Volume 5, Issue 4, April 2015 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Object Oriented Software Testability (OOSTe) Metrics Assessment Framework

[1]Harsha Singhani*, [2]Dr. Pushpa R. Suri

[1]Student,

[1, 2] Department of Computer Science and Applications, Kurukshetra University, Kurukshetra, Haryana, India

Abstract— Testability is an important and essential software quality factor which is not easy to measure. Lot of object oriented metrics are found to be relevant in doing so. These metrics have been applied for object oriented software testability assessment mostly during software in software design phase. In this paper a framework using static and dynamic metric along with set of various popular metrics option under each category is proposed to calculate the object oriented software testability. So that one can go beyond the design time testability evaluation and instead do assessment on overall system at various development phases. This research paper provides the theoretical framework based on related work done on software testability measurement using object oriented metrics suite.

Keywords— Object Oriented Software Testability, Testability Framework, Static Testability, Dynamic Testability

I. INTRODUCTION

Software metrics measure different aspects of various software factors and therefore play an important role in analysing and improving software quality. Several studies have indicated that metrics provides useful information on external quality aspects of software such as its maintainability, reusability, testability and reliability, and provide a means of estimating the effort needed for testing. The categorization of these metrics can be done in several ways. One such way would be whether the metrics is applicable at design time with static code known as Static metrics or during application runtime known as Dynamic metrics. However, the ability of such static metrics to accurately predict the dynamic behaviour of an application may not be sufficient. As, static metrics alone may be insufficient in evaluating the dynamic behaviour of an application at runtime, as its behaviour will be influenced by the operational environment as well as the complexity of the source code.

The metrics investigated related to object oriented software testability assessment mostly belong to static software metrics category. These metrics mostly adapted from CK, MOOD, Brian, Henderson-Sellers metric suite [1]–[4]. The popularity amongst the researchers for dynamic metrics has also been equally there for runtime testability measurement [5]–[8] but not much has been adapted by the industrial practitioners. A thorough literature survey on the same revealed the preferences and practices amongst the researchers, industry practitioners in the direction software testability assessment and analysis. We would like to bring forward this work along with a proposed framework for object oriented software testability assessment which is based testability model for the same.

This paper is organized as follows: Section 2 gives brief overview of software testability. Section 3 gives the overview of relative object oriented metrics work. Section 4 provides review of research work related to static and dynamic metrics study for object oriented software testability done till date. Section 5 presents the details of proposed framework for object oriented software testability followed by conclusion drawn in section 6.

II. SOFTWARE TESTABILITY

Software Testability as defined by IEEE standards [9] as: "(1) Degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and the performance of tests to determine whether those criteria have been met." .Thus, Testability actually acts as a software support characteristic for making it easier to test. As stated by Binder and Freedman a Testable Software is one that can be tested easily, systematically and externally at the user interface level without any ad-hoc measure [10][11]. Whereas [12] describe it as complimentary support to software testing by easing down the method of finding faults within the system by focussing more on areas that most likely to deliver these faults. The insight provided by testability at designing, coding and testing phase is very useful as this additional information helps in product quality and reliability improvisation [13][14]. All this has lead to a notion amongst practitioners that testability should be planned early in the design phase though not necessarily so. As seen by experts like Binder it involves factors like controllability and observability i.e. ability to control software input and state along with possibility to observe the output and state changes that occur in software. So, overall testable software has to be controllable and observable[10].But over the years more such quality factors like understandability, traceability, complexity and test–support capability have contributed to testability of a system[15].All these factors make testability a core quality factor.

© 2015, IJARCSSE All Rights Reserved

Hence, over the years Testability has been diagnosed as one of the core quality indicators, which leads to improvisation of test process. Several approaches as Program Based , Model Based and Dependability Testability assessment for Testability estimation have been proposed [16]. The studies mostly revolve around the measurement methods or factors affecting testability. We would take this study further keeping focus on mainly object oriented system. As object oriented technology has become most widely accepted concept by software industry nowadays. But testability still is a taboo concept not used much amongst industry mainly due to lack of standardization, which may not be imposed for mandatory usage but just been looked upon for test support[17].

III. RELATIVE OBJECT ORIENETD METRICS OVERVIEW

Over the years a lot of OO design and coding metrics have been adopted or discussed by research practitioners for studying to be relevantly adopted in quantification of software testability. Most of these metrics are proposed by Chidamber and Kemerer [1], which is found to be easily understandable and applicable set of metrics suite. But along with that there are other metrics suites also available such as MOOD metrics suite[2].These metrics can be categorized as one of the following object oriented characteristic metrics- Size, Encapsulation, Polymorphism, Coupling, Cohesion, Inheritance and Complexity. Few of them were applied on static code or design hence often categorized Static Metrics and few others are applicable at software runtime categorized as Dynamic metrics. Now from testability perspective, which is the main motive of our study, we have studied and listed here the important of these popular object oriented metrics used till date.

I. CK Metrics Suite [1],[18]

CK Metrics suite contains six metrics, which are indicative of object oriented design principle usage and implementation in software.

i. ***Number of Children (NOC):*** It is a basic size metrics which calculates the no of immediate descendants of the class. It is an inheritance metrics, indicative of level of reuse in an application. High NOC represents a class with more children and hence more responsibilities.

ii. ***Weighted Method per class (WMC):*** WMC is a complexity metrics used for class complexity calculation. Any complexity measurement method can be used for WMC calculation most popular amongst all is cyclomatic complexity method[19]. WMC values are indicators of required effort to maintain a particular class. Lesser the WMC value better will be the class.

iii. ***Depth of Inheritance Tree (DIT):*** DIT is an inheritance metrics whose measurement finds the level of inheritance of a class in system design. It is the length of maximum path from the node to the root of the hierarchy tree. It is a helps in understanding behaviour of class, measuring complexity of design and potential reuse also.

iv. ***Coupling between Objects (CBO):*** This is a coupling metrics which gives count of no of other classes coupled to a class, which method of one class using method or attribute of other class. The high CBO indicates more coupling and hence less reusability.

v. **Lack of Cohesion Metrics (LCOM)**: It is a cohesion metrics which measures count of methods pairs with zero similarity minus method pairs with non zero similarity. Higher cohesion values lead too complex class bringing cohesion down. So, practitioners keep cohesion high by keeping LCOM low. LCOM was later reformed as LCOM* by [4] and used in few researches.

vi. **Response for a class (RFC):** RFC is the count of methods implemented within a class. Higher RFC value indicates more complex design and less understandability. Whereas, lower RFC is a sign of greater polymorphism. Hence, it is generally categorized as complexity metrics.

