CHAPTER 2
LITERATURE SURVEY

2.1 INTRODUCTION

This chapter presents the result of a systematic literature review conducted to collect evidence on software testability measurement of object oriented design. The first research on software testability is appeared in the year 1975. It is adopted in Boehm and McCall software quality model, which build the foundation of ISO-9126 software quality model. Since 1990s, software engineering community began to introduce experimental research on software testability. Testability analysis has been an important research area since 1990s, and became more pervasive in 21st century. A number of researchers Stevens et al. (1974), Karoui et al. (1996), Kolb et al. (2006), Behshid et al. (2009) & Nguyen et al. (2010) worked on software testability, but in the perspective of conventional structured design. The question of software testability has been revived with the object oriented development process by experts R.V. Binder (1994), Voas & Miller (1995) & Voas J. M. (1996). Despite the fact, that object oriented technology has now been widely used by the software industry, but only a few research works have been devoted to explore the concepts of software testability in object oriented systems. Unfortunately, these research works have not been widely accepted and hence, not been adopted in practice by industry personnel. Following sections systematically summarizes the relevant efforts made by researchers in this area.

The existing literature on testability can be divided according to the software development life cycle (SDLC) phases at which it is considered.

2.2 RELATED WORK

The critical review of the related work on the topic can be categorized as follows:

2.2.1 Testability at Analysis Phase

Study done by Goel et al. (2012) focused on the testability of the object oriented software systems and identified that flexibility at the variable points of the object oriented development, selected for framework instantiation, significantly influence the testability of object oriented software at every level of testing. In this study, author proposed a testability guidelines taking into account the flexible aspect of the variable point to measure testability in the requirement analysis phase. The constraint of this study is that the explanations are based on the hook documented framework and an empirical validation has been needed for the proposed guidelines.

2.2.2 Testability at Design Phase

Many researchers explored testability at design phase which is explained as follows:

Karoui et al. (1996) discussed about communication software in order to handle the complexity of tests for communication protocols and acknowledged it as the design for testability (DFT). The key objective of DFT is to decrease the cost and the complexity of test processes. Study stated testability movement and its analysis require the use of estimation methods or measures. Estimation method of testability helps software designers to recognize those parts of the requirement specifications that are complex to test. Subsequently, theoretical guidelines can be proposed to design for testability.

Baudry et al. (2005) highlighted the importance of individual types of class coupling for testability metrics, as well as established a relation of coupling and class interaction metrics that finalize testability. Study mainly concerned with the testability relation to the testing effort focusing on object oriented static designs based on UML (Unified Modeling Language) class diagrams. This work mainly focused on the testability of UML class diagrams and approximates the number of class interactions from UML class diagram, which can be applied to estimate the testing effort and the design testability. Study also suggests for the improvement of class interactions regarding reduction in number and complexity. The author gave a graph based model, 'the class dependency graph', using UML class diagram features in order to evaluate the complexity of class interactions which is used for the testability measurement. Still the model is not validated and does not clearly indicate the cause effect relation between class interactions metric and testability.

In the study done by Dino Esposito (2008) they argued that testability and security are two major quality attributes to guarantee the quality of a developed software system. Considering view of this fact author stated that it should be planned near the beginning exclusively at the design phase of development life cycle. Dino argued in his study that software testability is an outcome from a number of characteristics of the software being tested. Study highlighted that software testability is a result from a number of characteristics of the code being tested that the development team should ideally guarantee: modifiable, control and simplicity. In conclusion, author theoretically highlighted to design the classes for testability taking into consideration of testability characteristics namely controllability, modifiability and simplicity. Work done by Chowdhary, V. (2009) also argues on applying testability notions and laying down rules to make sure testability consideration are made in features planning and design phase. Study put the foundation by briefly going over testability concept and discussed about the typical thought process in a test developers mind and present key approaching into why practicing testability is complex and tedious task. Author then present real life examples that he has encountered in his profession which show how testability considerations could have made our testing simpler and discussed the impact of testability on design phase of development life cycle. Building from these examples, he presented a checklist for each of the testability main beliefs. Study finally presents an exercise where we can apply available checklist to a complex event processing module and understand the benefits of applying testability concepts.

