International Journal of Computer Applications Technology and Research
Volume 4– Issue 5, 359 - 367, 2015, ISSN:- 2319–8656

Object Oriented Software Testability (OOST) Metrics Analysis

Pushpa R. Suri
Department of Computer Science and Applications, Kurukshetra University, Kurukshetra -136119, Haryana, India

Harsha Singhani
Institute of Information Technology & Management (GGSIPU), Janak Puri, New Delhi -110058, India

Abstract: One of the core quality assurance feature which combines fault prevention and fault detection, is often known as testability approach also. There are many assessment techniques and quantification method evolved for software testability prediction which actually identifies testability weakness or factors to further help reduce test effort. This paper examines all those measurement techniques that are being proposed for software testability assessment at various phases of object oriented software development life cycle. The aim is to find the best metrics suit for software quality improvisation through software testability support. The ultimate objective is to establish the ground work for finding ways reduce the testing effort by improvising software testability and its assessment using well planned guidelines for object-oriented software development with the help of suitable metrics.

Keywords: Software Testability, Testability Metrics, Object Oriented Software Analysis, OO Metrics

1. INTRODUCTION

The testing phase of the software life-cycle is extremely cost intensive 40% or more of entire resources from designing through implementation to maintenance are often spent on testing[1].This is due to the enlargement of software scale and complexity, leading to increasing testing problems. A major means to solve these problems is making testing easier or efficient by improving the software testability. Software testability analysis can help developing a more test friendly testable applications. Software testability analysis helps in quantifying testability value. Test designers can use this value to calculate the test cases number that is needed for a complete testing [2]. Software designers can use these values to compare different software components testability, find out the software weakness and improve it and project managers can use the value to judge the software quality, determine when to stop testing and release a program[3].

The purpose of this paper is to examine the software testability measurement metrics at various stages of software development life cycle in object oriented system. The study is done to analyze various OO metrics related to testability and study the literature for various other techniques and metrics for evaluation of testability at design and analysis phase as well as at coding and implementation phase respectively. The study is done because metrics are a good driver for the investigation of aspects of software. The evaluation of these metrics has direct or indirect impact on the testing effort and thus, it affects testability. So, by this study we would be able to serve two objectives: (1) Provide practitioners with information on the available metrics for Object Oriented Software Testability, if they are empirically validated (from the point of view of the practitioners, one of the most important aspects of interest, i.e., if the metrics are really fruitful in practice), (2) Provide researchers with an overview of the current state of metrics for object oriented software testability (OOST) from Design to Implementation phase, focusing on the strengths and weaknesses of each existing proposal. Thus, researchers can have a broad insight into the work already done.

Another aim of this work is to help reveal areas of research either lacking completion or yet to undertaken. This work is organised as follows: After giving a brief overview of software testability in section 2, the existing proposals of OO metrics that can be applied to OO software presented is in Section 3. Section 4 presents an overall analysis of all the proposals. Finally, Section 5 presents some concluding remarks and highlights the future trends in the field of metrics for object oriented software testability.

2. SOFTWARE TESTABILITY

Software Testability as defined by IEEE standards [4] 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 [5][6]. Whereas [2] 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 [7][8]. 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[5].But over the years more such quality factors like understandability, traceability, complexity and test–support capability have contributed to testability of a system[3].All these factors make testability a core quality factor.

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 [9]. 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[10].

3. SIGNIFICANT OBJECT ORIENTED METRICS USED FOR TESTABILITY ASSESSMENT

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 [11], 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 [12].These metrics can be categorized as one of the following object oriented characteristic metrics- Size, Encapsulation, Polymorphism, Coupling, Cohesion, Inheritance and Complexity. Along with that from testability perspective, which is the main motive of study, we have discussed few important UML diagram metric suite too. So, now we present those OO metrics selected for consideration and that may best demonstrate the present-day context of metrics for OOST:

I. CK Metrics Suite [11],[1]

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[13]. 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 Henderson-Sellers [14] 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[14]

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 [12][1]

Metrics for object oriented design (MOOD) metrics suite consists of encapsulation (MHF, AHF), inheritance (MIF, AIF), polymorphism (POF) and coupling metrics (COF). This model was 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. The metrics are as follows:

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.