II. HS Metric Suite[4]

i. **Line of Code (LOC) or Line of Code per Class (LOCC):** It is a size metrics which gives total no of lines of code (non comment & non blank) in a class.

i. **Number of Classes (NC / NOC):** The total number of classes.

ii. **Number of Attributes (NA / NOA)**: The total number of attributes.

ii. **Number of Methods (NM / NOM)**: The total number of methods

iii. **Data Abstraction Coupling (DAC):** The DAC measures the coupling complexity caused by Abstract Data Types (ADTs)

iv. **Message Passing Coupling (MPC):** number of send statements defined in a class

v. **Number of Overriden Methods (NMO):** defined as number of methods overridden by a subclass

III. MOOD Metrics Suite [2][18]

Metrics for object oriented design (MOOD) The basic metrics consists of encapsulation (MHF, AHF), inheritance (MIF, AIF), polymorphism (POF) and coupling metrics (COF). This model based on two major features of object oriented classes i.e. methods and attributes. Each feature is either hidden or visible from a given class. Each metrics thus calculates values between lowest (0%)-highest (100%) indicating the absence or presence of a particular feature. Where

i. **Method Hiding Factor (MHF):** This metric is computed by dividing the methods hidden to the total methods defined in the class. By this an estimated encapsulation value is generated. High value indicates more private attribute and low value indicates more public attributes.

© 2015, IJARCSSE All Rights Reserved

ii. **Attribute Hidden Factor (AHF):** It shows the attributes hidden to the total attributes defined in the class. By this an estimated encapsulation value is generated.
iii. **Method Inheritance Factor (MIF):** This metrics is the sum of all inherited methods in a class. Low value indicates no inheritance.
iv. **Attribute Inheritance Factor (AIF):** This is ratio of sum of all inherited attributes in all classes of the system. Low value indicates no inherited attribute in the class.
v. **Polymorphism Factor (POF)/(PF):** This factor represents the actual number of possible polymorphic states. Higher value indicates that all methods are overridden in all derived classes.
vi. **Coupling Factor (COF):** The coupling here is same as CBO. It is measured as ratio of maximum possible couplings in the system to actual number of coupling. Higher value indicates rise in system complexity as it means all classes are coupled with each other thus increasing hence reducing system understandability and maintainability along with less reusability scope.

IV. Genero's UML Class Diagram Metrics Suite [20]
iii. **Number of Associations (NAssoc):** The total number of associations
iv. **Number of Aggregation (NAgg) :** The total number of aggregation relationships within a class diagram (each whole-part pair in an aggregation relationship)
v. **Number of Dependencies (NDep):** The total number of dependency relationships
vi. **Number of Generalisations (NGen):** The total number of generalisation relationships within a class diagram (each parent-child pair in a generalisation relationship)
vii. **Number of Aggregations Hierarchies (NAggH):** The total number of aggregation hierarchies in a class diagram.
viii. **Number of Generalisations Hierarchies (NGenH):** The total number of generalisation hierarchies in a class diagram
ix. **Maximum DIT:** It is the maximum between the DIT value obtained for each class of the class diagram. The DIT value for a class within a generalisation hierarchy is the longest path from the class to the root of the hierarchy.
x. **Maximum HAgg:** It is the maximum between the HAgg value obtained for each class of the class diagram. The HAgg value for a class within an aggregation hierarchy is the longest path from the class to the leaves.
xi. **Coupling Between Classes (CBC):** it is same as CBO.

V. MTMOOD Metrics [21]**:**
i. **Enumeration Metrics (ENM)**: it is the count of all the methods defined in a class.
ii. **Inheritance Metrics (INM) / Reuse Metrics (REM):** it is the count of the number of class hierarchies in the design.
iii. **Coupling Metrics (CPM):** it is the count of the different number of classes that a class is directly related to.
iv. **Cohesion Metrics (COM):** This metric computes the relatedness among methods of a class based upon the parameter list of the methods [computed as LCOM, 1993 Li and Henry version]

VI. Other Important OO Metrics:
Apart from above mentioned metrics there are few other significant structural as well as object oriented metrics which have been significantly used in testability research:
i. **No of Object (NOO)** [4]: which gives the number of operations in a class
ii. **McCabe Complexity Metrics**[19] / **Cyclomatic Complexity (CC):** It is equal to the number of decision statements plus one. It predicts the scope of the branch coverage testing strategy. CC gives the recommended number of tests needed to test every decision point in a program.
iii. **Fan-out (FOUT)**[22]: FOUT of any method A is the number of local flows from method A plus the number of data structures which A updates. In other words FOUT estimates the number of methods to be stubbed, to carry out a unit testing of method A.

VII. Test Class Metrics:
These test class metrics used for the study actually correlate the various testability affecting factors identified through above metrics with testing effort required at unit testing or integration testing level in object oriented software's. Few of these metrics are TLOC/TLOCC(Test class line of code), TM(no of test methods), TA/TAssert(no of asserts/test cases per class), NTClass(no of test classes), TNOO(test class operation count), TRFC(test class RFC count), TWMC(test class complexity sum)[23], [24]. The metrics are calculated with respect to the unit test class generated for the specific module. These metrics are analytically correlated with specific metrics suite for analysing testing effort required at various testing level by many researchers.