In the study done by Khan et al. (2009) proposed metrics based testability model for object oriented design (MTMOOD).The developed model estimates testability with the help of object oriented properties namely Encapsulation, Inheritance and Coupling. Subsequently, the proposed model for the estimation of object oriented software testability has been validated. In this study author measured object oriented software testability directly, without including testability factors. On the other hand researchers and practitioners have made significant amount of effort and contribution in the way of investigating testability factors in common and object oriented software in particular .For the reason that an accurate measure of software quality depends on testability measurement, which in turn depends on the testability factors that can affect object oriented software testability at design phase.

Study done by Anthony et al. (2009) described how the Lean 123 with automated software tests approach yields significant cost saving and quality improvement benefits in software products. Lean 123 is a Lean+ initiative that establishes a three item checklist for executing tasks. The checklists are 1) Establish Clear Priorities 2) Eliminate Bad Multitasking-Focus and Finish 3) Limit the Release of Work In Process (WIP) to Deliver product. Furthermore, this study provided recommendations on the best practices of designing for software testability using a Lean 123 approach and technology enablers for testing frameworks such as NUnit that can be applied to all software projects. The NUnit interface provides a visual stop light report of the test results in which a green light represents a test that passed and a red light represents a test that failed. Author claimed that designing for testability should always be done at the design phase of development life cycle. Moreover, no quantitative testability measurement model has been presented in this study.

Work done by Aminata Sabane (2010) highlighted the importance of reducing testing effort by using design patterns and anti patterns of micro architectures. Study includes identification and quantification impact of micro architectures on system testability and testing. For example, work presented a list of micro architectures and request testers to recognize from this list, the simplest and the worst to test. Such an analysis would allow collecting opinions on the testability of systems containing micro architectures with design patterns and anti patterns. Depend on questionnaires analysis; study will classify the micro architectures according to their impact on system testability. Further based on outcomes, author proposed refactoring technique and guidelines for making design decisions, and provides theoretical guidelines to improve system testability at design phase of the development life cycle. In this study the quantitative measure for improvement of testability was not given and theoretical guidelines are not clear about the cause effect relation between design patterns and anti patterns.

In the study done by Singh & Saha (2010) they highlighted the importance of software contracts for testability improvement and reduce the testing effort of object oriented class at design phase of development life cycle. Study established the relationship between testability and software contracts. The same is explained through an example of a queue class, written in C++ language. Authors claim to reduce the number of test cases, to test a class with the help of software contracts. Moreover, this method was not empirically validated and not applicable in the context. Unfortunately; these outcomes have not been widely accepted and hence, have not been adopted in practice by the practitioners. In addition, the guidelines provided by the authors are not sufficient for both structural and behavioral architecture.

Bousquet et al. (2010) highlighted testing procedure carried out to find faults/errors in a system and briefly present the synchronous approach and its specificities with respect to testability. Study used testability metrics to identifying parts of a design that are complicated to test. In this study, author focus on two testability metrics defined for system, written in LUSTRE/SCADE, is a declarative data flow language. An intuitive interpretation was produces for these metrics. The objective of this research is to confirm whether intuitive interpretation can be consolidated with actual evidences. In this work, study focus on testability of synchronous reactive data flow systems, expressed as controllability and Observability. Intuitively, they can be used to detect low testable parts of the systems. Moreover, the reactive system outputs usually depend on the system history, not on its current input. In this work author told, validation of testability is a hard work. The main difficulty relies on the fact that the effort to test is subjective and depends on each point of view.

Work done by Khatri et al. (2011) developed software reliability growth model parameters and its impact on software testability. The Software Reliability Growth Model (SRGM) is the technique, which can be used to measure the software reliability, develop schedule status, test status, and monitor the changes in software reliability performance. The overall impact and application of SRGM parameter measures have been used in quantifying the testability of software. Study used failure data of object oriented software, developed under open source software platform, namely MySQL, Python and SQL Client. In this study, author has not given the quantitative measure of software testability. However, only discussed theoretical approach for measuring software testability. In this work it has been revealed that earlier knowledge of proportion of fault of complexity in the software can simplify the process of revealing faults and as a result testability of software can be improved.

