Effect of Different UML Diagrams to Evaluate the Size Metric for Different Software Projects

1Preety Verma Dhaka, 2Dr. Kavita
1Research Scholar, 2Associate Professor
Department of CS & IT
Jayoti Vidyapeeth Women's University, Jaipur

Abstract: **As we know that in Software Engineering, measuring the software is an important activity. For measuring the software appropriate metrics are needed. Using software metrics we are able to attain the various qualitative and quantitative aspects of software. Software Metrics are a unit of measurement to measure the software in terms of quality, size, efforts, efficiency, reliability, performance etc. Measures of specific attributes of the process, project and product are used to compute software metrics.**

INTRODUCTION

The objectives of this research are to make an empirical evaluation of software size metrics based on UML with the help of two case studies and then calculate that empirical data consisting of actual values and thereby showing that how the software size metrics will be derived from an UML model via Class Diagrams and the below listed interaction diagrams.

1. Activity Diagrams
2. State chart Diagrams
3. Component Diagrams
4. Collaboration Diagrams

For carrying out this research, two real case studies namely (i) Virtual Class Room and (ii) Data Secrecy System will be taken for practical evaluation. The UML modelling of these systems will be done and the software size metrics of these systems will be evaluated based on the UML models, using the non-functional techniques (LOC, FP, and COCOMO-II).The metrics will be specified using UML extension mechanism and then will be calculated with the help of a tool. The estimated values will be compared with the actual software. Thus, the aim of our research is to evaluate the empirical value sets of UML models and thereby, showing the use of various size metrics and validate their extraction procedure from UML design with the help of interaction diagrams.

PROPOSED METHODOLOGY

This work is using the UML diagrams to calculate the size metrics. It has been found that existing researches focus on the USE CASE to be the UML diagram for evaluation of the size metric. Inclusion of the other UML diagrams in evaluation process of the size metric has been proposed in this research. The complete work is being carried in following steps:

1) Taken two case studies and their source code as the input of this work
2) UML diagrams of the case studies has been drawn and included for the evaluations of the size metric
3) Meta Mil software is being used to generate the XMI document for evaluation of the size metric
4) Generated XMI file is used with the SD Metric tool for evaluation of the metric values
5) For comparison purpose two other size metric techniques have been used i.e. Lines of Codes and Function Point Analysis
6) After evaluation of the metrics using various methods, a chart of the all the metric values will be generated to show the results.

The proposed work shall be carried out using the following structural diagram:

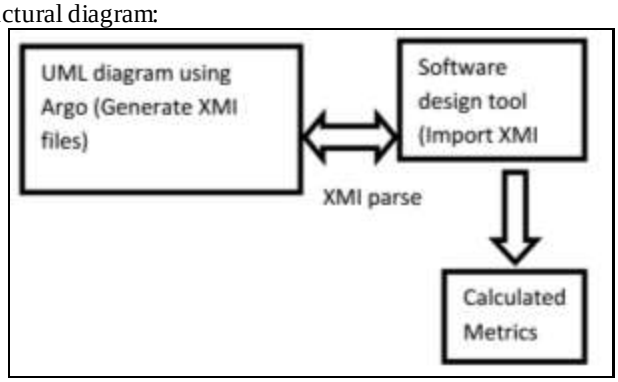

Figure: Structural diagram of the proposed work

Unified Modeling Language (UML) is popular today for capturing requirements and for describing the overall architecture of a software-intensive system. One of the UML constructs is a use case, which graphically depicts the way in which a user will interact with the system to perform one function or one class of functions. Three aspects of use cases can be helpful as inputs to a size estimate: the number of use cases, the number of actors involved in each use case, and the number of scenarios. An actor is a person or system that interacts with the system under consideration; typically, there is

one actor per use case, but sometimes there are more. A scenario is a potential outcome from using the software; the number of scenarios can range from one to thousands or millions, depending on the system and its complexity.

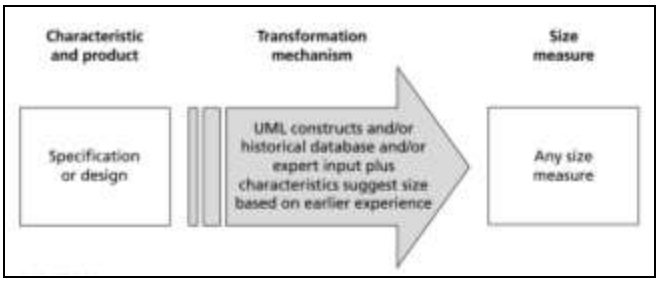

Figure: Characteristic Flow and Transformation Process Applied in UML Designing Tool

This technique can be useful when the size estimate is required after a UML specification is done. It can also be used as a cross-check of another method; if the answers from both methods are similar, the analysts may have more confidence in the result.