IV. STATIC AND DYNAMIC TESTABILITY METRICS SURVEY

Before we categorize static and dynamic testability metrics we should understand why testability measurement is important. *Software testability measurement* refers to the activities and methods that study, analyze, and measure software testability during a software product life cycle. Unlike software testing, the major objective of software

© 2015, IJARCSSE All Rights Reserved

testability measurement is to find out which software components are poor in quality, and where faults can hide from software testing. Now these measurements can be applied at various phases during software development life cycle of a system. In the past, there were a number of research efforts addressing software testability measurement. The focus of past studies was on how to measure software testability at the at various phase like Design Phase[10][25]–[27][14], [28] and Coding Phase[29][30][31][23]. Quite recently there has been some focus on Testing & Debugging Phase also[7][32]. These metrics are closely related to the Software quality factors i.e. Controllability, Observability, Built in Test Capability, Understandability and Complexity, all these factors are independent to each other.

Now, all these measurement methods specifically from object oriented software systems perspectives involving various static and dynamic metrics are discussed below in brief in coming sections. Here we provide the survey along with giving a framework model for testability implementation during object oriented software development life cycle using testability metrics support.

A. Static Metrics Survey

Static Metrics belong to static code or design of software. These metrics generally revolves around basic object oriented features to be analysed for the system under study. These features mainly evolve around four basic core aspects Encapsulation, Inheritance, Coupling and Cohesion within object entities at source code or code design level. Although software testability is most obviously relevant during testing, but paying attention to testability early in the development process can potentially enhance testing along with significantly improving testing phase effectiveness. This is done for early stage software design improvisation which has highly beneficial impact on the final testing cost and its efficiency. All above mentioned core features are relevant from testability perspective as discussed below in the work of various researchers.

Binder,1994 [10] was amongst few of the early researchers who proposed design by testability concept [10] which revolved around a basic fishbone model for testability with six main affecting factors though not exactly giving any clear metrics for software design constructs, as all these factors namely Representation , Implementation , Built In Test, Test Suite, Test Tool & Test process are related to higher level abstraction. But his work highlighted some of the key features such as controllability, observability, traceability, complexity, built in test and understandability which were later used & identified as critical assessment attributes of testability.

He identified various structural metrics for testability assessment from CK metric suite[1], Henederson-Sellers metrics[4] and McCabe complexity metrics[19], which may be relatively useful for testability assessment from encapsulation, inheritance, polymorphism, and complexity point of view to indicate complexity, scope of testing or both under all above mentioned features. The effect of all complexity metrics indicated the same: a relatively high value of the metric indicates decreased testability and relatively low value indicates increased testability. Scope metrics indicated the quantity of tests: the number of tests is proportional to the value of the metric. Later lot of work has been done focussed around Binders theory and lot of other new found factors for testability measurement.

McGregor & Srinivas, 1996 [33] study elaborated a Testability calculation technique using visibility component metrics. The proposed method used to estimate the effort that is needed to test a class, as early as possible in the development process by assessing the testability of a method in a class. Testability of a method into the class depends upon the visibility component (VC) as elaborated below:

- Testability of method is **Tm=k *(VC),** Where visibility component **(VC** = **Possible Output / Possible Input)** and
- Testability of the class is **Tc=min(Tm)**

The visibility component (VC) has been designed to be sensitive to object oriented features such as inheritance, encapsulation, collaboration and exceptions. Due to its role in early phases of a development process the VC calculations require an accurate and complete specification of documents.

Bruntink 2003[24], [34] used various metrics based on source code factors for testability analysis using dependency of test case creation and execution on these factors. The number of test cases to be created and executed is determined by source code factors as well as the testing criterion. In many cases, the testing criterion determines which source code factors actually influence the number of required test cases. The testability was not directly quantified tough, but the results were influential in other research studies.

- The nine popular design metrics DIT, FOUT, LCOM, LOCC, NOC, NOF, NOM, RFC, and WMC from CK metrics suite [1] were identified and considered for analysing their impact on test case generation.
- dLOCC, dNOTC were the two proposed test suite metrics for analysing the effect of above metrics in test case construction.

The research resulted in finding the correlation between source code metrics themselves like LOCC & NOM and DIT & NOC. Also there is a significant correlation between class level metrics (most notably FOUT, LOCC, and RFC) and test level metrics (dLOCC and dNOTC). Though there was no quantification of testability as such but based on Binders theory of testability and factors which were studied further in this paper. Hence the study on source code factors: factors that influence the number of test cases required to test the system, and factors that influence the effort required to develop each individual test case, helped giving testability vision, which further need refinement.

© 2015, IJARCSSE All Rights Reserved

Baudry et. al. 2004-05[35], [36] work was based on testability analysis on OO designs, focusing in UML class diagrams as a reference model. They proposed two configurations in a UML class diagram that can lead to code difficult to test. These configurations are called testability anti-patterns, and can be of two types, either class interaction system and due to inheritance and dynamic binding, the control or self-usage interaction. A second contribution of their work was proposal of UML stereotypes to add information on relationships in the class diagram that can help improve the testability of the design. They used class diagrams for the study and hence, the complexity estimation may go high during implementation, for that they proposed refactoring and UML stereo type refinement method for designs that are closer to implementation code. The study was not conducted on large scale industrial data.

Khan & Mustafa,2009 [21] proposed a design level testability metrics name MTMOOD, which was calculated on the basis of key object oriented features such as encapsulation, Inheritance, coupling and cohesion. The models ability to estimate overall testability from design information has been demonstrated using six functionally equivalent projects where the overall testability estimate computed by model had statistically significant correlation with the assessment of overall project characteristics determined by independent evaluators. The proposed testability metrics details are as follows:

- ***Testability= -0.08 * Encapsulation + 1.12 * Inheritance + 0.97 * Coupling***

The three standard metrics used for incorporating above object oriented features mentioned in the equation were ENM, REM & CPM respectively as explained in section 2. The proposed model for the assessment of testability has been validated by author using structural and functional information from object oriented software. Though the metrics is easy but is very abstract, it does not cover major testability affecting features of any object oriented software in consideration such as cohesion , polymorphism etc.

