ISSN: 2277-3754
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 1, Issue 2, February 2012

Comparative analysis of Software testing measurement technique

Reema ajmera, Ripu R. Sinha, Dr C.S Lamba
ajmera.reema17@gmail.com, Itct.ripu@gmail.com, Profflamba@gmail.com,

Abstract- Complexity measurement means an investment in success by facilitating early detection of problems, and by providing quantitative clarification of critical development phases. Metrics give us the ability to identify, resolve, and/or curtail risk issues before they arrive. Measurement must not be a goal in itself. In this paper we are presenting various measurement techniques for the purpose to find out the requirement of agent best testing for effective testing measurement technique.

Index Terms—**MASTT (Multi Agent System testing technique, Complexity measurement, Measuring Techniques, ABTT (agent based testing technique)**

I. INTRODUCTION

Software /product are outcome of a systematic life cycle which has a set of phases. After evaluating every phase deliver some deliverables that becomes input for next phase. Every phase has some designated goals. Each phase tried to identify, resolve, and/or to curtail risk issues at their level to make next level's task less complex. To achieve the above given every phase of life cycle implement some complexity measurement technique. To deliver accurate and without an error prone software complexity measure must be integrated and run along with the total software life cycle — not independent of it. Complexity measure helps in to achieve more predictability, lower the risk by introducing risk and reduce the software maintenance cost. These are required for accurate schedule and cost estimates, better quality products, and higher productivity. These help the developer in identifying source code that is error prone and difficult to test. Metrics also provide an insight into the software, as well as the processes used to develop and maintain it. Complexity measure plays a vital role in analyzing and focusing on particular phases of life cycle in respect of resource and effort. Testing complexity analysis can be used to allocate testing resources to those portions of the application that are most likely to contain errors. Using this technique, all subsystems are analyzed to determine a numerical level of complexity. Then testing resources are allocated to each subsystem accordingly. Alternatively, levels of required test coverage could be determined based on the complexity analysis. For example, any subsystem above certain, predefined complexity threshold would require more complete test coverage before it was released than a subsystem of lesser complexity. Functionality-correctness, reliability, etc. and economy-cost effectiveness are major issues to measure. The Figure1 given below is software measurement which depicts that how measurement is associated with whole life cycle.

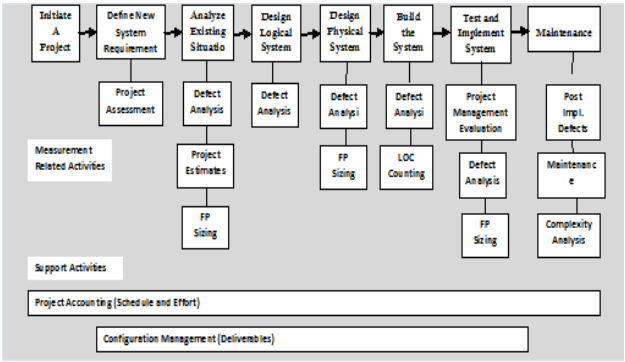

Fig. 1. Software Measurement Life Cycle [41]

II. MEASUREMENT TECHNIQUES

Software Metrics are standards to determine the size of an attribute of a software product, a way to evaluate it and these also be applied to the software process. We can classify software metrics in to two broad categories:

A. Software product metrics:

These metrics measure the software product at any stage of its development, from requirements to installed system. They are often classified according with the design, size, complexity, and quality and data dependency. Size Metrics: A number of metrics attempt to quantify software "size." The metric that is most widely used, LOC, suffers from the obvious deficiency that its value cannot be measured until after the coding process has been completed. Function points and system Bang have the advantage of being measurable earlier in the development process—at least as early as the design phase, and possibly earlier. Some of Halstead's metrics are also used to measure software size, but these are discussed later. Lines of code (LOC) a line of code (LOC) is possibly the most widely used metric for program size. It would seem to be easily and precisely definable; however, there are a number of different definitions for the number of lines of code in a particular program. These differences involve treatment of blank lines and comment lines, non-executable statements, multiple statements per line, and multiple lines per statement, as well as the question of how to count reused lines of code. The most common definition of LOC seems to count any line that is not a blank or comment line, regard less of the number of statements per line [1, 2].

LOC has been theorized to be useful as a predictor of program complexity, total development effort, and

ISSN: 2277-3754
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 1, Issue 2, February 2012

programmer performance (debugging, productivity). Numerous studies have attempted to validate these relationships [1, 2].

Continuing our example . . .

Complex internal processing	= 3
Code to be reusable	= 2
High performance	= 4
Multiple sites	= 3
Distributed processing	= 5
Project adjustment factor	= 17

Adjustment calculation:

Adjusted FP = Unadjusted FP X [0.65 + (adjustment factor X 0.01)]

= 240 X [0.65 + (17 X 0.01)]

= 240 X [0.82]

= 197 Adjusted function points

Function Points: Albrecht has proposed a measure of software size that can be determined early in the development process. The approach is to compute the total function points (FP) value for the project, based upon the number of external user inputs, inquiries, outputs, interfaces and master files. The value of FP is the total of these individual values, with the following weights applied: inputs: 4, outputs: 5, inquiries: 4 and master files: 10. These are given in Table 1. Each FP contributor can also be adjusted within a range of ± 35% for specific project complexity [35]. These function-point counts are then weighed (multiplied) by their degree of complexity. In addition to these individually weighted function points, there are factors that affect the project and/or system as a whole. There are a number (± 35) of these factors that affect the size of the project effort, and each is ranked from "0"- no influence to "5"- essential.

The following are some examples of these factors:

- Is high performance critical?
- Is the internal processing complex?
- Is the system to be used in multiple sites and/or by multiple organizations?
- Is the code designed to be reusable?
- Is the processing to be distributed?

and so forth . . .

Table 1. Function Point Degree of Complexity

A simple example:				
inputs				
3 simple	X	2	=	6
4 average	X	4	=	16
1 complex	X	6	=	6
outputs				
6 average	X	5	=	30
2 complex	X	7	=	14
files				
5 complex	X	15	=	75
inquiries				
8 average	X	4	=	32
interfaces				
3 average	X	7	=	21
4 complex	X	10	=	40
Unadjusted function points				240

But how long will the project take and how much will it cost? As previously measured, programmers in our organization average 18 function points per. Thus . . . 197 FP divided by 18 = 11 man-months If the average programmer is paid □ 30000 per month (including benefits), then the [labor] cost of the project will be . . .

