (i,j)	DESCRIPTION	ABBREVIATION	ESTIMATED-DURATION	ACTUAL-DURATION
(1,2)	Data Requirement	DR	3.12	2
(1,3)	Physical Requirement	PR	1.56	NA
(2,3)	Hardware Requirement	HR	1.04	NA
(3,4)	Hardware Procurement	HP	0.52	NA
(4,5)	Hardware Installation	HI	0.78	NA
(5,6)	Hardware Testing	HT	0.26	NA
(6,7)	Software Requirement	SR	1.56	2
(7,8)	File Design	FD	2.6	4
(7,9)	Personnel Requirement	PR	1.04	0
(8,9)	Report Design	RD	3.12	5
(9,10)	Program Design	PD	4.16	5
(10,11)	Coding	C	6.24	6
(11,12)	Walkthrough	W	1.56	1
(12,13)	Compilation	CM	4.16	3
(13,14)	Data Creation	DC	7.8	5
(13,15)	Dummy	D1	0	0
(14,15)	Data Update	DU	2.6	3
(15,16)	Documentation	DOC	0.52	2
(16,17)	Test Run	TE	5.2	5
(16,18)	Dummy	D2	0	0
(17,18)	Training	TR	1.56	1
(18,19)	Life Run	LR	1.04	2
(19,20)	Performance Evaluation	PE	1.56	2
(20,21)	Maintenance	MT	208	130

NA = Not Applicable

APPENDIX VI

Actual Score of metrics of software quality of FUTA Payroll System.

QUALITY FACTOR	METRICS	SCORE-OBTAINABLE	ACTUAL-SCORE
1. Correctness	Completeness	5	4
	Consistency	5	4
	Traceability	5	2
2. Reliability	Accuracy	5	4
	Consistency	5	5
	Error tolerance	5	3
	Modularity	5	4
3. Efficiency	Conciseness	5	4
	Execution efficiency	5	4
	Operability	5	5
4. Integrity	Auditability	5	4
	Instrumentation	5	2
	Security	5	4
5. Maintainability	Conciseness	5	3
	Consistency	5	5
	Instrumentation	5	2
	Modularity	5	5
	Self-documentation	5	5
6. Flexibility	Conciseness	5	3
	Consistency	5	5
	Expandability	5	3
	Modularity	5	5
	Self-documentation	5	5
	Simplicity	5	4
7. Testability	Auditability	5	4
	Instrumentation	5	2
	Modularity	5	5
	Self-documentation	5	5
	Simplicity	5	4
8. Portability	Hardware independence	5	5
	Modularity	5	5
	Self-documentation	5	5
	Software system independence	5	4
9. Reusability	Generality	5	2
	Hardware independence	5	4
	Modularity	5	5
	Software system independence	5	4
10. Interoperability	Communication commonality	5	2
	Data commonality	5	4
	Modularity	5	5
11.Usability	Operability	5	5
	Training	5	4