Singh & Saha (2010) [37] performed empirical study was to establish relation between various source code metrics from past [1][4] and test metrics proposed by [24] and others. The study was conducted on large Java system Eclipse. The study showed a strong correlation amongst four test metrics and all the source code metrics (explained briefly in section 2), which are listed below:

- Five Size Metrics: LOC, NOA, NOM, WMC and NSClass.
- Three Cohesion Metrics: LCOM, ICH and TCC
- Three Coupling Metrics: CBO, DAC, MPC, & RFC
- Two Inheritance Metrics: DIT & NOC.
- One Polymorphism Metrics: NMO
- Four Test Metrics : TLOC, TM, TA & NTClass

The study showed that all the observed source code metrics are highly correlated amongst themselves. Second observation was that, the test metrics are also correlated. The size metrics are highly correlated to testing metrics. Increase in Software Size, Cohesion, Coupling, Inheritance and Polymorphism metrics values decreases testability due to increase in testing effort.

M. Badri et. al.,2011 [23]study was based on adapted model MTMOOD proposed by [21], at source code level named as MTMOOP. They adapted this model to the code level by using the following source code metrics: NOO [4], DIT and CBO [1]. Using these three source code metrics they proposed a new testability estimation model. The model was empirically verified against various test class metrics of commercial java systems. The proposed testability metrics was:

- **Testability = -0.08*NOO + 1.12*DIT + 0.97*CBO**
- Five Test Class Metrics Used: TLOC, TAssert, TNOO, TRFC, TWMPC

The basic purpose was to establish the relationship between the MTMOOP model and testability of classes (measured characteristics of corresponding test classes).The result showed positive correlation between the two.

Khalid et. al. ,2011 [38] proposed five metrics model based on CK metrics suite[1] and MTMOOD[21] for measuring complexity & testability in OO designs based on significant design properties of these systems such as encapsulation, inheritance and polymorphism along with coupling & cohesion. These metrics are: AHF, MHF, DIT, NOC, and CBC, as explained in section 3. With findings that High AHF and MHF values implies less complexity and high testability value making system easy to test. On the other hand DIT, NOC, and CBC are directly proportional to complexity as higher values of any of these will increase system complexity making it less testable and hence making system more non test friendly.

Badri et. al.,2012 [39], [40] study was basically to identify the relationship between major object oriented metrics and unit testing along with that they also studied the impact of various lack of cohesion metrics on testability at source code level from unit testing point of view using existing commercial java software's with junit test class. The cohesion metrics and other object oriented metrics used for the study were explained in section 3 already are listed below:

- Three Cohesion metrics: LCOM, LCOM* and LCD
- Seven object oriented metrics: CBO, DIT, NOC, RFC, WMC, LCOM, LOC
- Two Test class metrics used: TAssert, TLOC

© 2015, IJARCSSE All Rights Reserved

The study performed at two stages actually showed significant correlation between the observed object oriented metrics and test class metrics.

Nazir Khan,2013[41]–[43] did their research from object oriented design perspective. The model proposed was on the basis of two major quality factors affecting testability of object oriented classes at design level named- understandability and complexity. The measurement of these two factors was established with basic object oriented features in other research [41], [42] The metrics used for the assessment of these two factors were based on Genero metrics suite [20] as well as some basic coupling, cohesion and inheritance metrics:

- Understandability = 1.33515 + 0.12*NAssoc + 0.0463*NA + 0.3405*MaxDIT
- Complexity = 90.8488 + 10.5849*Coupling - 102.7527*Cohesion + 128.0856*Inheritance
- **Testability = - 483.65 + 300.92*Understandability - 0.86*Complexity**

Where the coupling, cohesion and Inheritance was measured using CPM, COM, INM metrics as explained in section 3. The Testability metrics was validated with very small scale C++ project data. Thus the empirical study with industrial data needs to be performed yet. Though the model found important from object oriented design perspective but lacked integrity in terms of complete elaboration of their study considering the frame work provided [44] by them. Also, not much elaborative study was conducted on complexity and understandability correlation establishment with basic object oriented features.

B. Dynamic Metrics Survey

The dynamic aspects of design quality are less frequently discussed in OO quality metrics literature as compared to static metrics. The complex dynamic behavior of many real-time applications motivates a shift in interest from traditional static metrics to dynamic metrics. Generally the metrics studied at source code level at runtime is not for code improvisation but rather to help systems identify hidden faults. So, basically here the metrics is not for finding alternatives to a predefined system but for establishing relation between source code factors affecting testability in terms of test case generation factors, test case affecting factors etc. as noticed by many researchers whose brief overview of work is discussed below:

Voas & Miller 1992 [12], [13], [45] concentrated their study of testability in the context of conventional structured design. The technique is also known as PIE technique. PIE measurement helps computing the sensitivity of individual locations in a program, which refers to the minimum likelihood that a fault at that location will produce incorrect output, under a specified input distribution. The concept here is of execution, infection and propagation of fault within the code and it outputs.

- ***Testability of a software statement T(s) = Re(s) ∗ Ri(s) ∗ Rp(s)***

Where, Re(s) is the probability of the statement execution, Ri(s) the probability of internal state infection and Rp(s) the probability of error propagation. PIE analysis determines the probability of each fault to be revealed. PIE original metric requires sophisticated calculations. It does not cover object-oriented features such as encapsulation, inheritance, polymorphism, etc. These studies were further analysed by many researchers [46] with many extensions and changes proposed to basic PIE model [47] .A tool named PISCES was developed for the implementation of PIE technique as well as to predict testability value [48].

Voas & Miller,1993 [5]proposed a simplification model of sensitivity analysis with the Domain-Range Ratio (DRR). DRR of a specification is defined as follows:

- **Domain-Range Ratio (DRR) =** it is defined as the ratio **d / r**, where d is the cardinality of the domain of the specification and r is the cardinality of the range
- **Testability =inversely proportional to (DRR).** It was found as the DRR of the intended function increases, the testability of an implementation of that function decreases. In other words, high DRR is thought to lead to low testability and vice versa.