Metrics of SDMetric

Metric NumAttr: The number of attributes in the class. The metric counts all properties regardless of their type (data type, class or interface), visibility, changeability (read only or not), and owner scope (class-scope, i.e. static, or instance attribute). Not counted are inherited properties, and properties that are members of an association, i.e., that represent navigable association ends.

Metric NumOps: The number of operations in a class. Includes all operations in the class that are explicitly modelled (overriding operations, constructors, destructors), regardless of their visibility, owner scope (class-scope, i.e., static), or whether they are abstract or not. Inherited operations are not counted.

Metric NumPubOps: The number of public operations in a class. This is same as metric NumOps, but only counts operations with public visibility. It measures the size of the class in terms of its public interface.

Metric Setters: The number of operations with a name starting with 'set'. Note that this metric does not always yield accurate results. For example, an operation settle Account will be counted as setter method.

Metric Getters: The number of operations with a name starting with 'get', 'is', or 'has'. Note that this metric does not always yield accurate results. For example, an operation isolate Node will be counted as getter method.

Metric Nesting: The nesting level of the class (for inner classes). Measures how deeply a class is nested within other classes. Classes not defined in the context of another class have nesting level 0, their inner classes have nesting level 1, etc. Nesting levels deeper than 1 are unusual; an excessive nesting structure is difficult to understand, and should be revised.

Metric IFImpl: The number of interfaces the class implements. This only counts direct interface realization links from the class to the interface. For example, if a class C implements an interface I, which extends some other interfaces, only interface I will be counted, but not the interfaces that I extends (even though class c implements those interfaces, too).

Metric NOC: The number of children of the class (UML Generalization). Similar to export coupling, NOC indicates the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class. A large number of child classes may indicate improper abstraction of the parent class.

Metric NumDesc: The number of descendents of the class (UML Generalization). This counts the number of children of the class, their children, and so on.

Metric NumAnc: The number of ancestors of the class. This counts the number of parents of the class, their parents, and so on. If multiple inheritances are not used, the metric yields the same values as DIT.

Metric DIT: The depth of the class in the inheritance hierarchy. This is calculated as the longest path from the class to the root of the inheritance tree. The DIT for a class that has no parents is 0.Classes with high DIT inherits from many classes and thus more difficult to understand. Also, classes with high DIT may not be proper specializations of all of their ancestor classes.

Metric CLD: Class to leaf depth. This is the longest path from the class to a leaf node in the inheritance hierarchy below the class.

Metric OpsInh: The number of inherited operations. A large number of child classes may indicate ion of the parent class. The number of descendents of the class (UML Counts the number of children of the class, their number of ancestors of the class. parents of the class, their parents, and so on. If multiple inheritances are not used, the metric yields the same values as The depth of the class in the inheritance This is calculated as the longest path from the root of the inheritance tree. The DIT for a class that has no parents is 0.Classes with from many classes and thus is more difficult to understand. Also, classes with high DIT may not be proper specializations of Class to leaf depth. The longest path from the class to a leaf node in the inheritance hierarchy number of inherited operations. This is calculated as the sum of metric NumOps taken over all ancestor classes of the class.

Lines of Codes

This method attempts to assess the likely number of lines of code in the finished software product. Clearly, an actual count can be made only when the product is complete; lines of code are often considered to be inappropriate for size estimates early in the project life cycle. However, since many of the size-estimation methods express size in terms of lines of code, we can consider lines of code as a separate method in that it expresses the size of a system in a particular way.

Function Point Analysis

Function points were developed by Albrecht (1979) at IBM as a way to measure the amount of functionality in a system. They are derived from the requirements. Unlike lines of code, which capture the size of an actual product, function points do not relate to something physical but, rather, to something logical that can be assessed quantitatively.

IFPUG FPA: Formal method to measure size of business applications. It introduces complexity factor for size defined as function of input, output, query, external input file and internal logical file.