11 man-months X □ 30000 = □ 330000

Table 2. Category of Simple, Average and Complex

	Simple	Average	Complex
Inputs	2	4	6
Outputs	3	5	7
Files	5	10	15
Inquires	2	4	6
Interfaces	4	7	10

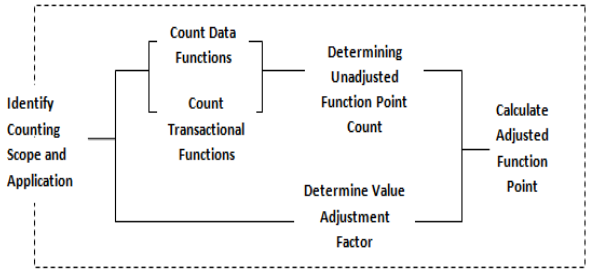

Fig. 2.Function Point Count

Complexity Metrics: Numerous metrics have been proposed for measuring program complexity—probably more than for any other program characteristic. A recent study by Li and Cheung compares 31 different complexity metrics, including most of those discussed below [3]. Another recent study by Rodriguez and Tsai compares LOC, v (G), Kafura's information flow metric, and Halstead's volume, V, as measures of program size, complexity, and quality [4]. Attempts to devise new measures of software complexity continue, as evidenced by recent articles [5,6]. As noted for size metrics, measures of complexity that can be computed early in the software development cycle will be of greater value in managing the software process. Theoretically, McCabe's measure [7] is based on the final form of the computer code. However, if the detailed design is specified in a program design language (PDL), it should be possible to compute v(G) from that detailed design. This is also true for the information flow metric of Kafura and Henry [8]. It should be noted here that Halstead's metrics [9] are often studied as possible measures of software complexity.

Cyclomatric Complexity: v(G) The Cyclomatic Complexity Number or short CCN is the oldest complexity metrics. The first time this software metric was mentioned was 1976 by Thomas J. McCabe. McCabe's (1976) cyclomatric complexity, V (G), is a measure in widespread use for assessing the control complexity (an internal characteristic) in a program. It aims to provide a basis set for constructing a testing programme (at the unit level) as well as identifying heuristically subprograms which might be regarded as "overly complex" (This metric count Cyclomatic

ISSN: 2277-3754
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 1, Issue 2, February 2012

complexity measures the number of linearly independent control paths through a software program. We can draw control flow graph of any given computer program, G, wherein each node corresponds to a block of sequential code and each arc corresponds to a branch or decision point in the program. The cyclomatic complexity of such a graph can be computed by a simple formula from graph theory, as v(G) = e -n + 2, where e is the number of edges, and n is the number of nodes in the graph. McCabe proposed that v(G) can be used as a measure of program complexity hence, as a guide to program development and testing. There are actually three ways to calculate the Cyclomatic complexity of a control flow graph.

Method 1: Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as: E(G) = E – N + 2 where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

Method 2: An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows: V(G) = Total number of bounded areas(R) + 1 In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area. This is an easy way to determine the McCabe's cyclomatic complexity. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability.

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of Predicate Nodes in the program. If N is the number of Predicate Nodes of a program, then the McCabe's metric is equal to N+1. McCabe's cyclomatic complexity metric has been related to programming effort, debugging performance, and maintenance effort. It is to Identifying the most complex module, basis path testing, pinpoint areas of potential instability, indicate a unit/component's testability & understand ability (maintainability), provide a quantitative indication of unit/component's control flow complexity and specify the effort required to test a unit/component

Example: Consider the C Program, which implements GCD.

```
int compute_GCD (int a, int b)
1. while (a!=b){
2.          if (a>b) then
3.                    a=a-b;
4.          else b=b-a;
5. }
6. Return a;}
```

CCN Using Method 1:

v(G)= E-V+2

7-6+2 =3

CCN Using Method 2:

v(G)= R+1

2+1=3

CCN Using Method 3:

v(G)= P +1

2+1=3

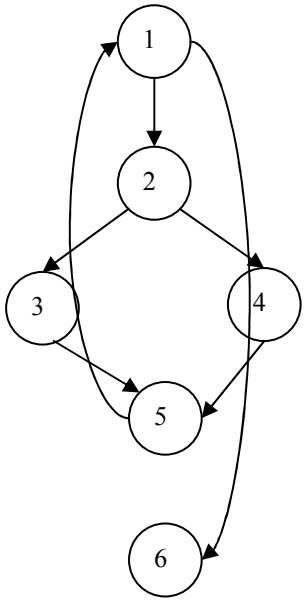

Fig. 3. Flow Graph

Extensions to v(G) Myers noted that McCabe's cyclomatic complexity measure, v(G), provides a measure of program complexity but fails to differentiate the complexity of some rather simple cases involving single conditions (as opposed to multiple conditions) in conditional statements. As an improvement to the original formula, Myers suggests extending v(G) to v ′(G) = [l:u], where l and u are lower and upper bounds, respectively, for the complexity. This formula gives more satisfactory results for the cases noted by Myers [10]. Stetter proposed that the program flow graph be expanded to include data declarations and data references, thus allowing the graph to depict the program complexity more completely. If H is the new program flow graph, it will generally contain multiple entry and exit nodes. A function f (H) can be computed as a measure of the flow complexity of program H. The deficiencies noted by Myers are also eliminated by f (H) [11].

Knots The concept of program knots is related to measuring the program control flow graph with a node for every statement or block of sequential statements. A knot is then defined as a necessary crossing of directional lines in the graph. The same phenomenon can also be observed by simply drawing transfer-of-control lines from statement to statement in a program listing. The number of knots in a program has been proposed as a measure of program complexity [12].