2.2.3 Testability at Source Code Level

Testability at source code level considered by various researchers which is enlighten as follows:

Voas et al. (1992) developed PISCES: a tool for predicting software testability. This is the commercial software testability tool which is written in C++. PISCES generates testability measures by creating an "instrumented" copy of the program and afterward compiling and executing the instrumented copy, which is about 10 times as large as the original source code, with inputs that are either supplied in a file or PISCES uses random distributions from which it generates inputs. PISCES testability postprocessor inputs all prospect measures and allows the user numerous choices of how the testability will be presented: moreover for a location, component or the whole program. Study argued this tool improves testing, debugging and hence improves software quality.

Voas & Miller (1996) have dealt with software testability metrics that depends upon inputs and outputs artifacts of a software module. To quantify testability, authors proposed propagation infection and execution (PIE) analysis technique; but quantifying testability through the PIE analysis technique was very complex and has high complexity. This is a dynamic method for statistically measuring the effects that a location of a program has on the program's computational behavior. PIE analysis gathers information relating to the semantics of faults. It reveals the existences of faults not the method that directly evaluate the ability of inputs to reveal the existences of faults. Instead, it identifies locations in a program where faults exist, that more likely to remain undetected during testing. Due to these limitations this technique is not adopted by industry personals.

Fault/Failure method given by Voas & Miller (1996) is based on fault/failure architecture and it estimates the probability of the three characteristics of a location, the probability that the location is executed on inputs, selected from the assumed input distribution of the software.

1. The probability that if a mutant exists at this location, it will adversely, change the states.
2. The probability that if the data state is adversely changed that will propagate to the output.
3. At a certain location, a probable fault/error could result in a failure if and only if:

- A fault must be executed.
- The fault must have an effect on the condition of the program in a way different than what the state of program would have been handle the fault not existed. This is termed as having an infection in the state.
- The erroneous program state must propagate to an output state.

If the possibility of a program's failure is confined to this model, the probability that a fault/error will turn into a failure, will be the product of execution probability, infection probability and propagation probability. Furthermore, a lower bound testability of a program is achieved as the least product above over all location in the program.

Work done by Baudry et al. (2001) discussed two configurations of object oriented software that can weaken its testability. Study described particular design patterns micro architectures, widely used in the object oriented domain, for achieving basic refinement, operators using the state design pattern and the Abstract Factory. At a first sight, it shows the normal way of including such testability constraints to a UML design, study found that this rules are very complex to implement in OCL (Object Constraint Language) and may lead to an unrealistic solution. Another option is to classify the pattern applications in respect of UML diagrams at the Meta model level of the UML: the models for the patterns are classified in terms of roles. This Meta model defines what a pattern application is, and embeds the desired testability characteristics at a general level. Finally author produced automatic verification tools to check whether a pattern is safely implemented at code level or not.

From the study of Bruntink & Deursen (2004) the class testability can be assessed by analyzing two categories of source code level factors namely, test case construction factors and test case generation factors. In addition, study presents a source code metrics for exploring the testability of object oriented Java system, and identified testability factors through source code metrics. This study is mainly concerned with identifying and evaluating the factors of testability in object oriented software and metrics related to the factors, which are been supported by the case studies. The authors used the source code analysis for characterizing the software testability. In this research, authors identify possible relevant metrics to predict the class testability and analyzed the approach in theoretical manner.

Work done by Liang Zhao (2006) proposed beta distribution method to indicate software testability. When integrating testing effectiveness, author theoretically proved that the beta distribution method can speed up testing process and test value at the same time. The proposed work can be separated into two parts to achieve the study objective. The first part of this work is to highlight the importance of distribution method as testability indication. Subsequently, second part of the study tries to find ways to deduce the beta distribution for related software and test criterion information. The second part of this study is further divided into three steps; first step is to classify appropriate testing criterion's effectiveness measure. The objective of second step is to introduce criterion's effectiveness information. The third step tries to prove that when integrating effectiveness information, the distribution method can provide reasonable measurement on the quantity and quality of testing.

Work done by Gonzalez et al. (2009) presented a qualitative model for runtime testability that complements Binder's classical testability model, and provide a generic guidelines for assessing the degree of runtime testability of a software, depend on the ratio of what can be tested at the runtime versus what would have been tested throughout development life cycle. A measurement is devised for the client server architecture based on test coverage with the help of graph model of the system's architecture. Subsequently, two software testability studies are presented for two component based systems, showing how to measure the runtime testability of a system. Moreover, this approach is only suitable for data flow or client server architectures.