All Components are rated as Low, Average or High

After the components have been classified as one of the five major components (EI's, EO's, EQ's, ILF's or EIF's), a ranking of low, average or high is assigned. For transactions (EI's, EO's, EQ's) the ranking is based upon the number of files updated or referenced (FTR's) and the number of data element types (DET's). For both ILF's and EIF's files the ranking is based upon record element types (RET's) and data element types (DET's). A record element type is a user recognizable subgroup of data elements within an ILF or EIF. A data element type is a unique user recognizable, non recursive, field.

Each of the following tables assists in the ranking process (the numerical rating is in parentheses). For example, an EI that references or updates 2 File Types Referenced (FTR's) and has 7 data elements would be assigned a ranking of average and associated rating of 4. Where FTR's are the combined number of Internal Logical Files (ILF's) referenced or updated and External Interface Files referenced.

Table 3.1: EI Table

FTR's	DATA ELEMENTS		
	1-4	5-15	>15
0-1	LOW	Low	Average
2	LOW	Average	High
3 or More	Average	High	High

Table 3.2: Shared EO and EQ Table

FTR's	DATA ELEMENTS		
	1-5	6-19	>19
0-1	LOW	Low	Average
2-3	LOW	Average	High
> 3	Average	High	High

Table 3.3: Values for transactions

Rating	VALUES		
	EO	EQ	EI
Low	4	3	3
Average	5	4	4
High	7	6	6

Like all components, EQ's are rated and scored. Basically, an EQ is rated (Low, Average or High) like an EO, but assigned a value like and EI. The rating is based upon the total number of unique (combined unique input and out sides) data elements (DET's) and the file types referenced (FTR's) (combined unique input and output sides). If the same FTR is used on the input and output side, then it is counted only one time. If the same DET is used on the input and output side, then it is only counted one time.

For both ILF's and EIF's the number of record element types and the number of data elements types are used to determine a ranking of low, average or high. A Record Element Type is a user recognizable subgroup of data elements within an ILF or EIF. A Data Element Type (DET) is a unique user recognizable, non recursive field on an ILF or EIF.

Table 3.4: Table used to evaluate Rating of EI, EO, EQ

RET's	DATA ELEMENTS		
	1-19	20-50	> 50
1	Low	Low	Average
2-5	Low	Average	High
> 5	Average	High	High

Table 3.5: Values for transactions for ILF & EIF

Rating	VALUES	
	ILF	EIF
Low	4	3
Average	5	4
High	7	6

The counts for each level of complexity for each type of component can be entered into a table such as the following one. Each count is multiplied by the numerical rating shown to determine the rated value. The rated values on each row are summed across the table, giving a total value for each type of component. These totals are then summed across the table, giving a total value for each type of component. These totals are then summed down to arrive at the Total Number of Unadjusted Function Points.

The value adjustment factor (VAF) is based on 14 general system characteristics (GSC's) that rate the general functionality of the application being counted. Each characteristic has associated descriptions that help determine the degrees of influence of the characteristics. The degrees of influence range on a scale of zero to five, from no influence to strong influence. The IFPUG Counting Practices Manual provides detailed evaluation criteria for each of the GSC'S, the table below is intended to provide an overview of each GSC. Rate each factor (Fi, i=1 to14) on a scale of 0 to 5:

Table 3.6: General System Characteristics

F1. Does the system require reliable backup and recovery?	
F2. Are data communications required?	
F3. Are there distributed processing functions?	
F5. Will the system run in an existing, heavily utilized operational environment?	
F6. Does the system require on-line data entry?	
F7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?	
F8. Are the master files updated on-line?	
F9. Are the inputs, outputs, files or inquiries complex?	
F10. Is the internal processing complex?	
F11. Is the code designed to be reusable?	
F12. Are conversion and installation included in the design?	
F13. Is the system designed for multiple installations in different organizations?	
F14. Is the application designed to facilitate change and ease of use by the user?	

Once all the 14 GSC's have been answered, they should be tabulated using the IFPUG Value Adjustment Equation (VAF) -- 14

VAF = 0.65 + [(Ci) / 100] .i = is from 1 to 14 representing each GSC.

where: Ci = degree of influence for each General System Characteristic

The final Function Point Count is obtained by multiplying the VAF times the Unadjusted Function Point (UAF).

FP = UAF * VAF

Summary of benefits of Function Point Analysis

Function Points can be used to size software applications accurately. Sizing is an important component in determining productivity (outputs/inputs).

They can be counted by different people, at different times, to obtain the same measure within a reasonable margin of error.

Function Points are easily understood by the non technical user. This helps communicate sizing information to a user or customer.

Function Points can be used to determine whether a tool, a language, an environment, is more productive when compared with others.