DRR depends only on the number of values in the domain and the range, not on the relative probabilities that individual elements may appear in these sets.DRR evaluates application fault hiding capacity. It is a priori information, which can be considered as a rough approximation of testability. This ratio was later reformed and named dynamic range–to-domain ratio (DRDR)[49].Which is a inverse ratio of DRR and determined dynamically to establish a link between the testability and DRDR, the results were though not influential.

Bainbridge 1994[Bainbridge 1994] .In this two flow graph metrics were defined axiomatically:

- ***Number of Trails*** metric which represents the number of unique simple paths through a flow graph (path with no repeated nodes),
- ***Mask*** [k=2] metric, which stands for "Maximal Set of K-Walks", where a k-walk is a walk through a flow graph that visits no node of the flow graph more than k times. Mask reflects a sequence of increasingly exhaustive loop-testing strategies.

These two metrics measure the structural complexity of the code. One of the main benefits of defining these testability metrics axiomatically is that flow graphs can be measured easily and efficiently with tools such as QUALMS.

© 2015, IJARCSSE All Rights Reserved

Yeh & Lin,1998 [50] proposed two families of metrics in their research to evaluate the number of elements which has to be covered with respect to the data-flow graph testing strategies respectively: testable element in all- paths, visit-each-loop-paths, simple paths, structured, branches, statements, and to develop a metric on the properties of program structure that affect software testability.

- **8 testable elements:** no of non comment code lines(NCLOC), p-uses(PU), defs(DEFS), uses(U), edges(EDGE), nodes(NODE), d-u-paths(D_UP) and dominating paths(PATH). As per definition, all those metrics used for normalized source code predict the scope of the associated testing strategies.
- **Testability Metrics**: The testability of each of these factors is calculated individually by taking inverse of the factor value. Thus giving an idea of testing effort required for individual codes.

The model focussed on how to measure software testability under the relationships between definitions and references (uses) of variables that are the dominant elements in program testing. The proposed model represents a beginning of a research to formalize the software testability. This metric can be practiced easily because only a static analysis of the text of a program is required.

Lo and Shi,1998[8]**,** defined testability and assessed testability from three factors namely-Structure factors, which reflect the testability of coding structure of the program segment, Communication factors which measure the testability of OO software by the degree of coupling among classes and objects, and Inheritance factors which contribute to the testability via the inheritance features of OO design. The fault failure model based framework was based on execute-infect-propagate action chain which is essential for the observation or detection of software faults.

- Method Testability $\mathrm{t(M)} = \frac{1}{k(DRR)} \sum_{i=0}^{n} \mathrm{e}(p_i)$ where 1/DRR calculates propagation rate and (1/k)*$\mathrm{e}(p_i)$ is the method execution rate.
- Testability of class=$\mathrm{t(C)} = \frac{1}{M^2} \sum_{i=0}^{n} \mathrm{t}(M_i)$ where is the no of methods and t(M) is method testability as defined above.
- Testability of a class after cohesion consideration T(C)=$\frac{|P \cup Q| - |P|}{|P \cup Q|}$ t(C), where P and Q represents two mutually exclusive classes.
- Testability of a class after coupling T(C)=$\frac{1}{m}$t(C), having *m* as coupling no of class C with t(C) defined as above.
- Testability of a class after Inheritance consideration T(C) = (1-$\frac{m-1}{n}$)t(C), where $\frac{m-1}{n}$ is depth of inheritance tree with t(C) defined as above.

There work was basically surrounded around factors from class to method level. The proposed framework for the estimation of the total testability of an OO system from the individual testability factors of its components based on the OO development paradigm. But the metrics are not empirically analysed further with industrial data.

Nguyen & Robach, 2005[51] focussed on controllability and observability issues. Testability of source code is measured in terms of controllability and observability of source data flow graph which was converted to ITG (Information Transfer Graph) and further to ITN (Information Transfer Net) using SATAN tool. Basically the no of flows within these graphs and diagrams highlighted the scope of testability effort calculation by finding couple value of controllability and observability metrics.

- **$TE_F(M)$= ($CO_F(M)$,$OB_F(M)$)**, the paired metrics for testability effort estimation for a module.
- **$CO_F(M)$**=$T(I_F;I_M)$ / $C(I_M)$ denoted controllability, where $T(I_F;I_M)$ is the maximum information quantity that module M receives from inputs I_F of flow F and $C(I_M)$ is the total information quantity that module M would receive if isolated
- **$OB_F(M)$**= $T(O_F;O_M)$ / $O(I_M)$ denoted observability measure of module M in flow graph. Where, $T(O_F;O_M)$ is the maximum information quantity that the outputs of flow F may receive from the outputs O_M of module M and $C(O_M)$ is the total information quantity that module M can produce on its outputs.

The relative case study showed the testability effort of few flows was (1, 1) which is ideal for testing and for few flows (1, 0.083) which indicates low observability. The SATAN tool used can be used for flow analysis at design as well as code level.

Gonzalez 2009 [52] worked for Runtime testability in component based system with mainly two issues test sensitivity, and test isolation. Where test sensitivity characterises which operations, performed as part of a test, interfere with the state of the running system or its environment in an unacceptable way and Test isolation techniques are the means test engineers have of preventing test operations from interfering with the state or environment of the system. The Runtime testability thus is defined

- **$RTM=M_r / M^*$** where M^* is a measurement of all those features or requirements which are to be tested we want to test and M_r be the same measurement but reduced to the actual amount of features or requirements that can be tested at runtime.

It was found in the study that amount of runtime testing that can be performed on a system is limited by the characteristics of the system, its components, and the test cases them-selves. Though the evaluation of accuracy of the predicted values and of the effect of runtime testability on the system's reliability was not yet established, but the study was useful from built in test capability of systems whether object oriented or component, which surely effects testability.