Information Flow The information flow within a program structure may also be used as a metric for program complexity. Kafura and Henry have proposed such a measure. Basically, their method counts the number of local information flows entering (fan-in) and exiting (fan-out) each procedure. The procedure's complexity is then defined as: c = [procedure length] * [fan-in * fan-out] 2 [8]. This information flow metric is compared with Halstead's E metric and

ISSN: 2277-3754
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 1, Issue 2, February 2012

McCabe's cyclomatic complexity in [13]. Complexity metrics such as v(G) and Kafura's information flow metric have been shown by Rombach to be useful measures of program maintainability [14].

Halstead's Product Metrics Most of the product metrics proposed have applied to only one particular aspect of the software product. In contrast, Halstead's software science proposed a unified set of metrics that apply to several aspects of programs, as well as to the overall software production effort. Thus, it is the first set of software metrics unified by a common theoretical basis. In this section, we discuss the program vocabulary (n), length (N), and volume (V) metrics. These metrics apply specifically to the final software product. Halstead also specified formulas for computing the total effort (E) and development time (T) for the software product.

a. Program Vocabulary

Halstead theorized that computer programs can be visualized as a sequence of tokens, each token being classified as either an operator or operand. He then defined the vocabulary, n, of the program as: n = n1 + n2, where n1 = the number of unique operators in the program and n2 = the number of unique operands in the program. Thus, n is the total number of unique tokens from which the program has been constructed [9].

b. Program Length

Having identified the basic tokens used to construct the program, Halstead then defined the program length, N, as the count of the total number of operators and operands in the program. Specifically = N1 +N2, where N1 = the total number of operators in the program and N2 = the total number of operands in the program. Thus, N is clearly a measure of the program's size, and one that is derivable directly from the program itself. In practice, however, the distinction between operators and operands may be non-trivial, thus complicating the counting process [9].Halstead theorized that an estimated value for N, designated N′, can be calculated from the values of n1 and n2 by using the following formula: N′ = n1 log2 n1 + n2 log2 n2. Thus, N is a primitive metric, directly observable from the finished program, while N′ is a computed metric, which can be calculated from the actual or estimated values of n1 and n2 before the final code is actually produced. A number of studies lend empirical support to the validity of the equation for the computed program length, N′. Examples are reported by Elshoff and can also be found in Halstead's book. Other studies have attempted to relate N and N′ to other length ware properties, such as complexity [15] and defect rates [16, 9, 17, 3, 18].

c. Program Volume

Another measure of program size is the program volume, V, which was defined by Halstead as:V = N log2 n Since N is a pure number, the units of V can be interpreted as bits, so that V is a measure of the storage volume required to represent the program. Empirical studies by Halstead and others have shown that the values of LOC, N, and V appear to be linearly related and equally valid as relative measures of program size [19, 16, 3].

Example:

```
#include <stdio.h>
void main()
{
   int    a,b,c;
    a=4;
   b=5;
   c=a+b;
   printf("Sum is  : %d "+ c);
}
```

n1= 2;
n2=3;
N1=4;
N2=9;
Difficulty of the program is
(2/2) * (9/3) = 3 (1.1)
Furthermore,
Program length is defined as L = 4 + 9,
vocabulary as V = 2/3,
volume as 13 * log2(.66)

Quality Metrics: One can generate long lists of quality characteristics for software—correctness, efficiency, portability, maintainability, reliability, etc. Early examples of work on quality metrics are discussed by Boehm, McCall, and others [1, 20]. Unfortunately, the characteristics often overlap and conflict with one another; for example, increased portability (desirable) may result in lowered efficiency (undesirable). Thus, useful definitions of general quality metrics are difficult to devise, and most computer scientists have abandoned efforts to find any single metric for overall software quality. Although a good deal of work has been done in this area, it exhibits less commonality of direction or definition than other areas of metric research, such as software size or complexity. Three areas that have received considerable attention are: program correctness, as measured by defect counts; software size, ability, as computed from defect data; and software maintainability, as measured by various other metrics, including complexity metrics. Examples from these areas are discussed briefly below. Software quality is a characteristic that, theoretically at least, can be measured at every phase of the software development cycle. Cerino discusses the measurement of quality at some of these phases in [21].

Defect Metrics: The number of defects in the software product should be readily derivable from the product it- self; thus, it qualifies as a product metric. However, since there is no effective procedure for counting the defects in the program, the following alternative measures have been proposed:

- Number of design changes
- Number of errors detected by code inspections
- Number of errors detected in program tests
- Number of code changes required

ISSN: 2277-3754
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 1, Issue 2, February 2012

These alternative measures are dependent upon both the program and the outcome or result of some phase of the development cycle. The number of defects observed in a software product provides, in itself, a metric of software quality. Studies have attempted to establish relationships between this and other metrics that might be available earlier in the development cycle and that might, therefore, be useful as predictors of program quality [22, 15, 18, and 4].

Reliability Metrics: It would be useful to know the probability of software failure, or the rate at which software errors will occur. Again, although this information is inherent in the software product, it can only be estimated from data collected on software defects as a function of time. If certain assumptions are made, these data can then be used to model and computer software reliability metrics. These metrics attempt to measure and predict the probability of failure during a particular time interval, or the mean time to failure (MTTF). Since these metrics are usually discussed in the context of developing a reliability model of the software product's behavior, more detailed discussion of this model is deferred to the section on process models. Significant references in this area are [24] and [25].

Maintainability Metrics: A number of efforts have been made to define metrics that can be used to measure or predict the maintainability of the software product [26]. For example, an early study by Curtis, et al., investigated the ability of Halstead's effort metric, E, and v (G) to predict the psychological complexity of software maintenance tasks [22]. Assuming such predictions could be made accurately, complexity metrics could then be profitably used to reduce the cost of software maintenance [38]. More recently, Rombach has published the results of a carefully designed experiment that indicates that software complexity metrics can be used effectively to explain or predict the maintainability of software in a distributed computer system [14]. A similar study, based on three different versions of a medium-sized software system that evolved over a period of three years, relates seven different complexity metrics to the recorded experience with maintenance activities [8]. The complexity metrics studied included both measures of the internal complexity of software modules and measures of the complexity of interrelationships between software modules. The study indicates that such metrics can be quite useful in measuring maintainability and in directing design or redesign activities to improve software maintainability.