3.1 COCOMO-II

The COCOMO II model makes its estimates of required effort (measured in Person-Months – PM) based primarily on your estimate of the software project's size (as measured in thousands of SLOC, KSLOC):

Effort = 2.94 * EAF * $(KSLOC)^E$ ... (3)

Where EAF Is the Effort Adjustment Factor derived from the Cost Drivers. E Is an exponent derived from the five Scale Drivers. As an example, a project with all Nominal Cost Drivers and Scale Drivers would have an EAF of 1.00 and exponent, E, of 1.0997. Assuming that the project is projected to consist of 8,000 source lines of code, COCOMO II estimates that 28.9 PersonMonths of effort is required to complete it: Effort = 2.94 * (1.0) * $(8)^{1.0997}$ = 28.9 Person-Months

MAINTAINABILITY

In engineering, maintainability is the ease with which a product can be maintained in order to:

- isolate defects or their cause,
- correct defects or their cause,
- repair or replace faulty or worn-out components without having to replace still working parts,
- prevent unexpected breakdowns,
- maximize a product's useful life,
- maximize efficiency, reliability, and safety,
- meet new requirements,
- make future maintenance easier, or

- Cope with a changed environment.

In some cases, maintainability involves a system of continuous improvement - learning from the past in order to improve the ability to maintain systems, or improve reliability of systems based on maintenance experience.

Software maintenance costs result from modifying your application to either support new use cases or update existing ones, along with the continual bug fixing after deployment. As much as 70-80% of the Total Ownership Cost (TCO) of the software can be attributed to maintenance costs alone!

Software maintenance activities can be classified as:

- Corrective maintenance – costs due to modifying software to correct issues discovered after initial deployment (generally 20% of software maintenance costs)
- Adaptive maintenance – costs due to modifying a software solution to allow it to remain effective in a changing business environment (25% of software maintenance costs)
- Perfective maintenance – costs due to improving or enhancing a software solution to improve overall performance (generally 5% of software maintenance costs)
- Enhancements – costs due to continuing innovations (generally 50% or more of software maintenance costs)
- Since maintenance costs eclipse other software engineering activities by large amount, it is imperative to answer the following question:

Measuring software maintainability is non-trivial as there is no single metric to state if one application is more maintainable than the other and there is no single tool that can analyze your code repository and provide you with an accurate answer either. There is no substitute for a human reviewer, but even humans can't analyze the entire code repositories to give a definitive answer. Some amount of automation is necessary.

So, how can you measure the maintainability of your application? To answer this question let's dissect the definition of maintainability further. Imagine you have access to the source code of two applications – A and B. Let's say you also have the super human ability to compare both of them in a small span of time. Can you tell, albeit subjectively, whether you think one is more maintainable than the other? What does the adjective maintainable imply for you when making this comparison – think about this for a second before we move on.

Done? So, how did you define maintainability? Most software engineers would think of some combination of testability, understand ability and modifiability of code, as measures of maintainability. Another aspect that is equally critical is the ability to understand the requirement, the "what" that is implemented by the code, the "how". That is, is there a mapping from code to requirements and vice versa that could be discerned from the code base itself? This information may exist externally as a traceability document, but even having some information in the source code – either by the way it's laid out into packages/modules, naming conventions or having READMEs in every package explaining the role of the classes, can be immensely valuable.

These core facets can be broken down further, to gain further insight into the maintainability of the application:

1) Testability – the presence of an effective test harness; how much of the application is being tested, the types of tests (unit, integration, scenario etc.,) and the quality of the test cases themselves?
2) Understandability – the readability of the code; are naming conventions followed? Is it self-descriptive and/or well commented? Are things (e.g., classes) doing only one thing or many things at once? Are the methods really long or short and can their intent be understood in a single pass of reading or does it take a good deal of screen staring and whiteboard analysis?
3) Modifiability – structural and design simplicity; how easy is it to change things? Are things tightly or loosely coupled (i.e., separation of concerns)? Are all elements in a package/module cohesive and their responsibilities clear and closely related? Does it have overly deep inheritance hierarchies or does it favor composition over inheritance? How many independent paths of execution are there in the method definitions (i.e., cycolmatic complexity)? How much code duplication exists?
4) Requirement to implementation mapping and vice versa – how easy is it to say "what" the application issupposed to do and correlate it with "how" it is being done, in code? How well is it done? Does it need to be refactored and/or optimized? This information is paramount for maintenance efforts and it may or may not exist for the application under consideration, forcing you to reverse engineer the code and figure out the 'what' yourself.