© 2015, IJARCSSE All Rights Reserved

Khatri,2011[7] presented an approach for improving testability of the software using software reliability growth models. The value of parameter estimates of models for given data set has been presented in the paper along with showing its role in improving the testability of software. It was concluded in the study that the knowledge of proportion of bug complexity helps in improving testability. It will help the project manager in allocating testing efforts and testing tools. The proposed testability measure can result in higher fault detection and can also be used for the determination of modules that are more vulnerable to hidden faults.

Though the study has not given the quantitative measure of improvement of testability but it had shown that prior knowledge of proportion of fault of different complexity lying dormant in the software can ease the process of revealing faults, improving software testability in turn.

V. OBJECT ORIENTED SOFTWARE TESTABILITY METRICS BASED ASSESSMENT VI. FRAMEWORK & TESTABILITY MODEL

All the above discussed metrics captures information about the design and code in terms of fundamental object oriented software features such as Encapsulation, Inheritance, Coupling,, Cohesion and Complexity. Our proposed framework as shown in Fig1 is composed of five major levels –(i) Initial Objet Oriented System which consists of original source code along with its design.(ii)Software Testability Metrics Evaluator related to various static and dynamic metrics techniques (iii) Testability Metrics Interpreter: It helps in interpreting and recommending the required design and code changes iv) Software Testability Transformation: the transformations within the design or code may be performed v) Test Effort Reduction Review : Its basically to verify the observed testability metrics for improvisation results.

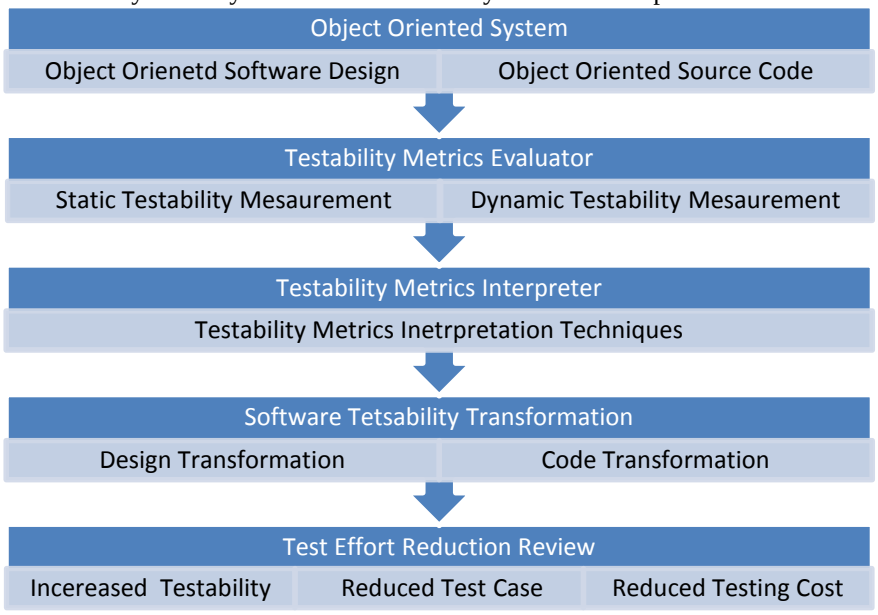

Fig.1 Object Oriented Software Testability Assessment Framework

The proposed framework components help in organizing the assessment process for object oriented software testability (OOST) and it may help in test case reduction by improvising system testability and hence reducing the cost too. The five components stated above rely mainly on suitable testability metrics selection and implementation. At first level after collecting design document with code and description of various modules and classes, a list of applicable testability metrics has to be selected from options as shown in Fig.2. Secondly, the metrics analyzer component will help in collecting the static and dynamic metrics values. Tools such as ECLIPSE, PISCES can be used for static and dynamic metric value collection. These tools may be used as external plug in to be applicable for implementing this framework. After this metric computation, the metrics analysis is performed from testability point of view on design or source code provided. Based on the analysis the design alteration or code testability transformation may be performed. It is followed by testability revaluation by reapplying the particular testability metrics for verification. Thus, this framework establishes a relation between the testability metrics and their role in object oriented software development process by taking into consideration set of static and dynamic testability metrics.

Our framework is based on choice of static and dynamic object oriented software testability metrics (as shown in Fig.2) selection and analysis with correct knowledge of metrics as well as various other aspects of system too. It would be better if the system design document is prepared using standard UML diagrams for various design based metrics assessment [35], [36], [53],etc. System source code under study should be evaluable through static metrics for significant features of an object oriented system such as size, encapsulation, inheritance, polymorphism, coupling, cohesion, and complexity as well as through dynamic metrics applicable at implementation code[6], [48], [51], [54] etc.. Thus by implementing this framework the testability value of a system will surely improvise. However, the metrics listed above need not be restricted to the above mentioned list only. As, we tried to cover all aspect and all metrics in this direction but still actual outcomes only measured after the actual analyses on object oriented software.