B. Software process metrics

These metrics measure the process in regards to the time that the project will take, cost, methodology followed and how the experience of the team members can affect these values. They can be classified as empirical, statistical, theory base and composite models. Software metrics may be defined without specific reference to a well-defined model, as, for example, the metric LOC for program size. However, more often metrics are defined or used in conjunction with a particular model of the software development process. In this curriculum module, the intent is to focus on those metrics that can best be used in models to predict, plan, and control software development, thereby improving our ability to manage the process. Models of various types are simply abstractions of the product or process we are interested in describing. Effective models allow us to ignore uninteresting details and concentrate on essential aspects of the artefact described by the model. Preference should be given to the simplest model that provides adequate descriptive capability and some measure of intuitive acceptability. A good model should possess predictive capabilities, rather than being merely descriptive or explanatory. In general, models may be analytic-constructive or empirical-descriptive in nature. There have been few analytic models of the software process, the most notable exception being Halstead's software science, which has received mixed reactions. Most proposed software models have resulted from a combination of intuition about the basic form of relationships and the use of empirical data to determine the specific quantities involved (the coefficients of independent variables in hypothesized equations, for example). Ultimately, the validity of software metrics and models must be established by demonstrated agreement with empirical or experimental data. This requires careful attention to taking measurements and analyzing data. In general, the work of analyzing and validating software metrics and models requires both sound statistical methods and sound experimental designs. Precise definitions of the metrics involved and the procedures for collecting the data are essential for meaningful results. Small-scale experiments should be designed carefully, using well established principles of experimental design. Unfortunately, validation of process models involving larger projects must utilize whatever data can be collected. Carefully controlled large experiments are virtually impossible to conduct. Guidance in the area of data collection for software engineering experiments is provided by Basili and Weiss [28].A knowledge of basic statistical theory is essential for conducting meaningful experiments and analyzing the resulting data. In attempting to validate the relationships of a given model, one must use appropriate statistical procedures and be careful to interpret the results objectively. Most studies of software metrics have used some form of statistical correlation, often without proper regard for the theoretical basis or limitations of the methods used. In practice, software engineers lacking significant background in statistical methods should consider enlisting the aid of a statistical consultant if serious metric evaluation work is undertaken. Representative examples of software models are presented below. For papers that compare the various models, see [29] and [30].

Empirical Models: One of the earliest models used to project the cost of large-scale software projects was described by Wolverton of TRW in 1974. The method relates a proposed project to similar projects for which historical cost data are available. It is assumed that the cost of the new project can be projected using this historical data. The method assumes that a waterfall-style life cycle model is used. A 25 × 7 structural forecast matrix is used to allocate resources to various phases

ISSN: 2277-3754
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 1, Issue 2, February 2012

of the life cycle. In order to determine actual software costs, each software module is first classified as belonging to one of six basic types—control, I/O, etc. Then, a level of difficulty is assigned by categorizing the module as new or old and as easy, medium, or hard. This gives a total of six levels of module difficulty. Finally, the size of the module is estimated, and the system cost is determined from historical cost data for software with similar size, type, and difficulty ratings [31].

Statistical Models: C. E. Walston and C. P. Felix of IBM used data from 60 previous software projects completed by the Federal Systems Division to develop a simple mode of software development effort. The metric LOC was assumed to be the principal determiner of development effort. A relationship of the form E = aLb was assumed, where L is the number of lines of code, in thousands, and E is the total effort required, in person-months. Regression analysis was used to find appropriate values of parameters a and b. The resulting equation was E = 5.2 L 0.91 Nominal programming productivity, in LOC per person-month, can then be calculated as L/E. In order to account for deviations from the derived form for E, Walston and Felix also tried to develop a productivity index, I, which would increase or decrease the productivity, depending upon the nature of the project. The computation of I was to be based upon evaluations of 29 project variables (culled from an original list of 68 possible determiners of I) [32].

Theory-Based Models: Few of the proposed models have substantial the work retical bases. Two examples that do are presented below.

- **Rayleigh Model:**L. H. Putnam developed a model of the software development process based upon the assumption that the personnel utilization during program development is described by a Rayleigh-type curve such as the following:

$$y = \frac{Kte^{-t^2/2T^2}}{T^2}$$ **Formulae 1.**

Where y = the number of persons on the project at any time, t; K = the area under the Rayleigh curve, equal to the total life cycle effort in person-years; and T = development time (time of peak staffing requirement). Putnam assumed that either the overall staffing curve or the staffing curves for individual phases of the development cycle can be modeled by an equation of this form. He then developed the categorizing lowing relationship between the size of the software product and the development time [40]:

S = CK1/ 3 T 4/ 3, **Formulae 2.**

Where S = the number of source LOC delivered;

K = the life-cycle effort in person-years; and

C = a state-of-technology constant.

- **Software Science Model—Halstead:**The software science equations can be used as a simple theoretical model of the software development process. The effort required to develop the software is given by the equation E=V/L, which can be approximated by: E =(n1n2 (n1 log2 n1 + n2 log2 n2) log2 n)/2n2 **Formulae 3.**The units of E are elementary mental discriminations. The corresponding programming time (in seconds) is simply derived from E by dividing by the Stroud number, S: T= E/ S **Formulae 4.**The value of S is usually taken as 18 for these calculations. If only the value of length, N, is known, then the following approximation can be used for computing T:T = N2 log2 n/ 4S , **Formulae 5.**Where n can be obtained from the relationship N= n log2 (n / 2) [9, 37].

- Composite Models:As experience has been gained with previous models, a number of more recent models have utilized some combination of intuition, statistical analyses, and expert judgment. These have been labelled composite models" by Conte, et al. Several models are listed below [33].
- COCOMO—Boehm: This is probably the best known and most thoroughly documented of all software cost estimating models. It provides three levels of models: basic, intermediate, and detailed. Boehm identifies three modes of product development organic, semidetached, and embedded—that aid in determining the difficulty of the project. The developmental effort equations are all of the form: E = aS b m , where a and b are constants determined for each mode and model level;S is the value of source LOC; and m is a composite multiplier, determined from 15 cost-driver attributes. Boehm suggests that the detailed model will provide cost estimates that are within 20% of actual values 70% of the time, or PRED (0.20) = 0.70 [1].
- **SOFTCOST—Tausworthe:** Tausworthe, of the Jet Propulsion Laboratory, attempted to develop a software cost estimation model using the best features of other relatively successful models available at the time. His model incorporates the quality factors from Walson-Felix and the Rayleigh model of Putnam, among other features. It requires a total of 68 input parameters, whose values are deduced from the user's response to some 47 questions about the project. Latest reports suggest that this model has not been tested or calibrated adequately to be of general interest [35, 33].

