Software Measurement and Software Metrics in Software Quality

Ming-Chang Lee[1] and To Chang[2]

[1]*National Kaohsiung University of Applied Science, Taiwan*
[2]*Shu-Te University, Taiwan*

[1]*ming_li@mail2000.com.tw*, [2]*Changt@mail.stu.edu.tw*

Abstract

Software measurement process must be a good oriented methodical process that measures, evaluates, adjusts, and finally improves the software development process. The main contribution of this work is the easy and extensible solution to software quality of validation and verification in software develop process. Therefore, we use formal approaches in order to describe the fundamental aspects of the software. This formalization supports the evaluation of the metrics or measurement level themselves. We discuss several metrics in each of five types of software quality metrics: product quality, in-process quality, testing quality, maintenance quality, and customer satisfaction quality.

Keywords: *Software metrics; software quality; software measurement*

1. Introduction

Software measurement process must be a good oriented methodical process that measures, evaluates, adjusts, and finally improves the software development process (Shanthi and Duraiswamy, 2011). All through the entire life cycle phase, quality, progress, and performance are evaluating utilizing the measure process (Liu *et al.*, 2008). Software measurement has become a key aspect of good software engineering practice (Farooq and Quadri, 2011). Software metrics deals with the measurement of software product and software product development process and it guides and evaluating models and tools (Ma *et al.*, 2006).

Metrics are managements of different aspects of an endeavor that help us determine whether or not we are progressing toward the goal of that endeavor. Many software measures activities have been proposed in the literature, some of them are (Baumert and McWihinnet, 1992; Hammer *et al.*, 1998; Janakiram and Rajasree, 2005; Loconsole, 2001; Paulk *et al.*, 1993). Software metrics can be classified under different categories although same metrics may belong to more than category. Table 1 lists some notable software metrics that are broken up into five categories (1) Commercial perspective (2) Significance perspective (3) Observation perspective (4) Measurement perspective (5) Software development perspective (Farooq and Quadri, 2011)

The definition of a framework for the study of metrics is to examine the influence of classes of metrics upon each other (Woodings and Bundell, 2001). In this work uses the framework to discuss the metrics of software quality. Distinguishing between product and process metrics has now become a well established practice. The attributes of the product supporting services (Maintenance) influence the mode of usage and degree of acceptance by the customer. A hierarchy of levels of influences in software production is depicted in Figure 1. Section 2 starts some formal approaches of software measurement in order to describe the fundamental aspects of the software. It includes functional approach, structure-based approach, information theoretic approach, and method of statistical analysis. In Section 3, we use the

fundamental aspects of the software to discuss software quality metrics. Finally, the conclusions are summed in Section 4.

Table 1. Lists some Notable Software Metrics

Perspective	Metrics	Description
Commercial	Technical metrics	To determine whether the code well-structured, the hardware and software are adequate, the documentation is complete, correct, and up to date.
	Defect metrics	Defect metrics
	End-user satisfaction metrics	End-user satisfaction metrics are used to describe the value received from using the system.
	Warranty metrics	These metrics are influenced of by the level of defects, willingness of users to come forthwith complaints, and the willingness and ability of the software developer to accommodate the user.
	Reputation metrics	Reputation metrics are used to assess perceived user satisfaction with the software and may generate the most value, since it can strongly influence with software is acquire.
Significance	Core metric	Core metric is a required metric that is essential to support solution delivery test management on systems development projects. Example, percentage of requirement met.
	Non-core metric	Non-core metric is an optional metric that can help to create a more balanced picture of the quality and effectiveness of test efforts. Example, total number of defects by test phase.
Observation	Primitive metric	The metrics can be directly observed, such as the program size (in LOC), number of defects observed in unit testing, or total development time for project (Millers, 1988)
	Computed metric	The metrics cannot be directly observed but are computed in some manner from other metrics. E.g. LOC produced per person-month (LOC /person-month), or for product quality, such as the number of defects per thousand lines of code (defects/ KLOC) (Millers, 1988).
Measurement	Direct metric	Direct measurement is assessment of something existing (Futrell et al.2002). E. g. number of line code.
	Indirect metric	A calculation involving other attributes or entities by using some mathematical model.
Software development	Process metric	Process metrics are measure of the software development process, such as overall development time, the average level of experience of the programming staff, or type of methodology used.
	Product metric	Product metrics are measures of the software product at any stage of its development, from requirements to installed system. Product metrics may measure the complexity of the software design, the size of the final program, or the number of pages of documentation production.
	Test metric	The test process metrics provide information about preparation for testing, test execution and test progress. Some test product metrics are number of test cases design, % of test cases execution, or % test cases failed. Test product metrics provide information of about the test state

		and testing status of a software product and are generated by execution and code fixes or deferment. Some rest product metrics are Estimated time for testing, average time interval between failures, or time remaining to complete the testing.
	Maintenance metric	The software maintenance phases the defect arrivals by time interval and customer problem calls. The following metrics are therefore very important: Fix backlog and backlog management index, fix response time and fix responsiveness, percent delinquent fixes, and fix quality.
	Subjective metric	Subjective metrics may measure different values for a given metric, since their subjective judgment is involved in arriving at the measured value. An example of a subjective product metric is a classification of the software as "organic" ,"semi-detached" or "embedded" as required in the COCOMO cost estimation model (Boehm, 1981).