APPENDIX VII

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
PROJECT SCHEDULE EVALUATION REPORT
DATE OF REPORT: 2-Dec-1999

S/NO	ACTIVITY-CODE	ACTIVITY-DESCRIPTION	ESTIMATED-DURATION	ACTUAL-DURATION	DIFFERENCE	%-DIFFERENCE	REMARK
1	DR	DATA REQUIREMENT	3.12	2.00	1.12	35.90	TIME UNDERRUN
2	SR	SOFTWARE REQUIREMENT	1.56	2.00	-.44	28.21	TIME OVERRUN
3	FD	FILE DESIGN	2.60	4.00	-1.40	53.85	TIME OVERRUN
4	PR	PHYSICAL REQUIREMENT	1.04	0.00	1.04	100.00	TIME UNDERRUN
5	RD	REPORT DESIGN	3.12	5.00	-1.88	60.26	TIME OVERRUN
6	PD	PROGRAM DESIGN	4.16	5.00	-.84	20.19	TIME OVERRUN
7	C	CODING	6.24	6.00	.24	3.85	TIME UNDERRUN
8	W	WALKTHROUGH	1.56	1.00	.56	35.90	TIME UNDERRUN
9	CM	COMPILATION	4.16	3.00	1.16	27.88	TIME UNDERRUN
10	DC	DATA CREATION	7.80	5.00	2.80	35.90	TIME UNDERRUN
11	DU	DATA UPDATE	2.60	3.00	-.40	15.38	TIME OVERRUN
12	DOC	DOCUMENTATION	.52	2.00	-1.48	284.62	TIME OVERRUN
13	TE	TEST RUN	5.20	5.00	.20	3.85	TIME UNDERRUN
14	TR	TRAINING	1.56	1.00	.56	35.90	TIME UNDERRUN
15	LR	LIFE RUN	1.04	2.00	-.96	92.31	TIME OVERRUN
16	PE	PERFORMANCE EVALUATION	1.56	2.00	-.44	28.21	TIME OVERRUN
17	MT	MAINTENANCE	208.00	130.00	78.00	37.50	TIME UNDERRUN

KEY
===
TIME-DIFFERENCE--------- INTERPRETATION
EQUAL TO 0 --------- BREAKEVEN
GREATER THAN 0 --------- TIME OVERRUN
LESS THAN 0 --------- TIME UNDERRUN

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
PROJECT SCHEDULE EVALUATION SUMMARY REPORT

TOTAL-ESTIMATED-DURATION	TOTAL-ACTUAL-DURATION	DIFFERENCE	%-DIFFERENCE	REMARK
260.00	178.00	82.00	31.54	TIME UNDERRUN

KEY
===
TOTAL TIME DIFFERENCE--------- INTERPRETATION

EQUAL TO 0 --------- BREAKEVEN
GREATER THAN 0 --------- TIME OVERRUN
LESS THAN 0 --------- TIME UNDERRUN

APPENDIX VIII

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT
QUALITY FACTOR BEING EVALUATED: CORRECTNESS
DATE OF REPORT: 2-Dec-1999

S/NO	QUALITY-CODE	METRIC	SCORE-OBTAINABLE	ACTUAL-SCORE-OBTAINED	REMARK
1	COR	COMPLETENESS	5	4	VERY GOOD
2	COR	CONSISTENCY	5	4	VERY GOOD
3	COR	TRACEABILITY	5	2	SATISFACTORY

KEY
===

ACTUAL SCORE OBTAINED	INTERPRETATION
1	POOR
2	SATISFACTORY
3	GOOD
4	VERY GOOD
5	EXCELLENT

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT
QUALITY FACTOR BEING EVALUATED: RELIABILITY
DATE OF REPORT: 2-Dec-1999

S/NO	QUALITY-CODE	METRIC	SCORE-OBTAINABLE	ACTUAL-SCORE-OBTAINED	REMARK
1	REL	ACCURACY	5	4	VERY GOOD
2	REL	CONSISTENCY	5	5	EXCELLENT
3	REL	ERROR TOLERANCE	5	3	GOOD
4	REL	MODULARITY	5	4	VERY GOOD

KEY
===

ACTUAL SCORE OBTAINED	INTERPRETATION
1	POOR
2	SATISFACTORY
3	GOOD
4	VERY GOOD
5	EXCELLENT

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT
QUALITY FACTOR BEING EVALUATED: EFFICIENCY
DATE OF REPORT: 2-Dec-1999

S/NO	QUALITY-CODE	METRIC	SCORE-OBTAINABLE	ACTUAL-SCORE-OBTAINED	REMARK
1	EFF	CONCISENESS	5	4	VERY GOOD
2	EFF	EXECUTION EFFIENCY	5	4	VERY GOOD
3	EFF	OPERABILITY	5	5	EXCELLENT