SPQR Model—Jones: T. Capers Jones has developed a software cost estimation model called the Software Productivity, Quality, and Reliability (SPQR) model. The basic approach is similar to that of Boehm's COCOMO model. It is based on 20 reasonably well-defined and 25 not-so-well-defined factors that influence software costs and productivity. SPQR is a commercial product, but it is not as thoroughly documented as some other models. The computer model requires user responses to more than 100 questions about the project in order to formulate the input parameters needed to compute development costs and schedules. Jones claims that it is possible for a model such as SPQR to provide cost estimations that will come within 15% of actual values 90% of the time, or PRED(0.15) = 0.90 [2].

COPMO—Thebaut: Thebaut proposed a software development model that attempts to account specifically for the additional effort required when teams of programmers are involved on large projects. Thus, the model is not appropriate for small projects. The general form of the equation for the effort, E, is assumed to be: E = a + bS + cPd, where a, b, c, and

ISSN: 2277-3754
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 1, Issue 2, February 2012

d are constants to be determined from empirical data via regression analysis; S is the program size, in thousands of LOC; and P is the average personnel level over the life of the project. Unfortunately, this model requires not one but two input parameters whose actual values are not known until the project has been completed. Furthermore, the constants b and c are dependent upon the complexity class of the software, which is not easily determined. This model presents an interesting form, but it needs further development and calibration to be of widespread interest. In view of its stage of development, no estimates of its predictive ability are in order [33, 34].

ESTIMACS—Rubin: Rubin has developed a proprietary software estimating model that utilizes gross business specifications for its calculations. The model provides estimates of total development effort, staff requirements, cost, risk involved, and portfolio effects. At present, the model addresses only the development portion of the software life cycle, ignoring the maintenance or post-deployment phase. The ESTIMACS model addresses three important aspects of software management—estimation, planning, and control. The ESTIMACS system includes the following modules:

• System development effort estimator.

This module requires responses to 25 questions regarding the system to be developed, development environment, etc. It uses a database of previous project data to calculate an estimate of the development effort.

• Staffing and cost estimator.

Inputs required are: the effort estimation from above, data on employee productivity, and salary for each skill level. Again, a database of project information is used to compute the estimate of project duration, cost, and staffing required.

• Hardware configuration estimator.

Inputs required are: information on the operating environment for the software product, total expected transaction volume, generic application type, etc. Output is an estimate of the required hard ware configuration.

• Risk estimator.

This module calculates risk using answers to some 60 questions on project size, structure, and technology. Some of the answers are computed automatically from other information already available.

• Portfolio analyzer.

This module provides information on the effect of this project on the total operations of the development organization. It provides the user with some understanding of the total resource demands of the projects.

The ESTIMACS system has been in use for only a short time. In the future, Rubin plans to extend the model to include the maintenance phase of the software life cycle. He claims that estimates of the total effort are within 15% of actual values [30]. The ESTIMACS model is compared with the GECOMO, JS-2, PCOC, SLIM, and SPQR/10 models in [30]

• **Reliability Models:** A number of dynamic models of software defects have been developed. These models attempt to describe the occurrence of defects as a function of time, allowing one to define the reliability, R, and mean time to failure, MTTF. One example is the model described by Musa, which, like most others of this type, makes four basic assumptions:

- Test inputs are random samples from the input environment.
- All software failures are observed.
- Failure intervals are independent of each other.
- Times between failures are exponentially distributed.

Based upon these assumptions, the following relationships can be derived (t) = D (1 − e −bct), where D is the total number of defects; b, c are constants that must be determined from historical data for similar software; d(t) is the number (cumulative total) of defects discovered at time t: MTTF(t) = e bct / cD As in many other software models, the determination of b, c and D is a nontrivial task, and yet a vitally important one for the success of the model [24, 25]. Outlined below is a general procedure for implementing an organizational software metrics program. The details of implementing such a program will, of course, vary significantly with the size and nature of the organization. However, all of the steps outlined are necessary, in one form or another, to achieve a successful implementation of a metrics program. The implementation plan that follows is presented as a sequence of distinct steps. In practice, the application of this plan will probably involve some iteration between steps, just as occurs with the application of specific software development life cycle models. Although it is not stated explicitly, those responsible for establishing the metrics program must be concerned at each step with communicating the potential benefits of the program to all members of the organization and selling the organization on the merits of such a program. Unless the organization as a whole understands and enthusiastically supports the idea, the program will probably not achieve the desired results.

Planning Process is the implementation of a metrics program requires careful planning.

Defining Objectives: What are the objectives of the proposed program? What is it supposed t o achieve, and how?

A specific approach which can be used to plan the software metrics program is the Goal/Question/ Metric (GQM) paradigm developed by Basili,et al. [28]. This paradigm consists of identifying the goals to be achieved by the metrics program and associating a set of related questions with each goal. The answers to these questions should make it possible to identify the quantitative measures that are necessary to provide the answers and, thus, to reach the goals.

1. Initial Estimates of Effort and Cost: Metrics programs are not free; they may require major commitments of resources. For this reason, it is especially important to sell the idea of such a program to top management. Estimates of program costs should be made early in the planning process, even though they may be very crude, to help the organization avoid major surprises later on. These effort/cost estimates will need continuous refinement as the project proceeds.

(i) Initial Implementation

What are the costs associated with the initial start-up of the program?

(ii) Continuing Costs

ISSN: 2277-3754
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 1, Issue 2, February 2012

Organizations must expect to incur continuing costs to operate the metrics program. These
include, for example, the costs of collecting and analyzing data and of maintaining the metrics database.