Source: Farooq and Quadri (2011)

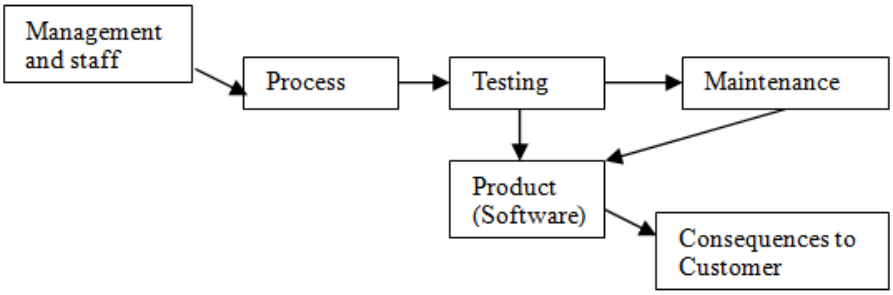

Figure 1. A Hierarchy of Levels of Influences in Software Production

2. Some of the Formal Approaches of Software Measurement

2.1. Axiomatic Approaches of Software Measurement

The measure value of a program was executed by use of three axioms (the sequence, the selection, and the repetition) (Prather, 1984; Tian and Zelkowitz 1995). The first application of measure theory consequently is Zuse and Bolimam (1989). The main idea is the definition of empirical relational system and a numerical relationship system (Zuse and Bolimam 1992).

2.2. Functional Approach of Software Measurement

2.2.1. Halsted's software science: A lexical analysis of source code was intended by Halsted (1977).

The measure of vocabulary: $n = n_1 + n_2$

Halstead defined the following formulas of software characterization for instance.

Program length: $N = N_1 + N_2$

Program volume: $V = N \log_2 \eta$

Program level: $L = \frac{V^*}{V}$

Where n_1 = the number of unique operators

n_2 = the number of unique operand
N_1 = the total number of operators
N_2 = the total number of operands

The best predictor of time required to develop and run the program successfully was Halstead's metric for program volume. Researchers at IBM (Christensen et al. 1988) have taken the idea further and produced a metrics called difficulty. V^* is the minimal program volume assuming the minimal set of operands and operators for the implementation of given algorithm:

Program effort: E = $\frac{V}{L}$

Difficulty of implementation: D = $\frac{n_1 N_2}{2n_2}$

Programming time in seconds: T = $\frac{E}{S}$

Difficulty: $n_1/2 + N_2/n_2$

With S as the Stroud number ($5 \le S \le 20$) which is introduced from the psychological science. Based on difficulty and volume Halstead proposed an estimator for actual programming effect, namely

Effort = difficulty * volume

2.2.2. Complexity Metrics: The lines of code (LOC) metric have also been proposed as a complexity metrics. McCable (McCable, 1976) has proposed a complexity metric on mathematical graph theory. The complexity of a program is defined in terms of its control structure and is represented by the maximum number of "linearly independent" path through the program. The formulas for the cyclomatic complexity proposed by McCable are:

V(G)= e - n + 2p

Where e = the number of edges in the graph
n = the number of nodes in the graph
P = the number of connected components in the graph.

According to Arthur (Arthu, 1985) the Cyclomatic complexity metric is based on the number of decision elements (IF-THEN-ELSE, DO WHILE, DO UNTIL, CASE) in the language and the number of AND, OR, and NOT phrases in each decision. The formula of the metric is: Cyclomatic complexity = number of decisions +number of conditions + 1. The calculation counts represent "the total number of structure test paths in the program" and "The number of the logic in the program". Information flow metric describes the amount of information which flows into and out of a procedure. The complexity of a procedure p definer as: (Lewis and Henry, 1990).

$$c_p = (fan-in * fan-out)^2$$

Where Fan-in: The number of local flows into a procedure plus the number of global data structures from which a procedure retrieves information.

Fan-out: The number of local flows into a procedure plus the number of global data structures from which a procedure updates.

2.2.3. Reliability Metrics: A varies often used measure of reliability and availability in computer-based system is mean time between failures (MTBF). The sum of mean time to failure (MTTF) and mean time to repair (MTTR) gives the measure, *i.e.*

$$MTBF = MTTF + MTTR$$

The availability measure of software is the percentage that a program is operating according to requirement at a given time and is given by the formula:

$$Availability = MTTP / (MTTF + MTTE) * 100\%$$

The reliability growth models assume in general that all defects during the development and testing phases are correct, and new errors are not introduced during theses phases. All models seem to include some constraints on the distribution of defects or the hazard rate, *i.e.* defect remaining in the system.

2.2.4. Readability Metrics: Walston and Fellx (1977) give a ration of document pages to LOC as

$$D = 49L^{1.01}$$

Where D= number of pages of document
L = number of 1000 lines of code.

2.2.5. Error Prediction Metrics: Halstead's program volume as base for her prediction of the number of error $B_1 = Volume / 3000$ found during the validation phase. She also gives in approximation of total number of error found during the entire development process as $B_2 = Volume / 750$

2.3. Structure-based Approach of Software Measurement

McCabe investigated the flow graph G in order to estimate the test paths including the test effort (McCabe, 1976). The Cyclomatic number is:

$$V(G) = no.of\ edges - no.of\ nodes + 2 \times connectedcomponent$$

The approach of Fenton and Pfleeger (1997) is based in the flow graphs based on the following Dijkstra structure.
The Cyclomatic numbers are V (a) = V (b) =V(c) = V (d) =2 and V (e) = n-1