KEY
===

ACTUAL SCORE OBTAINED	INTERPRETATION
1	POOR
2	SATISFACTORY
3	GOOD
4	VERY GOOD
5	EXCELLENT

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT
QUALITY FACTOR BEING EVALUATED: INTEGRITY
DATE OF REPORT: 2-Dec-1999

S/NO	QUALITY-CODE	METRIC	SCORE-OBTAINABLE	ACTUAL-SCORE-OBTAINED	REMARK
1	INT	AUDITABILITY	5	4	VERY GOOD
2	INT	INSTRUMENTATION	5	2	SATISFACTORY
3	INT	SECURITY	5	4	VERY GOOD

KEY
===

ACTUAL SCORE OBTAINED	INTERPRETATION
1	POOR
2	SATISFACTORY
3	GOOD
4	VERY GOOD
5	EXCELLENT

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT
QUALITY FACTOR BEING EVALUATED: MAINTANAIBLITY
DATE OF REPORT: 2-Dec-1999

S/NO	QUALITY-CODE	METRIC	SCORE-OBTAINABLE	ACTUAL-SCORE-OBTAINED	REMARK
1	MAI	CONCISENESS	5	3	GOOD
2	MAI	CONSISTENCY	5	5	EXCELLENT
3	MAI	INSTRUMENTATION	5	2	SATISFACTORY
4	MAI	MODULARITY	5	5	EXCELLENT
5	MAI	SELF-DOCUMENTATION	5	5	EXCELLENT
6	MAI	SIMPLICITY	5	4	VERY GOOD

KEY
***
ACTUAL SCORE OBTAINED----------INTERPRETATION
1----------POOR
2----------SATISFACTORY
3----------GOOD
4----------VERY GOOD
5----------EXCELLENT

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT
QUALITY FACTOR BEING EVALUATED: FLEXIBILITY
DATE OF REPORT: 2-Dec-1999

S/NO	QUALITY-CODE	METRIC	SCORE-OBTAINABLE	ACTUAL-SCORE-OBTAINED	REMARK
1	FLE	CONCISENESS	5	3	GOOD
2	FLE	CONSISTENCY	5	5	EXCELLENT
3	FLE	EXPANDABILITY	5	3	GOOD
4	FLE	MODULARITY	5	5	EXCELLENT
5	FLE	SELF-DOCUMENTATION	5	5	EXCELLENT
6	FLE	SIMPLICITY	5	4	VERY GOOD

KEY
===
ACTUAL SCORE OBTAINED----------INTERPRETATION

1----------POOR
2----------SATISFACTORY
3----------GOOD
4----------VERY GOOD
5----------EXCELLENT

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT
QUALITY FACTOR BEING EVALUATED: TESTABILITY
DATE OF REPORT: 2-Dec-1999

S/NO	QUALITY-CODE	METRIC	SCORE-OBTAINABLE	ACTUAL-SCORE-OBTAINED	REMARK
1	TES	AUDITABILITY	5	4	VERY GOOD
2	TES	INSTRUMENTATION	5	2	SATISFACTORY
3	TES	MODULARITY	5	5	EXCELLENT
4	TES	SELF-DOCUMENTATION	5	5	EXCELLENT
5	TES	SIMPLICITY	5	4	VERY GOOD

KEY
===
ACTUAL SCORE OBTAINED----------INTERPRETATION
1----------POOR
2----------SATISFACTORY
3----------GOOD
4----------VERY GOOD
5----------EXCELLENT

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT
QUALITY FACTOR BEING EVALUATED: PORTABILITY
DATE OF REPORT: 2-Dec-1999

S/NO	QUALITY-CODE	METRIC	SCORE-OBTAINABLE	ACTUAL-SCORE-OBTAINED	REMARK
1	POR	HARDWARE INDEPENDENCE	5	5	EXCELLENT
2	POR	MODULARITY	5	5	EXCELLENT
3	POR	SELF-DOCUMENTATION	5	5	EXCELLENT
4	POR	SOFTWARE SYSTEM INDEPENDENCE	5	4	VERY GOOD