2. Selection of Model and Metrics: A specific model and set of metrics is selected, based upon the objectives defined and cost considerations identified. Given a choice of several models that seem capable of meeting the objectives and cost requirements, the simplest model that is not intuitively objectionable should be chosen. The GQM paradigm provides a practical procedure for the selection of software metrics [30]. Important considerations in this selection process are the following items.

a. Projected Ability to Meet Objectives

Metrics and models available should be compared with respect to their apparent ability to meet the objectives (goals) identified.

b. Estimated Data Requirements and Cost

Models identified as capable of meeting the objectives of the organization should be compared in terms of data requirements and associated cost of implementation. As indicated above, parsimonious models are to be preferred, if adequate.

3. Data Requirements and Database Maintenance: Once a specific model has been chosen, the data requirements and cost estimates must be spelled out in detail and refined. At this point, care must be taken to collect enough, but not too much data. Often, the initial reaction is to collect masses of data without regard to how it will be used, "just in case it might be useful." The result is usually extinction by drowning in data. For this part of the task, the work of Basili and Weiss on collecting data for software engineering projects may be especially helpful com [28]. Steps include the following considerations:

a. Specific Data Required: Data must be gathered throughout the software life cycle. The specific information required at each phase must be identified.

b. Data Gathering Procedures: Once the necessary data have been identified, the specific methods and procedures for gathering the data must be described, and responsible personnel identified.

c. Database Maintenance: The database of metric data becomes an important corporate resource. Funds, procedures, and responsibilities for its maintenance must be spelled out.

d. Refined Estimates of Efforts and Costs: The information generated in the preceding steps should now make it possible to compute fairly accurate estimates of the effort and costs involved implementing and continuing the software metrics program.

4. Initial Implementation and Use of the Model: Assuming that the above steps have been carried out successfully and that the estimated costs are accept able, the program can now be initiated. The following items should be re-emphasized at this time

a. Clarification of Use: The intended use of the metrics program should have been made clear early on. However, it is appropriate to restate this clearly when the program is initiated. Are the metrics to be used only for project management purposes? What about their use as tools for evaluating personnel? Responsible Personnel Specific obligations of personnel who gather, maintain, or analyze the data should be made very clear. It is impossible to collect some types of data after the project has been completed.

5. Continuing Use and Refinement: For the metrics program to be successful it must be continuously applied and the results reviewed periodically. The following steps are involved:

a. Evaluating Results should be carefully summarized and compared with what actually happened. This is often not done because "there wasn't enough time."

b. Adjusting the Model Most models in use require a calibration process, adapting the values of multiplicative constants, etc., to the empirical data for the environment of application. Based upon the results achieved, these calibration constants should be reviewed and adjusted, if appropriate, especially over a long period of use, during which the environment itself may change significantly.

III. Implementation of a Metrics Program

There is growing evidence, both from university research and from industry experience, that the conscientious application of software metrics can significantly improve our understanding and management of the software development process. For example, a number of software estimating models have been developed to aid in the estimation, planning, and control of software projects. Generally, these models have been developed by calibrating the estimating formulas to some existing database of previous software project information. For new projects that are not significantly different from those in the database, reasonably accurate predictions (say, ±20%) are often possible [1,2].However, numerous studies have shown that these models cannot provide good estimates for projects that may involve different environments, languages, or methodologies [29,30]. Thus, great care must be taken in selecting a model and recalibrating it, if necessary, for the new application environment. The selective application of software metrics to specific phases of the software development cycle can also be productive. For example, certain complexity metrics have been shown to be useful in guiding software design or redesign (maintenance) activities [8, 14, 26]. Encouraging reports on the use of metrics are coming from industry also. A recent example is that of Grady and Caswell on the experience of Hewlett-Packard [42]. They describe HP's experience implementing a corporate-wide software metrics program designed to improve project management, productivity, and product quality. The program described appears to probe helping to achieve these goals, both in the short run (on individual projects) and in the long run (with improved productivity on future projects). This program may serve as a model for other organizations interested in improving their software development results. The HP experience in establishing an organizational software metrics program provides a number of useful insights, including the following:

ISSN: 2277-3754
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 1, Issue 2, February 2012

• In addition to planning carefully for the technical operation of the metrics program, the idea of such a program must be "sold" to all individuals involved, from top management, who must find the resources to support it, to entry level programmers, who may feel threatened by it. Although some short-range benefits may be realized on current projects, the organization should expect to collect data for at least three years before the data are adequate for measuring long-term trends. Outlined below is a general procedure for implementing an organizational software metrics program. The details of implementing such a program will, of course, vary significantly with the size and nature of the organization. However, all of the steps outlined are necessary, in one form or another, to achieve a successful implementation of a metrics program. The implementation plan that follows is presented as a sequence of distinct steps. In practice, the application of this plan will probably involve some iteration between steps, just as occurs with the application of specific software development life cycle models. Although it is not stated explicitly, those responsible for establishing the metrics program must be concerned at each step with communicating the potential benefits of the program to all members of the organization and selling the organization on the merits of such a program. Unless the organization as a whole understands and enthusiastically supports the idea, the program will probably not achieve the desired results.

1. **Planning Process** The implementation of a metrics program requires careful planning.

a. Defining Objectives: What are the objectives of the proposed program? What is it supposed to achieve, and how? A specific approach which can be used to plan the software metrics program is the Goal/Question/ Metric (GQM) paradigm developed by Basili, et al. [28]. This paradigm consists of identifying the goals to be achieved by the metrics program and associating a set of related questions with each goal. The answers to these questions should make it possible to identify the quantitative measures that are necessary to provide the answers and, thus, to reach the goals.

b. Initial Estimates of Effort and Cost: Metrics programs are not free; they may require major commitments of resources. For this reason, it is especially important to sell the idea of such a program to top management. Estimates of program costs should be made early in the planning process, even though they may be very crude, to help the organization avoid major surprises later on. These effort/cost estimates will need continuous refinement as the project proceeds.

(i) Initial Implementation

What are the costs associated with the initial start-up of the program?

(ii) Continuing Costs

Organizations must expect to incur continuing costs to operate the metrics program. These include, for example, the costs of collecting and analyzing data and of maintaining the metrics database.