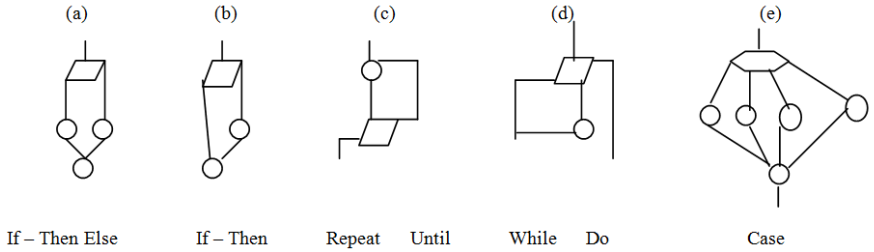

Figure 2. The Dojkstra Graphs for the Structure Programming

2.4. Information Theoretic Approach of Software Measurement

The design of software is often depicted by graphs that show components and their relationships. For example, a structure chart shows the calling relationships among

components. A graph is an abstraction of a software system and a subgraph represent a module (subsystem). Some authors are used information theory in software measurement (Khoshgoftaar and Seliya, 2002; Munson, 2003). For a discrete random variable x distributed according to probability mass function p the entropy is defined as

$$H(x) = \sum_{l=1}^{n_x} p_l(-\log p_l)$$

Where l is an index over the domain of x and n_x is the cardinality of the domain x. H(x) is the average information per sample measurement object from the distribution. The probability mass function p is determined by the situation of the executed operators or functions in software system. The entropy can be very helpful consideration the dynamic software measurement.

2.5. Method of Statistical Analysis

The most used statistical methods are given in the following table (see lei and smith, 2003; pandian, 2004; Juristo and Moreno, 2003; Dumake *et al.*, 2002; Dao *et al.*, 2002; Fenton *et al.*, 2002). Some commonly used statistical methodology (include nonparametric tests) are discussed in Table 2.

Example A: Spearman rank correlation coefficient

H_0 : No population correlation between ranks

H_1 : Population correlation between ranks

Test statistic: Let the correction coefficient be r_s , and defined as $1-\frac{6}{(n^3-n)}\sum_{i=1}^{n} d_i{}^2$, where n is the sample size, and d_i is the difference in rank of the i^{th} pair data.

Decision rule: The r_s value must exceed a specified threshold, or $r_s > r_{\alpha,}$ with significance level α.

We reject H_0 , otherwise accept H_0 .

Example B: Mann-Whitency U test

H_0 : A and B population are identical

H_1 : There are some different in sample A and B

Test statistic: Let $U = \min(U_1, U_2)$. Let n_1 be the size of smallest sample and n_2 is the size of the biggest sample. R_1 and R_2 are the total ranks of each sample.
Where

$$U_1 = n_1 n_2 + \frac{n_1(n_1+1)}{2} - R_1$$

$$U_2 = n_1 n_2 + \frac{n_2(n_2+1)}{2} - R2$$

Critical value: Use the table to find the critical value for the U statistic at 5% level for sample size n_1 and n_2.

Decision rule: If $U \le U_c$ reject H_0

For example, a group of subjects had been instructed to solve the same problem (Sort Experiment) in C++, while another group of subjects had been instructed to solve the same problem in Pascal. Table 3 is showed as the ranks programming times for the Sorting Experiment.

R_1 = 183.5 (C++) and R_2 = 281.5 (PASCAL). Consequently, U_1 = 161.5 and U_2 = 63.5, leading to U = 63.5. From Critical values for the Critical Values for the Mann-Whitney table, If α = 0.05, we calculate Uc =64

U= 63.5＜(U_c =) 64, we will reject H_0.

Therefore, we can conclude the performance for the two languages are different, with H_0 rejected at the 0.05 levels.

Table 2. Some commonly used Statistical Methodology

	Type of methodology	Application
1	Ordinary least square regression models	Ordinary least square regression (OLS)model is used to subsystem defects or defect densities prediction
2	Poisson models	Poisson analysis applied to library unit aggregation defect analysis
3	Binomial analysis	Calculation the probability of defect injection
4	Ordered response models	Defect proneness
5	Proportional hazards models	Failure analysis incorporating software characteristics
6	Factor analysis	Evaluation of design languages based on code measurement
7	Bayesian networks	Analysis of the relationship between defects detecting during test and residual defects delivered.
8	Spearman rank correlation coefficient	Spearman's coefficient can be used when both dependent (outcome; response) variable and independent (predictor) variable are ordinal numeric, or when one variable is an ordinal numeric and the other is a continuous variable.
9	Pearson or multiple correlation	Pearson correlation is widely used in statistics to measure the degree of the relationship between linear related variables. For the Pearson correlation, both variables should be normally distributed
10	Mann – Whitney U test	Mann – Whitney U test is a non-parametric statistical hypothesis test for assessing whether one of two samples of independent observations tends to have larger values than the other.
11	Wald-Wolfowitz two-sample Run test	Wald-Wolfowitz two-sample Run test is used to examine whether two samples come from populations having same distribution.
11	Median test for two sample	To test whether or not two samples come from same population, median test is used. It is more efficient than run test each sample should be size 10 at least.

12	Sign test for match pairs	When one member of the pair is associated with the treatment A and the other with treatment B, sign test has wide applicability.
13	Run test for randomness	Run test is used for examining whether or not a set of observations constitutes a random sample from an infinite population. Test of randomness is of major importance because the assumption of randomness underlies statistical inference.
14	Wilcoxon signed rank test for matcher pairs	Where there is some kind of pairing between observations in two samples, ordinary two sample tests are not appropriate.
15	Kolmogorov-Smirnov test	Where there is unequal number of observations in two samples, Kolmogorov-Smirnov test is appropriate. This test is used to test whether there is any significant difference between two treatments A and B.