KEY
===
ACTUAL SCORE OBTAINED----------INTERPRETATION

1----------POOR
2----------SATISFACTORY
3----------GOOD
4----------VERY GOOD
5----------EXCELLENT

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT
QUALITY FACTOR BEING EVALUATED: REUSABILITY
DATE OF REPORT: 2-Dec-1999

S/NO	QUALITY-CODE	METRIC	SCORE-OBTAINABLE	ACTUAL-SCORE-OBTAINED	REMARK
1	REU	HARDWARE INDEPENDENCE	5	4	VERY GOOD
2	REU	MODULARITY	5	5	EXCELLENT
3	REU	SOFTWARE SYSTEM INDEPENDENCE	5	4	VERY GOOD

KEY
===
ACTUAL SCORE OBTAINED----------INTERPRETATION
1----------POOR
2----------SATISFACTORY
3----------GOOD
4----------VERY GOOD
5----------EXCELLENT

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT
QUALITY FACTOR BEING EVALUATED: INTEROPERABILITY
DATE OF REPORT: 2-Dec-1999

S/NO	QUALITY-CODE	METRIC	SCORE-OBTAINABLE	ACTUAL-SCORE-OBTAINED	REMARK
1	ITE	COMMUNICATION COMMONNALITY	5	2	SATISFACTORY
2	ITE	DATA COMMONALITY	5	4	VERY GOOD
3	ITE	MODULARITY	5	5	EXCELLENT

KEY
===
ACTUAL SCORE OBTAINED----------INTERPRETATION
1----------POOR
2----------SATISFACTORY
3----------GOOD
4----------VERY GOOD
5----------EXCELLENT

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT
QUALITY FACTOR BEING EVALUATED: USABILITY
DATE OF REPORT: 2-Dec-1999

S/NO	QUALITY-CODE	METRIC	SCORE-OBTAINABLE	ACTUAL-SCORE-OBTAINED	REMARK
1	USA	OPERABILITY	5	5	EXCELLENT
2	USA	TRAINING	5	4	VERY GOOD

KEY
===
ACTUAL SCORE OBTAINED----------INTERPRETATION

1----------POOR
2----------SATISFACTORY
3----------GOOD
4----------VERY GOOD
5----------EXCELLENT

APPENDIX IX

COMPUTER AIDED SYSTEM FOR MONITORING AND EVALUATION OF SOFTWARE DEVELOPMENT, OPERATIONS AND MAINTENANCE
SOFTWARE QUALITY PERFORMANCE EVALUATION REPORT SUMMARY

QUALITY-CODE	TOTAL-SCORE-OBTAINABLE	TOTAL-SCORE-OBTAINED	SCORE-DIFFERENCE	%-ACHIEVEMENT	REMARK
COR	15.00	10.00	5.00	66.67	GOOD
REL	20.00	16.00	4.00	80.00	EXECELLENT
EFF	15.00	13.00	2.00	86.67	EXECELLENT
INT	15.00	10.00	5.00	66.67	GOOD
MAI	30.00	24.00	6.00	80.00	EXECELLENT
FLE	30.00	25.00	5.00	83.33	EXECELLENT
TES	25.00	20.00	5.00	80.00	EXECELLENT
POR	20.00	19.00	1.00	95.00	EXECELLENT
REU	15.00	13.00	2.00	86.67	EXECELLENT
ITE	15.00	11.00	4.00	73.33	VERY GOOD
USA	10.00	9.00	4.00	90.00	EXECELLENT

KEY
===
%-ACHIEVEMENT----------REMARK

>= 80 ---------- EXCELLENT
>= 70 < 80 ---------- VERY GOOD
>= 60 < 70 ---------- GOOD
>= 50 < 60 ---------- SATISFACTORY
< 50 ---------- POOR