Those are the four major dimensions on which one can measure maintainability. Each of the facets can (and is) broken down further for a more granular comparison. These may or may not be the exact same ones that you thought of, but there will be a great deal of overlap. Also, not every criterion is equally important. For some teams, testability may trump structural/design simplicitly. That is, they may care a lot more about the presence of test cases (depth and breadth) than deep inheritance trees or a slightly more tightly coupled design. It is thus vital to know which dimension of maintainability is more important for your maintenance team when measuring the quality of your application and carry out the reviews and refactoring with those in mind.

The table below, towards the end of the article, shows a detailed breakdown of the above dimensions of maintainability and

elaborates on their relevance to measuring the quality of the source code [2]:

1) Correlation with quality: How much does the metric relate with our notion of software quality? It implies that nearly all programs with a similar value of the metric will possess a similar level of quality. This is a subjective correlational measure, based on our experience.
2) Importance: How important is the metric and are low or high values preferable when measuring them? The scales, in descending order of priority are: Extremely Important, Important and Good to have
3) Feasibility of automated evaluation: Are things fully or partially automable and what kinds of metrics are obtainable?
4) Ease of automated evaluation: In case of automation how easy is it to compute the metric? Does it involve mammoth effort to set up or can it be plug-and-play or does it needs to be developed from scratch? Any OTS tools readily available?
5) Completeness of automated evaluation: Does the automation completely capture the metric value or is it inconclusive, requiring manual intervention? Do we need to verify things manually or can we directly rely on the metric reported by the tool?
6) Units: What units/measures are we using to quantify the metric?

DECISION TREE

Decision Trees are excellent tools for helping you to choose between several courses of action.

They provide a highly effective structure within which you can lay out options and investigate the possible outcomes of choosing those options. They also help you to form a balanced picture of the risks and rewards associated with each possible course of action.

Drawing a Decision Tree

You start a Decision Tree with a decision that you need to make. Draw a small square to represent this towards the left of a large piece of paper.

From this box draw out lines towards the right for each possible solution, and write that solution along the line. Keep the lines apart as far as possible so that you can expand your thoughts.

At the end of each line, consider the results. If the result of taking that decision is uncertain, draw a small circle. If the result is another decision that you need to make, draw another square. Squares represent decisions, and circles represent uncertain outcomes. Write the decision or factor above the square or circle. If you have completed the solution at the end of the line, just leave it blank.

Starting from the new decision squares on your diagram, draw out lines representing the options that you could select. From the circles draw lines representing possible outcomes. Again make a brief note on the line saying what it means. Keep on doing this until you have drawn out as many of the possible outcomes and decisions as you can see leading on from the original decisions.

Once you have done this, review your tree diagram. Challenge each square and circle to see if there are any solutions or outcomes you have not considered. If there are, draw them in. If necessary, redraft your tree if parts of it are too congested or untidy. You should now have a good understanding of the range of possible outcomes of your decisions.

RESULT AND DISCUSSION

Results of the Proposed UML Diagram Based Metric Calculation & Count of Operations in Actual Software

CASE STUDY	UML DESIGN METRIC NUMOPSCLS VALUE	ACTUAL SOFTWARE OPERATIONS COUNT
DSS	1	1
VCR	12	12

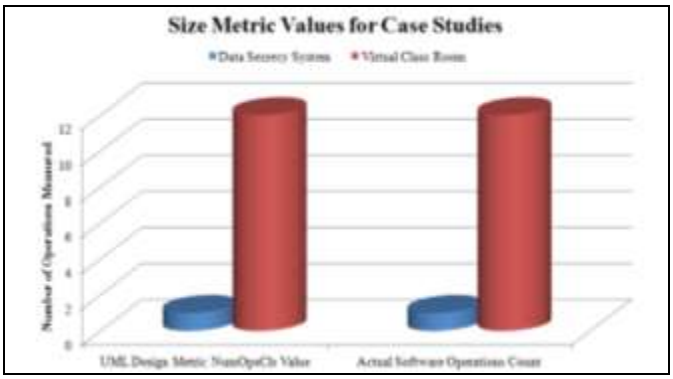

Figure: Graph showing comparison of the number of operations evaluated using two different methods

Table: Average Permissible Error obtained from the Proposed Algorithms and Other Techniques