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):** 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** [15]

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 [16]**:**

i. **Enumeration Metrics (ENM)**: it is the count of all the methods defined in a class.

ii. **Inheritance 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)** [14]: which gives the number of operations in a class

ii. **McCabe Complexity Metrics**[13] **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)**[17]: 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)[18], [19]. 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.

4. SOFTWARE TESTABILITY MEASUREMENT SURVEY

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 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 various development phase like Design Phase[5][20]–[22][8], [23] and Coding Phase[24][25] [26][18]. Quite recently there has been some focus on Testing & Debugging Phase also[27][28]. 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. All these measurement methods specifically from object oriented software systems perspectives are discussed below in brief in coming sections. Our work is the extension of work done by Binder[5] and Bousquet [29] along with giving a framework model for testability implementation during object oriented software development using testability metrics support in upcoming papers.

4.1 Metrics Survey at Design & Analysis Phase

Early stage software design improvisation techniques have highly beneficial impact on the final testing cost and its efficiency. 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. Binder was amongst few of the early researchers who proposed design by testability concept [5] 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 metrics from CK metric suite [11] and McCabe complexity metrics [13] which may be relatively useful for testability measurement. Later lot of work has been done focussed around Binders theory and lot of other new found factors for testability measurement. Voas and Miller [30] [31] also spoke about some factors but mainly in context with conventional structured programming design. Below is the brief description of major contributions made by researchers in the direction of software testability metrics in past few years.

Binder,1994 [5] suggested few basic popular structural metrics 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.
Binder's work which was based on Ck metric suite along with few other object oriented metrics under review has been kept as benchmark during many studies found at later stages. The study and reviews did not lead to concrete testability metrics but laid a ground work for further assessment and analysis work.

McGregor & Srinivas, 1996 [32] 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 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.

Khan & Mustafa,2009 [16] proposed a design level testability metrics name Metrics Based Testability Model for Object Oriented Design (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.

Khalid et. al. ,2011 [33] proposed five metrics model based on CK metrics suite[11] and MTMOOD[16] 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 2. 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.

Nazir Khan,2013[34]–[36] 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 [34], [35] The metrics used for the assessment of these two factors were based on Genero metrics suite [15] 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 2. 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 [37] by them. Also, not much elaborative study was conducted on complexity and understandability correlation establishment with basic object oriented features.

4.2 Metrics Survey at Coding & Implementation Phase

The metrics study at source code level has gained more popularity in the industry for planning and resource management. Generally the metrics used at this level 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 Bruntink and others [38].

Voas & Miller 1992 [2], [7], [39] 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 [40] with many extensions and changes proposed to basic PIE model [41] .

Voas & Miller,1993 [42]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)[43].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] propose testability assessment on flow graphs. 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.

Yeh & Lin,1998 [44] 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.

Jungmayr 2002 [45] study was basically on metrics based on software dependencies and certain system testability metrics. The study was based on four metrics required to analyse component testability from dependency perspective. Such dependencies called test-critical dependencies were identified and their impact was evaluated on overall testability. To automate the identification of test-critical dependencies a prototype tool called ImproveT. The Metrics used for the analysis were:

- **Average Component Dependency (ACD):** It is the total count of component dependency by total no of components in the system.
- **No of Feedback Dependency (NFD):** It is the total number of feedback dependency.
- **Number of Stubs to Break Cycles (NSBC):** It is the total number of stubs required to break cycles.
- **No of Component within Dependency Cycles (NCDC):** It is the total number of components within all dependency cycles.
- **Reduction metrics r(d)** – These metrics were further reduced in percentile form and named **rACD, rNFD, rNSBC, rNCDC**. These reduction metrics which are themselves not highly correlated were then studied for system structure, class coupling, etc. and other perspectives.

It was found in the research that smaller metric values mean better testability for all metrics described above. The approach was helpful in identifying design and test problems.

Bruntink 2003[19], [38] 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 [11] 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.

Nguyen & Robach, 2005[46] 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 [47] 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 themselves. 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.

Singh & Saha (2010) [48]did empirical study to establish relation between various source code metrics from past [11][14] and test metrics proposed by [19] 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 [18] study was based on adapted model MTMOOD proposed by [16], at source code level named as MTMOOP. They adapted this model to the code level by using the following source code metrics: NOO [14], DIT and CBO [11]. 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.