In the study done by Tsung et al. (2009) author argued that research on software testability has been developed in different perspectives. In the past, a dynamic technique for quantifying software testability was proposed and called 'propagation infection and execution' (PIE) approach. Previous research works show that ''propagation infection and execution' technique can complement of software testing. Despite the fact that, PIE technique required a lot of computational effort in measuring the testability of software components. Considering view of this fact, author proposed an Extended PIE (EPIE) method to accelerate the previous PIE analysis, depend on generating group testability as a replacement for component testability. Study divided Extended PIE technique into three steps:

- Breaking a program into blocks.
- Dividing blocks into groups.
- Marking target statements.

To implement Extended PIE (EPIE) method, study further developed a mechanism called ePAT (extended PIE Analysis Tool) to support for identify the components which will be analyzed. The extra overhead is required in this study is to identify and mark analyzed component before executing Extended PIE (EPIE) analysis.

Work done by Jianping Fu & Minyan Lu (2009) proposed a request oriented method for testability measurement. Study stated testability measurement method is limited there is maybe no appropriate method presented for testability measurement requests. To solve this problem author used request oriented approach. This approach can select appropriate elements from a self contained software testability measurement guideline according to the different measurement requests to complete testability measurement. Firstly, all testability measurement requests are identified. Secondly testing requests is derived from testability estimation guidelines consisting of factors related with software testability measurement of all kind of software. At last a new measurement guideline is provided with the help of selected elements and results can be measured based on the request of users. However, this approach was not validated and applicable only at implementation stage of development life cycle.

2.2.4 Testability at Testing Phase

Many researchers explored testability at testing time which is explained as follows:

An approach adopted by Jungmayr (2002) shows a concept for estimating software testability through integration testing. He identified local dependencies that positively contribute and are responsible for overall testability. Jungmayr's concept used reduction metric to calculate the effect of individual factors in software testability to find out required testability metric. This metrics has been introduced to assess the impact of a particular dependency on a particular quality characteristic (like testability). A value of reduction metrics bigger than zero in general means that testability improves if dependency in removed. It is advocated that reduction metrics can be used to rank dependencies based on their impact on the overall testability.

In the study done by Jianping et al. (2010) they summarized available methods of software testability analysis from the aspect of test cost, sensitivity and testable characteristics. By analyzing the analysis object, test strategies and measurement results, the present state and shortage of software testability analysis are summed up. According to the development trend of software testability analysis, some future research directions are listed by author as following:

- Research for software testability analysis based on testable characteristics.
- Research for factor analysis of software testable characteristics.
- Research for software testability evaluation.
- Research for object oriented design testability analysis.
- Crossover research of above directions.

In the study done by Mourad and Fadel (2012) investigated experimentally the relationship between object oriented metrics and testability of classes. Author highlighted testability in the context of unit testing effort. Study collected data from open source Java software systems for which JUnit test cases exist. To capture the testing effort of classes, study used object oriented metrics to measure the matching JUnit test cases. Classes were organized, according to the required JUnit testing effort in two categories namely high and low. In order to calculate the relationship between object oriented metrics and JUnit testing effort of classes, author used logistic method. The findings in this work viewed as explanatory rather than conclusive.

2.2.5 Testability at Development Life Cycle

Testability throughout development life cycle considered by various researchers which is enlighten as follows:

In the study done by Voas & Miller (1992) suggested to take software testability analysis into consideration throughout the development life cycle. This analysis can be made from software requirement specifications, planning, designing and the coding itself. Author highlighted the guidelines provided by software testability is helpful during system design, implementation, testing, and quality assurance. In this work, study has illustrated how testability with respect to random black box testing has importance throughout the software development lifecycle. Besides, static analysis of the domain range ratio (DRR) gives insight early in the specification and design stages. In all these applications, testability gives a new perspective on the relationship between software quality and ability to measure that quality.

R.V. Binder (1994) had done a work showing the importance for improving software testability in system development life cycle. He proposed a fishbone model, which shows that, software testability is a result of six factors: (1) Characteristics of the representation (2) Characteristics of the implementation (3) Built in test capabilities (4) The test suite (test cases and associated information) (5) The test support environment (6) The software process in which testing is conducted. But unfortunately all above factors are applicable only at later stages of development life cycle. Binder's work has no clear relationship with object oriented design constructs and proposed testability factors. Moreover, proposed model was not validated and not suited for measuring testability of object oriented software at initial stage of development life cycle.