Table 3. The Ranks Programming Times for the Sorting Experiment

Subject	C++	Rank	PASCAL	Rank	Subject	C++	Rank	PASCAL	Rank
1	12	1	20	5	9	28	13	39	22
2	13	2	21	6.5	10	30	14	40	23
3	14	3	25	8.5	11	32	16	41	24
4	19	4	26	10	12	34	17	42	26.5
5	21	6.5	31	15	13	36	19	42	26.5
6	25	8.5	35	18	14	42	26.5	42	26.5
7	27	11.5	38	20.5	15	46	30	44	29
8	27	11.5	38	20.5					

Example C : Wald-Wolfowitz Run Test

H_0 : Two samples come from populations having same distribution

H_1 : Two samples come from populations having different distribution

Test statistic: Let r denote the number of runs. To obtain r, list the $n_1 + n_2$ observations from two samples in order of magnitude. Denote observation from one sample by x's and y's. Count the number of runs.

Critical value: Consequently, critical region for this test is always one-side. The critical value to decide whether or not the run of runs are few is obtained from the table. The table gives critical value r_c for n_1 and n_2 at 5% level of significance.

Decision rule: If $r \le r_c$ reject H_0. For sample sizes large than 20 critical value r_c is given,

$r_c = \mu - 1.96\sigma$ at 5% level of significance.

Where $\mu = 1 + \frac{2n_1n_2}{n_1 + n_2}$ and $\sigma = \sqrt{\frac{2n_1n_2(2n_1n_2 - n_1n_2)}{(n_1 + n_2)^2(n_1 + n_2 - 1)}}$

For example, a group of subjects had been instructed to solve the same problem (Sort Experiment) in C++, while another group of subjects had been instructed to solve the same problem in Pascal. For sample pieces of executive time were collected in A (Pascal), and five

sample pieces of executive time were collected in B (C++). Table 4 shows origin of piece and ranks. Table 4 is showed as the origin of piece and ranks. Table 5 is showed as the combined ordered data.

H_0 : A and B population are identical

H_1 : There are some different in sample A and B

Table 4. The Origin of Piece and Ranks

FORTRAN	Rank	PASCAL	Rank
1.96	4	2.11	6
2.24	7	2.43	9
1.71	2	2.07	5
2.42	8	1.62	1
		1.93	3

Table 5. The combined Ordered Data

Origin of piece execution time	B A B A B B A A B
Rank	1 2 3 4 5 6 7 8 9

Test statistic r = 6 (total number of runs). For $n_1 = 4$, $n_2 = 5$, $\alpha = 0.05$, critical value $r_c = 4$, we may accept the null hypothesis, and conclude that A and B have identical distribution.

3. Software Quality Metrics

3.1. Productive Quality Metrics

Lewis and Henry (1990) divide the software complexity metrics into three categories are: Code metrics, structure metrics, and hybrid metrics. Code metrics examine the internal complexity of a procedure. Example of code metrics are LOC, Healstead's software science, and McCabe's Cyclometric complexity. Structure metrics examine the relationship between a section code and the rest of the system. Example of Structure metrics are Information flow metrics. The hybrid metrics are combined the internal code metrics with of the communication connection s between the code and the rest of the system.

Example 1: Lines of Code Defect Rate

Because the LOC count is based on source instructions, then there are two size metrics are shipped source instruction (SSI) and new and changed source instructions (CSI). The relationship between the SSI and CSI count can be expressed with the following formula:
SSI (current release) = SSI (previous release) +

CSI (new and changed source instructions for release) –

Deleted code (usually very small) –

Changed code (to avoid double count in both SSI and CSI)

The several post-release defect rate metrics per thousand SSI (KSSI) or per thousand CSI (KCSI) are:

(1) Total defects pear KSSI (a measure of code quality of the total product)

(2) Field defects per KSSI (a measure of defect rate in the field)

(3) Release-origin defects (field and internal) per KCSI (a measure of development quality)

(4) Release-origin field defects per KCSI (a number of development quality per defects found by customers)

Consider the following hypothetical example:

Initial release of product X

KCSI = KSSI = 50KLOC

Defects / KCSI = 2.0

Total number of defects =2.0 x 50 = 100

Second release of product X

KCSI = 20

KSSI = 50 + 20 (new and changed lines of code) – 4 (assuming 20% are changed)

Line of codes) = 66

Defects / KCSI = 1.8 (assuming 10% improvement over the first release)

Total number of defects =1.8 x 20 = 36

Example 2: Function Points

Step 1: calculation the function counts (FCs) based on the following formula:

$$FC = \sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} \times x_{ij}$$

Where w_{ij} are the weighting factors of the five components by complexity level (low, average, high) and x_{ij} are the numbers of each component in the application. It is a weighted of five major components are:

- External input: Low complexity, 3; average complexity, 4; high complexity, 6
- External output:Low complexity, 4; average complexity, 5; high complexity, 7
- Logical internal file: Low complexity, 5; average complexity, 7; high complexity, 10
- External interface file: Low complexity, 7; average complexity, 10; high complexity, 15
- External inquiry: Low complexity, 3; average complexity, 4; high complexity, 6

Step 2: it involves a scale from 0 to 5 to assess the impact of 14 general system characteristics in terms of their likely on the application. There are 14 characteristics: data communication distributed function, heavily used configuration, transaction rate, online data entry, end user efficiency, online update, complex processing, reusability, installation ease, operational ease, multiple sites, and facilitation of change.