Badri et. al.,2012 [49], [50] further did study, which 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 2 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

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

5. CONCLUSION

This paper analysed and surveyed the role of various object oriented metrics in software testability. The purpose was to increase the basic understanding of testability evaluation and quantification techniques for object oriented systems using various researched metrics based on popular OO metrics suits. We mainly wanted to survey the existing relevant work related to metrics for object oriented software testability at various stages of software development, providing practitioners with an overall view on what has been done in the field and which are the available metrics that can help them in making decisions in the design as well as implementation phases of OO development. This work will also help researchers to get a more comprehensive view of the direction that work in OO testability measurement is taking.

During the study we found out the number of existent measures that can be applied to object oriented software at initial design stage is low in comparison with the large number of those defined for coding or implementation phase. What we found is that despite of all the efforts and new developments in research and international standardization during the last two decades, there is not a consensus yet on the concepts, techniques and standard methods used in the field of software testability. This, in turn, may serve as a basis for discussion from where the software engineering community can start paving the way to future agreements.

6. REFERENCES

[1] R S Pressman, *Software Engineering*. McGraw-Hills, 1992.

[2] J. M. Voas and K. W. Miller, "Software Testability : The New Verification," pp. 187–196, 1993.

[3] 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.

[4] IEEE, "IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)," 1990.

[5] R. V Binder, "Design For Testabity in Object-Oriented Systems," *Commun. ACM*, vol. 37, pp. 87–100, 1994.

[6] R. S. Freedman, "Testability of software components -Rewritten," *IEEE Trans. Softw. Eng.*, vol. 17, no. 6, pp. 553–564, 1991.

[7] J. M. Voas and K. W. Miller, "Improving the software development process using testability research," *Softw. Reliab. Eng. 1992. …*, 1992.

[8] D. Esposito, "Design Your Classes For Testbility." 2008.

[9] 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.

[10] 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.

[11] 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.

[12] A. Fernando, "Design Metrics for OO software system," *ECOOP'95, Quant. Methods Work.*, 1995.

[13] T. J. McCabe and C. W. Butler, "Design complexity measurement and testing," *Commun. ACM*, vol. 32, no. 12, pp. 1415–1425, 1989.

[14] B. Henderson and Sellers, *Object-Oriented Metric*. New Jersey: Prentice Hall, 1996.

[15] M. Genero, M. Piattini, and C. Calero, "Early measures for UML class diagrams," *L'Objet 6.4*, pp. 489–515, 2000.

[16] 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.

[17] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Softw. Eng.*, vol. 7, no. 5, pp. 510–518, 1981.

[18] 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.

[19] M. Bruntink, "Testability of Object-Oriented Systems : a Metrics-based Approach," Universiy Van Amsterdam, 2003.

[20] S. Jungmayr, "Testability during Design," pp. 1–2, 2002.

[21] B. Pettichord, "Design for Testability," *Pettichord.com*, pp. 1–28, 2002.

[22] E. Mulo, "Design for testability in software systems," 2007.

[23] J. E. Payne, R. T. Alexander, and C. D. Hutchinson, "Design-for-Testability for Object-Oriented Software," vol. 7, pp. 34–43, 1997.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] M. R. Shaheen and L. Du Bousquet, "Survey of source code metrics for evaluating testability of object oriented systems," *ACM Trans. Comput. Log.*, vol. 20, pp. 1–18, 2014.

[30] J. M. Voas, “Factors that Affect Software Testability,” 1994.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] M. Nazir, R. A. Khan, and K. Mustafa, “Testability Estimation Framework,” *Int. J. Comput. Appl.*, vol. 2, no. 5, pp. 9–14, 2010.

[38] 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.

[39] J. M. Voas, L. Morell, and K. W. Miller, “Predicting where faults can hide from testing,” *IEEE Softw.*, vol. 8, pp. 41–48, 1991.

[40] 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.

[41] 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.

[42] J. M. Voas, K. W. Miller, and J. E. Payne, “An Empirical Comparison of a Dynamic Software Testability Metric to Static Cyclomatic Complexity,” 1993.

[43] 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.

[44] 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.

[45] S. Jungmayr, “Testability measurement and software dependencies,” 2002.

[46] 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.

[47] 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.

[48] Y. Singh and A. Saha, “Predicting Testability of Eclipse: Case Study,” *J. Softw. Eng.*, vol. 4, no. 2, pp. 122–136, 2010.

[49] 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.

[50] 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.