Algorithm	Average Permissible Error
LOC	27.5
FPA	7.5
UML Tools	3.5

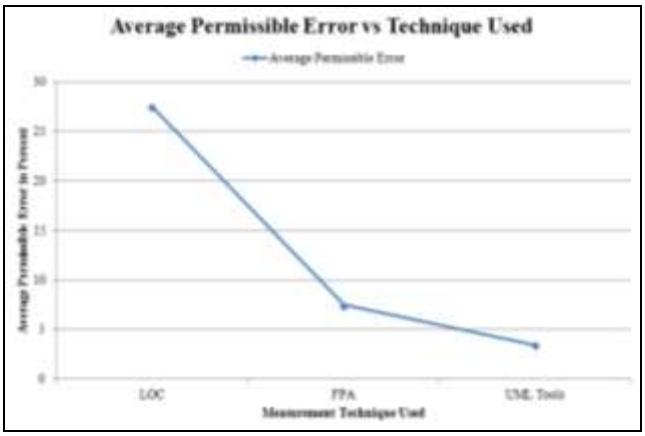

Figure : Graph showing Average Permissible Error in Percent for the different techniques

CONCLUSION

This work has been done to evaluate the effect of different UML diagrams to evaluate the size metric for the software projects. Size metric is a valuable measurement in defining the cost of the software. In this work different UML diagrams such as collaboration diagram, state flow chart, activity diagram, use case, component diagram are used together to evaluate the software size metric. For confirmation and proof two other techniques of lines of codes (LOC) and function point analysis (FPA) have been applied to measure the software size metrics. From the results obtained from the output of SD Metric Tool, LOC and FPA, it is found that the results obtained from the inclusion of the different UML diagrams and most accurate and matches with the actual software source code.

REFERENCES

[1]. Pressman S. Roger "Software Engineering" Sixth Edition, McGraw Hill International 2005, pg649, chap 22, ISBN : 007-124083-7.

[2]. Rumbaugh James, Jacobson, Ivar and Booch Grady, "The Unified Modeling Language User Guide" Second Edition2008, pg 5, chap1, ISBN: 978-81-317-1582-6.

[3]. http://www.uml.org

[4]. Jacobson Magnus Christerson , Patrick Jonsson ,Gunnar Overgaard" Object-oriented Software Engineering" 2008, pg 66, chap3, Isbn: 81-317-0408-4.

[5]. Yi Tong et. al, "A Comparison of Metrics for UML Class Diagrams" ACM SIGSOFT Software Engineering Notes Page 1, September 2004, Volume 29 .

[6]. Li Wei et .al, "An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes" IEEE Transactions On Software Engineering , November 2003 ,Volume 29 NO. 11, 1043.

[7]. Mitchell Aine et. al , "Toward a definition of run-time object-oriented metrics" 7TH ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering , 2003.

[8]. Xenos M. et al., "Object-oriented metrics – a survey" Proceedings of the FESMA 2000, Federation of European Software Measurement Associations, Madrid, Spain, 2000.

[9]. arasimhan Lakshmi.V et.al, " Evaluation of a Suite of Metrics for Component Based Software Engineering (CBSE)" Issues in Informing Science and Information Technology Volume 6, 2009.

[10]. Shaik Amjan et.al , " Metrics for Object Oriented Design Software Systems: A Survey" Journal of Emerging Trends in Engineering and Applied Sciences (JETEAS) 1 (2): 190-198 c, 2010.Jahan Vafaei et.al ," A New Method Software Size Estimation based on UML Metrics".

[11]. Chen Yue , Boehm Barry et.al ,"An Empirical Study of eServices Product UML Sizing Metrics.

[12]. Linda Edith P et. al ," Metrics for Component based Measurement Tools", International Journal of Science & Engineering ,Research Volume 2,Issue 5,May -2011

[13]. Subramanyam Ramanath et al, "Empirical Analysis of CK Metrics for Object-oriented Design Complexity: Implications for Software Defects", IEEE Transactions on Software Engineering , Vol. 29. NO.4 , April 2003.

[14]. Tegarden P. David et al., "Effectiveness of Traditional Software Metrics for Object-Oriented Systems"

[15]. Doban Orysolya et. al ,"Cost Estimation Driven Software Development Process"

[16]. Lavazza Luigi et al .,"Using Function Point in the Estimation of Real-Time Software: an Experience", Proceedings 5th Software Measurement European Forum, Milan 2008.

[17]. Chidamber et al., "Managerial use of metrics for Object-oriented software: an exploratory analysis", IEEE