The scores (ranging from 0 to 5) for these characteristics are then summed, based on the following formulas, to arrive at the value adjustment factor (VAF)

$$VAF = 0.65 + 0.01 \sum_{i=1}^{14} c_i$$

Where c_i is the score for general system characteristic. The number of function points is obtained by multiplying function counts and the value adjustment factor:

$$FP = FC \times VAF$$

Example 3: by applying the defect removal efficiency to the oval defect rate per function point, the following defect rates for the delivered software were estimated. On Software Engineering Institute (SEI) capability maturity model (CMM), the estimated defect rates per function point are follows:

- SEI CMM level 1: 0.75
- SEI CMM level 2: 0.44
- SEI CMM level 3: 0.27
- SEI CMM level 1: 0.14
- SEI CMM level 1: 0.05

Example 4: A further functional approach of measurement is the function point method of Jones (1991) that was based in the execution of the unadjusted function point (UFP)

$$\text{UFP} = \text{a} \times \text{inputs} + \text{b} \times \text{outputs} + \text{c} \times \text{requires} + \text{d} \times \text{internal data} + \text{e} \times \text{external data}$$

with the factors divided in "simple", "average', and "complex" as $a \in [3,4,6]$, $b \in [4,5,7]$, $c \in [3,4,6]$, $d \in [7,10,15]$, and $e \in [5,7,10]$ for every software component. This function points can be mapped to the personal month (effort) of the software product development with the approximate execution of the person month PM related to the function points FP (Bundschuh and Fabry, 2000).

$$FP = 0.015216 FP^{1.29}$$

3.2. Process Quality Metrics

3.2.1. Defect Density during Testing: Defect rate during formal testing is usually positively correlated with the defect rate in the field. Higher defect rates found during testing in an indicator that the software has experienced higher error injection during testing effort – for example, additional testing or a new testing approach that was demand more effective in detecting defects. Some metrics for defect density during testing are:

- Error discovery rate: number of total defects found / number of test procedures execution.
- Defect acceptance: (Number of valid defects / total number of defects) * 100
- Test case defect density: (Number of failed tests / Number of executed test cases)*100

3.2.2. Defect Arrival / removal During Testing: The objective is always to look for defect arrivals that stabilize at a very low level, or times between failures that are far apart, before ending effort and releasing the software to the field. Some metrics for defect arrival during testing are:

- Bad Fix defect: defect whose resolution give rise to new defects are bad fix defect. Bad Fix defect = (Number of Bad Fix defects / Total number of valid defects)*100

◦ Defect removal effectiveness (DRE): (Defects removed during a development phase / Defects latent in the product)* 100. The denominator of the metric can only be approximated by defects removed during the phase + defects found later.

3.2.3. Metric-based Estimation Models: Most of the models presented in this subsection are estimators of the effort needed to produce a software product. Probably the best known estimation model is Boehm's COCOMO model (Boehm, 1981). The first one is a basic model which is a single-value model that computes software development effort and cost as a function of program size expressed as estimated lines of code (LOC). The second COCOMO model computes software development effort as a function of program size and a set of "coat drives" that include subjective assessment of product, hardware, personal, and project attributes. The third COCOMO model is an advanced model that incorporates all characteristics of the intermediate version with the assessment of the cost driver's impact on each step of the software engineering process.

The basic COCOMO equations are: $E = a_i (KLOC)^{b_i}$, $D = c_i E^{d_i}$

Where E is the effort applied in person-month.

D is the development time in chronological months

The coefficients a_i and c_i and the exponents b_i and d_i are given in Table 6.

Table 6. Basic COCOMO

Software project	a_i	b_i	c_i	d_i
Organic	2.4	1.05	2.5	0.36
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

The Putnam estimation model (Putnam, 1978) assumes a specific distribution of effort over the software development project. The distribution of effort can be described by the Royleigh-Norden curve. The equation is:

$$L = c_k K^{1/3} t_d 4/3$$

Where c_k is the state of technology constant,

k is the effort expended (in person-years) over the whole life cycle.

t_d is the development time in year.

The c_k valued ranging from 2000 for poor to 11000 for an excellent environment is used (Pressman, 1988).

Walston and Fellix (1977) give a productivity estimator of a similar form at their document metric. The programming productivity is defined as the ratio of the delivered source lines of code to the total effort in person-months required to produce the delivered product.

$$E = 5.2L^{0.91} \text{ E}$$

Where E is total effort in person-month

L is the number of 1000lines of code.

3.3. Software Maintenance Metrics

During the maintenance phase, the following metrics are very important:

- Fix backlog and backlog management index
- Fix response time and fix responsiveness
- Percent delinquent fixes
- Fix quality

Fix backlog is a workload statement for software maintenance. To manage the backlog of open, unresolved, problems is the backlog management index (BMI). If BMI is large then 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased.

$$BMI = \frac{Number\ of\ \ problrms closed\ during the\ month}{Number of\ \ problem arrivals\ during\ the\ month} \times 100\%$$

The fix response time metric is usually calculated as follows for all problems as well as by severity level: Mean time of all problems from open to close. A more sensitive metrics is the percentage of delinquent fix. For each fix, if the turnaround time greatly exceeds the require response time, then it is classified as delinquent:

$$Percent\ delinquent\ fixes = \frac{\begin{array}{c} number\ of\ \ fixes\ that\ exceeded\ the \\ response\ time\ criticia\ by\ severity\ level \end{array}}{\begin{array}{c} Number\ of\ \ fixes \\ delivered\ in\ a\ specified\ time \end{array}}$$

This metrics is not a metric for real-time delinquent management because it is for closed problem only.