© 2015, IJARCSSE All Rights Reserved

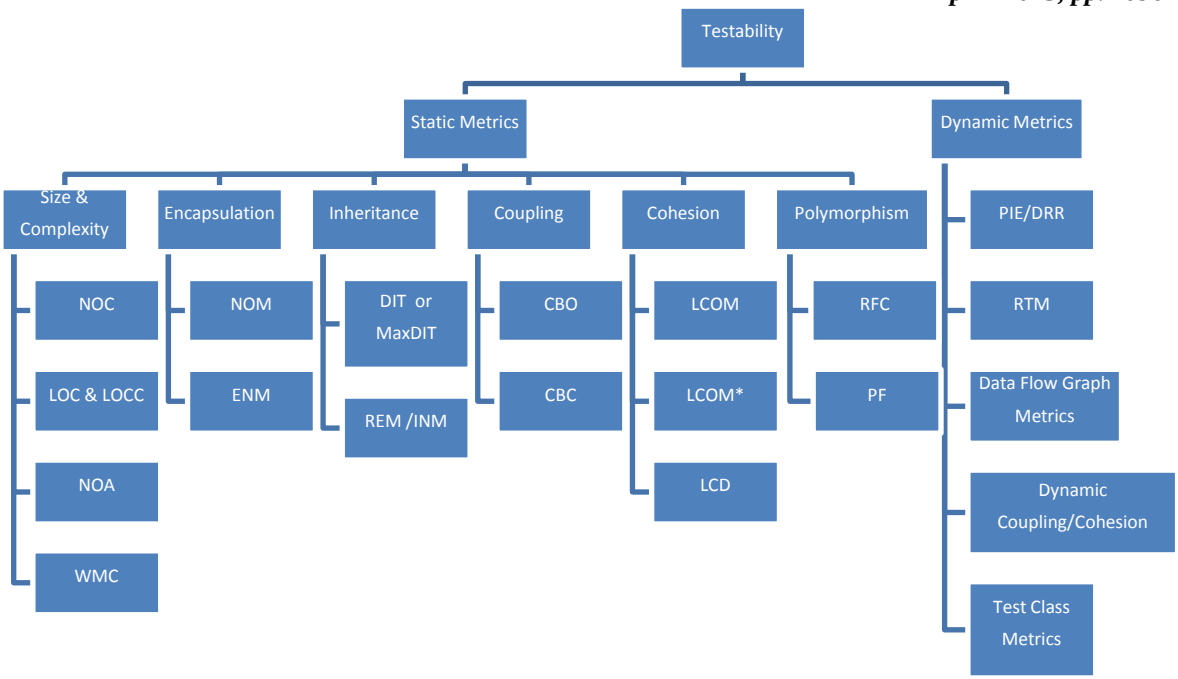

Fig.2 Testability Assessment Metrics Options

VII. CONCLUSION & FUTURE WORK

In this paper a framework using static and dynamic metric along with set of various popular metrics option under each category is proposed to calculate the object oriented software testability. This research paper provides the theoretical framework based on related work done on software testability measurement using object oriented metrics suite. We are trying to gather data from industry practitioners and experts for the choice of selection of an important feature affecting testability and suitable metrics for the assessment of the same. The gathered data will be analysed using appropriate statistical tool for further guidelines to be provided for testability improvisation.

REFERENCES

[1] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.
[2] A. Fernando, "Design Metrics for OO software system," *ECOOP'95, Quant. Methods Work.*, 1995.
[3] L. C. Briand, J. Wust, S. V. Ikonomovski, and H. Lounis, "Investigating quality factors in object-oriented designs: an industrial case study," *Proc. 1999 Int. Conf. Softw. Eng. (IEEE Cat. No.99CB37002)*, 1999.
[4] B. Henderson and Sellers, *Object-Oriented Metric*. New Jersey: Prentice Hall, 1996.
[5] J. M. Voas, K. W. Miller, and J. E. Payne, "An Empirical Comparison of a Dynamic Software Testability Metric to Static Cyclomatic Complexity," 1993.
[6] A. González, É. Piel, H.-G. Gross, and A. J. C. van Gemund, "Minimising the preparation cost of runtime testing based on testability metrics," in *34th IEEE Computer Software and Applications Conference*, 2010, pp. 419–424.
[7] S. Khatri, "Improving the Testability of Object-oriented Software during Testing and Debugging Processes," *Int. J. Comput. Appl.*, vol. 35, no. 11, pp. 24–35, 2011.
[8] B. W. N. Lo and H. Shi, "A preliminary testability model for object-oriented software," *Proceedings. 1998 Int. Conf. Softw. Eng. Educ. Pract. (Cat. No.98EX220)*, pp. 1–8, 1998.
[9] IEEE, "IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)," 1990.
[10] R. V Binder, "Design For Testabity in Object-Oriented Systems," *Commun. ACM*, vol. 37, pp. 87–100, 1994.
[11] R. S. Freedman, "Testability of software components -Rewritten," *IEEE Trans. Softw. Eng.*, vol. 17, no. 6, pp. 553–564, 1991.
[12] J. M. Voas and K. W. Miller, "Software Testability : The New Verification," pp. 187–196, 1993.
[13] J. M. Voas and K. W. Miller, "Improving the software development process using testability research," *Softw. Reliab. Eng. 1992. ...*, 1992.
[14] D. Esposito, "Design Your Classes For Testbility." 2008.
[15] J. Fu, B. Liu, and M. Lu, "Present and future of software testability analysis," *ICCASM 2010 - 2010 Int. Conf. Comput. Appl. Syst. Model. Proc.*, vol. 15, no. Iccasm, 2010.
[16] M. Ó. Cinnéide, D. Boyle, and I. H. Moghadam, "Automated refactoring for testability," *Proc. - 4th IEEE Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2011*, pp. 437–443, 2011.
[17] J. W. Sheppard and M. Kaufman, "Formal specification of testability metrics in IEEE P1522," *2001 IEEE Autotestcon Proceedings. IEEE Syst. Readiness Technol. Conf. (Cat. No.01CH37237)*, no. 410, pp. 71–82, 2001.