2. Selection of Model and Metrics: A specific model and set of metrics is selected, based upon the objectives defined and cost considerations identified. Given a choice of several models that seem capable of meeting the objectives and cost requirements, the simplest model that is not intuitively objectionable should be chosen. The GQM paradigm provides a practical procedure for the selection of software metrics [28]. Important considerations in this selection process are the following items.

a. Projected Ability to Meet Objectives: Metrics and models available should be compared with respect to their apparent ability to meet the objectives (goals) identified.

b. Estimated Data Requirements and Cost: Models identified as capable of meeting the objectives of the organization should be compared in terms of data requirements and associated cost of implementation. As indicated above, parsimonious models are to be preferred, if adequate.

3. Data Requirements and Database Maintenance: Once a specific model has been chosen, the data requirements and cost estimates must be spelled out in detail and refined. At this point, care must be taken to collect enough, but not too much data. Often, the initial reaction is to to collect masses of data without regard to how it will be used, "just in case it might be useful." The result is usually extinction by drowning in data. For this part of the task, the work of Basili and Weiss on collecting data for software engineering projects may be especially helpful com [28]. Steps include the following considerations:

a. Specific Data Required: Data must be gathered throughout the software life cycle. The specific information required at each phase must be identified.

b. Data Gathering Procedures: Once the necessary data have been identified, the specific methods and procedures for gathering the data must be described, and responsible personnel identified.

c. Database Maintenance: The database of metric data becomes an important corporate resource. Funds, procedures, and responsibilities for its maintenance must be spelled out.

d. Refined Estimates of Efforts and Costs: The information generated in the preceding steps should now make it possible to compute fairly accurate estimates of the effort and costs involved implementing and continuing the software metrics program.

4. Initial Implementation and Use of the Model: Assuming that the above steps have been carried out successfully and that the estimated costs are accept able, the program can now be initiated. The following items should be re-emphasized at this time

a. Clarification of Use: The intended use of the metrics program should have been made clear early on. However, it is appropriate to restate this clearly when the program is initiated. Are the metrics to be used only for project management purposes? What about their use as tools for evaluating personnel? Responsible Personnel Specific obligations of personnel who gather, maintain, or analyze the data should be made very clear. It is impossible to collect some types of data after the project has been completed.

5. Continuing Use and Refinement: For the metrics program to be successful it must be continuously applied and

ISSN: 2277-3754
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 1, Issue 2, February 2012

the results reviewed periodically. The following steps are involved:

a. Evaluating Results: Results should be carefully summarized and compared with what actually happened. This is often not done because "there wasn't enough time."

b. Adjusting the Model Most models in use require a calibration process, adapting the values of multiplicative constants, etc., to the empirical data for the environment of application. Based upon the results achieved, these calibration constants should be reviewed and adjusted, if appropriate, especially over a long period of use, during which the environment itself may change significantly.

IV. Conclusion

In this paper we present various measurement testing technique and after the study of various testing measurement technique we think now it is requirement of a testing measurement technique which should be in terms of multi agent system technique. In coming future we need a multi agent system based software testing. So, in the next part of our research we will deals integrated concepts of Multi agent System testing technique (MASTT).

References

[1] Boehm, B. W. Software Engineering Economics. Englewood Cliffs, N. J.: Prentice-Hall, 1981.

[2] Jones, T. C. Programming Productivity. New York: McGraw-Hill, 1986.

[3] Li, H. F. and W. K. Cheung. "An Empirical Study of Software Metrics." IEEE Trans. Software Eng. SE-13, 6 (June 1987), 697-708.

[4] Rodriguez, V. and W.-T. Tsai. "Software Metrics Interpretation through Experimentation." Proc. COMPSAC 86. Washington, D. C.: IEEE Computer Society Press, Oct. 1986, 368-374.

[5] Card, D. N. and W. W. Agresti. "Measuring Software Design Complexity." J. Syst. and Software 8, 3 (June 1988), 185-197.

[6] Harrison, W. and C. Cook. "A Micro/Macro Measure of Software Complexity." J. Syst. and Software 7, 3 (Sept. 1987), 213-219.

[7] McCabe, T. J. "A Complexity Measure." IEEE Trans. Software Eng. SE-2, 4 (Dec. 1976), 308-320.

[8] Kafura, D. and J. Canning. "A Validation of Software Metrics Using Many Metrics and Two Resources." Proc. 8th Intl. Conf. on SoftwareEngineering. Washington, D. C.: IEEE Computer Society Press, 1985, 378-385.

[9] Halstead,M. H. Elements of Software Science. New York: Elsevier North-Holland, 1977.

[10] Myers, G. J. "An Extension to the Cyclomatic Measure of Program complexity." ACM SIGPLAN Notices 12, 10 (Oct. 1977), 61-64.

[11] Stetter, F. "A Measure of Program Complexity." Computer Languages 9, 3-4 (1984), 203-208.

[12] Woodward, M. R., M. A. Hennell, and D. Hedley. "A Measure of Control Flow Complexity in Program Text." IEEE Trans. Software Eng. SE-5, 1 (Jan.1979), 45-50.

[13] Henry, S., D. Kafura, and K. Harris. "On the Relationships among Three software Metrics."Performance Eval. Rev. 10, 1 (Spring 1981), 81-88.

[14] Rombach, H. D. "A Controlled Experiment on the Impact of Software Structure on Maintainability." IEEE Trans. Software Eng. SE-13, 3 (March 1987), 344-354.

[15] Potier, D., J. L. Albin, R. Ferreol, and A. Bilodeau."Experiments with Computer Software Complexity and Reliability." Proc. 6th Intl. Conf. on Software Engineering. New York: IEEE, Sept. 1982, 94-103.

[16] Potier, D., J. L. Albin, R. Ferreol, and A. Bilodeau."Experiments with Computer Software Complexity and Reliability." Proc. 6th Intl. Conf. on Software Engineering. New York: IEEE, Sept. 1982, 94-103.

[17] Levitin, A. V. "How to Measure Software Size, and How Not To." Proc. COMPSAC 86. Washington, D. C.: IEEE Computer Society Press, Oct. 1986, 314-318.