Fix quality or the number of defective fixes is another important quality metric for the maintenance phase.

3.4. Software Testing Metrics

The software metrics that the quality assurance (QA) team procedures are connected with the test activities that are part of test phase and so are formally known as software testing metrics (Kaur *et al.*, 2007). Some testing metrics showed on Table 7 (Premal. and Kale, 2011; .Kuar *et al.*, 2007; Farooq *et al.*, 2011).

Table 7. Some Testing Metrics

1	Test metric	Definition / purpose	Formula	Effect
2	Test case Productivity (TCP)	This metrics gives the test case writing productivity based on which one can have a conclusive remark.	Total efforts took for writing / Efforts (Hours)	One can compare the test case productivity value with previous release for getting better picture.
3	Defect Acceptance (DA)	This metric determine the number of valid defects that testing team has identified during execution.	DA = Number of valid defects / Total number of defects 100%	The value of this metric can be compared with previous release for getting better picture.

4	Defect Rejection (DR)	This metric determine the number of defects rejected during execution.	DA = Number of defects rejected / Total number of defects *100%	The metric gives the percentage of the invalid defect the testing team has opened and one can control.
5	Bad Fix Defect (B)	Defect whose resolution give rise to new defect are bad fix defect. This metric determine the effectiveness of defect resolution process.	Bad Fix Defect =Number of Bad Fix Defect / Total number of valid defects *100%	This metric gives the percentage of the bad defects resolution which needs to be controlled.
6	Test case defect density	This metric may help us to know the efficiency and effectiveness of our test cases.	Test case defect density = (Number of failed tests / Number of executed test cases) *100	Higher value of this metrics indicates that the case is effective and efficient, because they are able to defect more number of defects.
7	Test Efficiency (TE)	This metric determine the efficiency of the testing team in identifying the defects. It also indicated the defects missed out during testing phase which migrated to the next phase.	Test Efficiency = $(D_T/(D_T+D_U)*100$	The higher the value of this metric, the better in the review efficiency
8	Test Effectiveness	It shows the relation between the number of defects detected during testing and the total number of defects in the product. Purpose to do deliver a high quality product.	Test Effectiveness = $(D_T/(D_F+D_U)*100$	The higher the TE, including a higher ratio of defects. Defects were detected before release; the higher is the effectiveness of the test organization to drive out defects.
9	Test improvement (TI)	It shows the relation between the number of defects detected by the test team during and the size of product release. Purpose is to deliver a high quality product.	TI = number of defects detected by the test team during / source lines of code in thousands	Defects were defected the higher the improvement to the quality of the product which can be attributed to the teams.
10	Test time over development time (TD)	It shows the relation between time spent on testing and the time spent on developing. Purpose is to decrease time-to-market.	TD = number of business days used for product testing / number of business days used for product development	The lower this number, the lower is the amount of time required by the test teams to test this product compared to the development team.
11	Test cost normalized to product size (TCS)	It shows the relation between resource or money spent on testing and the size of	TCS = total cost of testing the product in dollars / source lines of code in thousands	The lower this number, the lower is the cost required to test each thousand lines of code.

		the product release. Purpose is to decrease cost-to-market.		
12	Test cost as a ration of development cost (TCD)	It shows the relation between testing cost and development cost of the product. Purpose is to decrease cost-to-market.	TCD = total cost of testing the product in dollars / total cost of developing the product in dollars	The lower this number, the lower is the cost of finding one defects unit, and the more cost-effective is the test process.
13	Test improvement in product quality	It shows the relation between this number of defects detected and the size of the product release	Number of defects found in the product after release / source lines of code in thousands	The higher this number, the higher is the improvement of the quality of the product contributed during this test phase.
14	Test time needed normalize to size of product	It shows the relation between tine spent on testing and the size of the product release.	Number of business days used for a specific test phase / source lines of code in thousands	The lower this number, the less time required for the test phase relatively.
15	Test cost normalize to size of product	It shows the relation between resource or money spent on the test phase and the size of product release.	Total cost of a specific test phase in dollars / source lines of code in thousands	The lower this number, the lower in the cost required to test each thousand lines of code in the test phase.
16	Cost per defect unit	It shows the relation between money spent on the test phase and the number of defects detected	Total cost of a specific test phase in dollars / number of defects found in the product after release	The lower this number, the lower is the cost of finding one defect unit in the test phase, and the more effective is the test phase.
17	Test effectiveness for driving out defects in each test phase	It shows the relation between the number of one type of defects detected in one specific test phase and the total number of this type of defect in the product.	$(D_D/(D_D+D_N)*100$	The higher this number, indicating a higher ratio of defect of important defects was detected in the appropriate test phase, the higher is the effectiveness of this test phase to drive out its target type of defects.
18	Performance scripting productivity (PSP)	This metric gives the scripting productivity for performance test script and have trend over a period of time.	Total number of performed(no. of clicks, no. of input parameter, no. of correlation parameter) / effort (hours)	The higher this number, the higher is the performance of the productivity.
19	Performance execution summary	This metric gives the classification with respect to number of test conducted along the status for various types of performance testing.	Breakpoint / stress test. Volume testing Load testing	Some of types of testing are peak volume test, breakpoint / stress test.