© 2015, IJARCSSE All Rights Reserved

[18] R S Pressman, *Software Engineering*. McGraw-Hills, 1992.
[19] T. J. McCabe and C. W. Butler, "Design complexity measurement and testing," *Commun. ACM*, vol. 32, no. 12, pp. 1415–1425, 1989.
[20] M. Genero, M. Piattini, and C. Calero, "Early measures for UML class diagrams," *L'Objet 6.4*, pp. 489–515, 2000.
[21] R. A. Khan and K. Mustafa, "Metric based testability model for object oriented design (MTMOOD)," *ACM SIGSOFT Softw. Eng. Notes*, vol. 34, no. 2, p. 1, 2009.
[22] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng.*, vol. 7, no. 5, pp. 510–518, 1981.
[23] M. Badri, A. Kout, and F. Toure, "An empirical analysis of a testability model for object-oriented programs," *ACM SIGSOFT Softw. Eng. Notes*, vol. 36, no. 4, p. 1, 2011.
[24] M. Bruntink, "Testability of Object-Oriented Systems : a Metrics-based Approach," Universiy Van Amsterdam, 2003.
[25] S. Jungmayr, "Testability during Design," pp. 1–2, 2002.
[26] B. Pettichord, "Design for Testability," *Pettichord.com*, pp. 1–28, 2002.
[27] E. Mulo, "Design for testability in software systems," 2007.
[28] J. E. Payne, R. T. Alexander, and C. D. Hutchinson, "Design-for-Testability for Object-Oriented Software," vol. 7, pp. 34–43, 1997.
[29] Y. Wang, G. King, I. Court, M. Ross, and G. Staples, "On testable object-oriented programming," *ACM SIGSOFT Softw. Eng. Notes*, vol. 22, no. 4, pp. 84–90, 1997.
[30] B. Baudry, Y. Le Traon, G. Sunye, and J. M. Jézéquel, "Towards a ' Safe ' Use of Design Patterns to Improve OO Software Testability," *Softw. Reliab. Eng. 2001. ISSRE 2001. Proceedings. 12th Int. Symp.*, pp. 324–329, 2001.
[31] M. Harman, A. Baresel, D. Binkley, and R. Hierons, "Testability Transformation: Program Transformation to Improve Testability," in *Formal Method and Testing, LNCS*, 2011, pp. 320–344.
[32] A. González, R. Abreu, H.-G. Gross, and A. J. C. van Gemund, "An empirical study on the usage of testability information to fault localization in software," in *Proceedings of the ACM Symposium on Applied Computing*, 2011, pp. 1398–1403.
[33] J. McGregor and S. Srinivas, "A measure of testing effort," in *Proceedings of the Conference on Object-Oriented Technologies, USENIX Association*, 1996, vol. 9, pp. 129–142.
[34] M. Bruntink and A. Vandeursen, "Predicting class testability using object-oriented metrics," in *Proceedings - Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, 2004, pp. 136–145.
[35] B. Baudry and Y. Le Traon, "Measuring design testability of a UML class diagram," *Inf. Softw. Technol.*, vol. 47, no. 13, pp. 859–879, 2005.
[36] B. Baudry, Y. Le Traon, and G. Sunye, "Improving the testability of UML class diagrams," *First Int. Work. onTestability Assessment, 2004. IWoTA 2004. Proceedings.*, 2004.
[37] Y. Singh and A. Saha, "Predicting Testability of Eclipse: Case Study," *J. Softw. Eng.*, vol. 4, no. 2, pp. 122–136, 2010.
[38] S. Khalid, S. Zehra, and F. Arif, "Analysis of object oriented complexity and testability using object oriented design metrics," in *Proceedings of the 2010 National Software Engineering Conference on - NSEC '10*, 2010, pp. 1–8.
[39] L. Badri, M. Badri, and F. Toure, "An empirical analysis of lack of cohesion metrics for predicting testability of classes," *Int. J. Softw. Eng. its Appl.*, vol. 5, no. 2, pp. 69–86, 2011.
[40] M. Badri, "Empirical Analysis of Object-Oriented Design Metrics for Predicting Unit Testing Effort of Classes," *J. Softw. Eng. Appl.*, vol. 05, no. July, pp. 513–526, 2012.
[41] M. Nazir, R. A. Khan, and K. Mustafa, "A Metrics Based Model for Understandability Quantification," *J. Comput.*, vol. 2, no. 4, pp. 90–94, 2010.
[42] M. Nazir, "An Empirical Validation of Complexity Quatification Model," *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, vol. 3, no. 1, pp. 444–446, 2013.
[43] M. Nazir and K. Mustafa, "An Empirical Validation of Testability Estimation Model," *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, vol. 3, no. 9, pp. 1298–1301, 2013.
[44] M. Nazir, R. A. Khan, and K. Mustafa, "Testability Estimation Framework," *Int. J. Comput. Appl.*, vol. 2, no. 5, pp. 9–14, 2010.
[45] J. M. Voas, L. Morell, and K. W. Miller, "Predicting where faults can hide from testing," *IEEE Softw.*, vol. 8, pp. 41–48, 1991.
[46] Z. a. Al-Khanjari, M. R. Woodward, and H. A. Ramadhan, "Critical Analysis of the PIE Testability Technique," *Softw. Qual. J.*, vol. 10, no. April 1998, pp. 331–354, 2002.
[47] J.-C. Lin and S. Lin, "An analytic software testability model," in *Proceedings of the 11th Asian Test Symposium, 2002. (ATS '02).*, 2002, pp. 1–6.
[48] J. M. Voas, K. W. Miller, and J. E. Payne, "PISCES: a tool for predicting software testability," *[1992] Proc. Second Symp. Assess. Qual. Softw. Dev. Tools*, 1992.
[49] Z. a. Al-Khanjari and M. R. Woodward, "Investigating the Relationship Between Testability & The Dynamic Range To Domain Ratio," *AJIS*, vol. 11, no. 1, pp. 55–74, 2003.

© 2015, IJARCSSE All Rights Reserved

[50] P.-L. Yeh and J.-C. Lin, "Software testability measurements derived from data flow analysis," in *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, 1998, pp. 1–7.

[51] T. B. Nguyen, M. Delaunay, and C. Robach, "Testability Analysis of Data-Flow Software," *Electron. Notes Theor. Comput. Sci.*, vol. 116, pp. 213–225, 2005.

[52] A. González, É. Piel, and H.-G. Gross, "A model for the measurement of the runtime testability of component-based systems," in *IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009*, 2009, pp. 19–28.

[53] M. Genero, M. Piattini, and C. Calero, "A survey of metrics for UML class diagrams," *J. Object Technol.*, vol. 4, no. 9, pp. 59–92, 2005.

[54] J. M. Voas, L. Morell, and K. W. Miller, "Using Dynamic Sensitivity Analysis to Assess Testability," 1991.

© 2015, IJARCSSE All Rights Reserved