[18] Shen, V. Y., T. J. Yu, S. M. Thebaut, and L. R. Paulsen. "Identifying Error-Prone Software—An Empirical Study." IEEE Trans. Software Eng. SE-11, 4 April 1985), 317-324.

[19] Christensen, K., G. P. Fitsos, and C. P. Smith. "A Perspective Software Science." IBM SystemsJ. 20, 4 (1981), 372-387.

[20] McCall, J. A., P. K. Richards, and G. F. Walters. Factors in Software Quality, Vol. I, II, III: FinalTech. Report. RADC-TR-77-369, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, N. Y., 1977.

[21] Cerino, D. A. "Software Quality Measurement Tools and Techniques." Proc. COMPSAC 86. Washing ton, D. C.: IEEE Computer Society, Oct. 1986, 160-167.

[22] Curtis, B., S. B. Sheppard, and P. Milliman. "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics." Proc. 4th Int. Conf. on Software Engineering. New York: IEEE, Sept. 1979, 356-360.

[23] Potier, D., J. L. Albin, R. Ferreol, and A. Bilodeau. "Experiments with Computer Software Complexity and Reliability."
Proc. 6th Intl. Conf. on Software Engineering. New York: IEEE, Sept. 1982, 94-103.

[24] Ruston, H. (Workshop Chair). Workshop on Quantitative Software Models for Reliability, Complexity and Cost: An Assessment of the State of the Art. New York: IEEE, 1979.

[25] Musa, J. D., A. Iannino, and K. Okumoto. Software Reliability: Measurement, Prediction, Application. New York: McGraw-Hill, 1987.

[26] Yau, S. S. and J. S. Collofello. "Design Stability Measures For Software maintenance." IEEE Trans. Software Eng. SE-11, 9 (Sept. 1985), 849-856.

[27] Rombach, H. D. "A Controlled Experiment on the Impact of Software Structure on Maintainability." IEEE Trans. Software Eng. SE-13, 3 (March 1987), 344-354.

[28] Basili, V. R. and D. M. Weiss. "A Methodology For proCollecting Valid Software Engineering Data." IEEETrans.Software Eng. SE-10, 6 (Nov. 1984), 728-738.

[29] Kemerer, C. F. "An Empirical Validation of Software Cost Estimation Models." Comm. ACM 30, 5(May 1987), 416-429.

ISSN: 2277-3754
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 1, Issue 2, February 2012

[30] Rubin, H. A. "Macro-Estimation of Software Development Parameters: The ESTIMACS System."Proc. SOFTFAIR: A Conference on Software Development Tools, Techniques, and Alternatives. New York: IEEE, July 1983, 109-118.

[31] Wolverton, R. W. "The Cost of Developing Large Scale Software." IEEE Trans. Computers C-23, 6 (June 1974), 615-636. Reprinted in [Putnam80], 282-303.

[32] Walston, C. E. and C. P. Felix. "A Method of Programming Measurement and Estimation." IBM Systems J. 16, 1(1977), 54-73. Reprinted in [Putnam80], 238-257.

[33] Conte,S. D., H. E. Dunsmore, and V. Y. Shen. Software Engineering Metrics and Models. Menlo Park, Calif.: Benjamin/Cummings, 1986.

[34] Thebaut, S. M. and V. Y. Shen. "An Analytic Resource Model For Large-Scale software Development." Information Processing and Management 20,1-2 (1984), 293-315.

[35] Tausworthe, R. C. Deep Space Network Software Cost Estimation Model. TR #81-7, Jet Propulsion Lab, Pasadena, Calif., 1981.

[36] Source Lines of Code, and Development Effort Prediction: A Software Science Validation." IEEE Trans. Software Eng. SE-9, 6 (Nov. 1983), 639-648.

[37] Woodfield, S. N., V. Y. Shen, and H. E. Dunsmore. "A Study of Several Metrics for Programming Effort." J. Syst. and Software 2, 2 (June 1981), 97-103.

[38] Harrison, W., K. Magel, R. Kluczny, and A. DeKock. "Applying Complexity Metrics to Program Maintenance." Computer 15, 9 (Sept. 1982), 65-79.

[39] Putnam, L. H. Tutorial, Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers. New York: IEEE, 1980.

[40] Jones, T. C. Programming Productivity. New York: McGraw-Hill, 1986.

[41] Grady, R. B. and D. R. Caswell. Software Metrics: Establishing a Company-Wide Program. Engle-wood Cliffs, N. J.: Prentice-Hall, 1987.

[42] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

AUTHORS PROFILE

Dr. C. S Lamba has PhD degree in Computer Science and currently working as a Head of department, RIET Jaipur Rajasthan India. He has more than 14 years of experience and holding diversified knowledge in the field of Information technology. He has published and Participated research paper / article in various conference / seminar globally.

Reema Ajmera received her M.Tech (C.S.) degree from Banasthali Vidhypeeph in 2007. She has some important books as a author as well as currently she is associating with the jaipur national university at department of computer and system sciences. She has also attended conference and seminar at repute.

Ripu R Sinha did Master Diploma in Software Technology (MDST) from IEC Software Ltd New Delhi in 1997. Then after he got Bachelor degree in Computer Application in 2001 from St Xavier College of Computer Studies Patna, India. He is M.C.A from India Gandhi National Open University India, in 2003, and currently pursuing Ph.D. degree in Computer Science engineering from National Institute of Medical Science University Jaipur India. He is an active member of International association of Engineers Hong Kong and International Association of Computer Science and Information Technology Singapore. He is having experience of SAP HR Consultant for ERP Technologies, Manager Technical recruitment from Tire 1 Companies like SAP Labs, IBM, NIIT Technologies, Accenture for bestinfo system ltd. Academic head and Sr. Teaching staff cum coordinator at Universal Institute of Computer technologies Delhi and Currently associate with Jaipur National University as a Assistant Professor at department of Computer and System Sciences. His research interests include Knowledge Management, Enterprise resource Planning, Software Engineering Practices and Project Management. He has published and Participated research paper / article in various conference / seminar worldwide. At present, He is engaged