20	Performance execution data-client side	This metric gives the detail information of client side for execution.	Unit testing Accept testing Response time	Some of the data points of this metrics is running users, response time, throughput, total transaction per second, error per second etc..
21	Performance execution data-server side	This metric gives the detail information of server side for execution.	CPU time Memory Utilization	Some of the data points of this metrics is CPU time, Memory Utilization, Data base connections per second etc.
22	Performance test efficiency (PTE)	This metric determine the quality of performance testing team in meeting the requirements which can be used as an input for further improvement.	PTE = requirement during perform test / (requirement during performance time + requirement after signoff of performance time) * 100%	Some of the requirements of performance testing are: Average response time, transaction per second, application must be able handle performance max user load, Sever stability.
23	Automation scripting productivity (ASP)	This metric gives the scripting productivity for automated test script on which can analyze and draw most effective conclusion from the same.	Total number of performed(no. of clicks, no. of input parameter, no. of checkpoint added) / effort (hours)	The higher this number, the higher is the automation scripting productivity.
24	Automatic converge	This metrics gives the percentage of mutual test cases automated.	Total number of test case automated / Total number of test case automated of manual	The higher this number, the higher is the improvement of the quality of the performance editing

D_D **:** Number of defects of this defect type that are detected after the test phase.

D_T **:** Number of defects found by the test team during the product cycle

DU **:** Number of defects of found in the product under test (before official release)

D_F **:** Number of defects found in the product after release the test phase

D_N **:** Number of defects of this defect type (any particular type) that remain uncovered after the test phase.

Example 5:

In production, average response time is greater than expected, requirement met during perform test = 4, requirement not met after signoff of perform test = 1. Consider, average response time is important requirement which has not met, then tester can open defect with severity as critical. Performance severity index = (4* 1)/1 = 4 (critical). Performance test efficiency = 4/ (4+1)* 100 = 80%.

3.5. Customer Problems Metric and Customer Satisfaction Metrics

From the customer's perspective, it is bad enough to encounter functional defects when running a business on the software. The problems metric is usually expressed in terms of problem per user month (PUM). PUM is usually calculated for each month after the software is released to market, and also for monthly averages by user. The customer problems metric can be regarded as an intermediate measurement between defects measure and customer satisfaction. To reduce customer problems, one has to reduce the functional defects in the products, and improve other factors (usability, documentation, problem rediscovery, *etc.*). To improve customer satisfaction, one has to reduce defects and overall problems. Several metrics with slight variations can be Constructed and used, depending on the purpose of analysis. For example:

- Percent of completely satisfied customers.
- Percent of satisfied customers (satisfied and completely satisfied)
- Percent of dissatisfied customers(dissatisfied and completely dissatisfied)
- Percent of non (neutral, dissatisfied, and completely dissatisfied).

Example 6:

Some companies use the net satisfaction index (NSI) to facilitate comparisons across product. The NSI has the following weight factors:

Completely satisfied = 100%, Satisfied = 75%, Neutral, = 50%, Dissatisfied = 25%, and completely dissatisfied =0.

4. Conclusions

This paper is an introduction of software quality found in the software engineering literature. Software measurement and metrics help us a lot of evaluating software process as well as the software product. The set of measures identified in this paper provide the organization with better insight into the validation activity, improving the software process towards the goal of the having a management process.

Well-designed metrics with documented objectives can help an organization obtain the information it needs to continue to improve its software product, processes, and customer services. Therefore, future research is need to extend and improve the methodology to extend metrics that have been validated on one project, using our criteria, valid measures of quality on future software project.

References

[1] L. J. Arthur, "Measuring programmer productivity and software quality", John Wiley & Son, NY, **(1985)**.
[2] J. H. Baumert and M. S. McWihinnet, "Software measurement and the capability maturity model", Software Engineering Institute Technical Report, cMMI/SEI-92-TR, ESC-TR-92-0, **(1992)**.
[3] B. W. Boehm, "Software Engineering Economics", Englewood Cliffs, NJ, Prentice Hall, **(1981)**.
[4] M. Bundschuh and A. Fabry, "Auswandschatzung von IT-project", MITP publisher, Bonn, **(2000)**.
[5] K. Christensen, G. P. Fistos and C. P. Smith, "A perspective on the software science", IBM systems Journal, **(1988)**, vol. 29, no. 4, pp. 372-387.
[6] M. Dao, M. Huchard, T. Libourel and H. Leblance, "A new approach to factorization-introducing metrics", Proceeding of the IEEE Symposium on Software Metrics METRICS, **(2002)** June 4-7, pp. 227-236.
[7] R. Dumake, M. Lother and C. Wille, "Situation and treads in Software Measurement-A statistical Analysis of the SML Metrics Biolography, Dumke / Abran: Software Measurement and Estimation, Shaker Publisher, **(2002)**, pp. 298-514.
[8] S. U. Farooq and A. M. K. Quadri, "Software measurements and metrics: Role in effective software testing", International Journal of Engineering Science and Technology, vol. 3, no. 1, **(2011)**, pp. 671-680.

[9] S. U. Farooq, S. M. K. and N. Ahmad, "Software measurements and metrics: role in effective software testing", Internal Journal of Engineering Science and Technology, vol. 3, no. 1, **(2011)**, pp. 671-680.
[10] N. E. Febton and S. L. Pfleeger, "Software Metrics-a rigorous and practical approach", Thompson Publ., **(1997)**.
[11] N. Fenton, P. Krause and M. Neil, "Software measurement: Uncertainty and causal modeling", IEEE Software, **(2002)** July-August, pp. 116-122.
[12] R. T. Futrell, F. Donald and L. S. Shafer, "Quality software project management", Prentice Hall Professional, **(2002)**.
[13] T. F. Hammer, L. J. Huffman and L. H. Rosenberg, "Doing requirements right the first time", CEOSSTALK, the journal of Defense Software Engineering, **(1998)** December, pp. 20-25,.
[14] D. Janakiram and M. S. Rajasree, "ReQuEst Requirements-driven quality estimator", ACM SIGSOFT software engineering notes, vol. 30, no. 1, **(2005)**.
[15] N. Juristo and A. M. Moreno, "Basic of software Engineering Experimentation", Kluwer Academic, publisher, Boston, **(2003)**.
[16] A. Kaur, B. Suri and A. Sharma, "Software testing product metrics-a survey", Proceedings of National Conference on Challenges & Opportunities in Information Technology (COIT-2007), RIMT-IET, Mandi Gobindgarh, **(2007)** March 23.
[17] T. M. Khoshgoftaar and N. Seliya, "Three-based Software Quality Estimation Models for Fault, Prediction, "Proceeding of eight IEEE Symposium on Software Metrics (METRICS 2002), **(2002)** June 4-7, Ottawa, pp. 203-215.
[18] M. Khraiwesh, "Validation measure in CMMI", World in computer science and information Technology Journal, vol. 1, no. 2, **(2011)**, pp. 26-33.
[19] A. Kuar, B. Suri and A. Sharma, "Software testing product metrics-A Survey", Proc. of national Conference in Challenges & Opportunities in Information Technology, RIMT-JET, Mandi Gobindgarti, **(2007)** March 23.
[20] S. Lei and M. R. Smith, "Evaluation of several non-paramedic bootstrap methods to estimate Conference Interval for Software Metrics", IEEE Trabsactions on Software Engineering, vol. 29, no. 1, **(2003)**, pp. 996-1004.
[21] J. A. Lewis and S. M. Henry, "On the benefits and difficulties of a maintainability via metrics methodology", Journal of Software maintenance, Research and Practice, vol. 2, no. 2, **(1990)**, pp. 113-131.
[22] Y. Liu, W. P. Cheah, B. K. Kim and H. Park, "Predict software failure-prone by learning Bayesian network", International Journal of Advance Science and Technology, vol. 1, no. 1, **(2008)**, pp. 35-42.
[23] A. Loconsole, "Measuring the requirements management key process area", Proceedings of ESCOM – European Software Control and Metrics Conference, London, UK, **(2001)** April.
[24] Y. Ma, K. He, D. Du, J. Liu and Y. Yan, "A complexity metrics set for large-scale object-oriented software system", Proceedings of the Sixth IEEE International Conference on Computer and Information Technology, Washington, DC, USA, **(2006)**, pp. 189-189.
[25] T. J. McCable, "A complexity measure", IEEE Transaction on Software Engineering, SE-2(4), **(1976)** December, pp. 308-320.
[26] E. E. Millers, "Software metrics SEI curriculum module SEI-CM-12", Carnegie Mellon University, Software Technology Engineering Institute, **(1988)** December.
[27] J. C. Munson, "Software engineering Measurement", CRC Press Company, Boca Raton London, NY, **(2003)**.
[28] C. R. Pandian, "Software metrics-A guide to planning, Analysis, and Application", CRC press Co., **(2004)**.
[29] M. C. Paulk, C. V. Weber, S. Garcia, M. B. Chrissis and M. Bush, "Key practices of the capability maturity model version 1.1", Software engineering institute technical report, CMU/SEI-93-TR-25vESC-TR-93-178, Pittsburgh, PA, USA, **(1993)** February.
[30] R. E. Prather, "An Axiomatic theory of software complexity measure", The Computer Journal, vol. 27, no. 4, **(1984)**, pp. 340-347.
[31] B. N. Premal and K. V. Kale, "A brief overview of software testing metrics", International Journal of Computer Science and Engineering, vol. 1, no. 3/1, **(2011)**, pp. 204-211.
[32] R. S. Pressman, "Making Software engineering happen: A Guide for instituting the technology", Prentice Hall, New Jersey, **(1988)**.
[33] L. H. Putnam, "A general empirical solution to the macro software and software sizing and estimating problem", IEEE Transaction on Software Engineering, SE-4 (4), **(1978)** July, pp. 345-361.
[34] P. M. Shanthi and K. Duraiswamy, "An empirical validation of software quality metric suits on open source software for fault-proneness prediction in object oriented system", European journal of Scientific Research, vol. 5, no. 2, **(2011)**, pp. 168-181.
[35] J. Tian and V. Zelkowitz, "Complexity measure evaluation and selection", IEEE Transaction on Software Engineering, vol. 21, no. 8, **(1995)**, pp. 641-650.
[36] D. K. Walace and R. U. Fujii, "Software verification and validation, an overview", IEEE Software, **(1989)** May, pp. 10-17.

[37] C. E. Walston and C. P. Felix, "A method of programming measurement, and estimation", IBM Systems Journal, vol. 16, **(1977)**, pp. 54-73.
[38] T. L. Woodings and G. A. Bundell, "A framework for software project metrics", Proceeding of the 12th European Conference on Software Control and Metrics, **(2001)**.
[39] H. Zuse and P. Bolimam, "Using measurement theory to describe the properties and scale of static software complexity metrics", SIGPLAN Notices, vol. 24, no. 8, **(1989)**, pp. 23-33.
[40] H. Zuse and P. Bolimam, "Measurement theory and software measure", Proceeding of the BCS-FACS workshop, London, **(1992)** May.