Work done by Jimenez et al. (2005) highlighted the benefits of improving the system development life cycle by making specific improvements in the area of software testing and validation. Study provided an overview of design for testability (DFT), its application in hardware and software development process, and its relationship to reliability and robustness. In addition, study demonstrated DFT is more expensive in the case of short term development process and it is cheaper in the long term development process. DFT can greatly decrease testing times and virtually removes production delays as well as facilitating diagnosis and repair in the development field. DFT will outcome in better fault isolation and fault coverage, shorter testing time, good quality product, required shorter time to deliver in market, and lower development lifecycle cost. As a result, authors provided only theoretical explanation for above stated theories as well as he argued DFT is basically a management issue and not a technology issue.

Work done by Mulo (2007) integrated the importance of testability measurement throughout the software development life cycle. Author focused on improving the controllability of a software project with minimal addressing of Observability characteristics. Study highlighted controllability along with Observability can become operational in practice, through strategy that are employed in the software development life cycle to make testing easier. These strategies in the development process should be considered in requirements and architecture specification, to implementation. However, estimating testability during the entire development life cycle is very expensive and error prone.

A complete charting of the existing models/methods has been done in Table 2.1.

Table 2.1: Comparison of Testability Measurement Models/Methods

S.N0.	**Researcher**	**Title of Study**	**Year**	**Model/ Method**	**Validation**	**SDLC Phase**
1	Goel et al.	Testability Estimation of Framework Based Applications	2012	Framework	No	**Analysis Phase**
2	Karoui et al.	Specification Transformations and Design for Testability	1996	DFT	No	**Design Phase**
3	Baudry et al.	Measuring Design Testability of a UML Class Diagram	2005	Class Dependency Graph	No	
4	Dino Esposito	Design Your Classes for Testability	2008	Design Tips	No	
5	V. Chowdhary	Practicing Testability in the Real World	2009	SOCK Approach	No (Based Experience)	
6	Khan et al.	Metric Based Testability Model for Object Oriented Design	2009	MTMOOD	Yes	
7	Anthony et al.	A Lean Approach to Designing for Software Testability	2009	Lean Approach	No	

8	A. Sabane	Improving System Testability and Testing with Micro Architectures	2010	Design Pattern	No	**Design Phase**
9	Singh & Saha	Improving the Testability of Object Oriented Software through Software Contracts	2010	Software Contracts	No	
10	Bousquet et al.	Analysis of Testability Metrics for Lustre/Scade Programs	2010	Data Flow	No	
11	Khatri et al.	Improving the Testability of Object Oriented Software During Testing and Debugging Processes	2011	SRGM	No	
12	Voas et al.	PISCES: A Tool for Predicting Software Testability	1992	PIE Method	Yes	**Coding Phase**
13	Voas and Miller	Dependability Certification of Software Components	1996	PIE	No	
14	Voas and Miller	A Tutorial on Software Fault Injection	1996	Fault/Failure Method	No	
15	Baudry et al.	Towards a 'Safe' Use of Design Patterns to Improve Object Oriented Software Testability	2001	Testing Conflict	No	

16	Bruntink et al.	Predicting Class Testability using Object Oriented Metrics	2004	Source Code Analysis	Theoretical Justification	**Coding Phase**
17	Liang Zhao	A New Approach for Software Testability Analysis	2006	Beta Distribution Technique	Program Test (Case Study)	
18	Gonzalez et al.	A Model for the Measurement of the Runtime Testability of Component Based Systems	2009	Component Interaction Graph	Conducted Test	
19	Tsai et al.	A Study of Applying Extended PIE Technique to Software Testability Analysis	2009	Extended PIE Technique	Conducted Test	
20	Jianping& Minyan	Request Oriented Method of Software Testability Measurement	2009	Request Oriented Method	No	
21	Jungmayr	Testability Measurement and Software Dependencies	2002	Integration Testing	No	**Testing Phase**
22	Jianping et al.	Present and Future of Software Testability Analysis	2010	Test cost Analysis	No	
23	Mourad and Fadel	Empirical Analysis of Object Oriented Design Metrics for Predicting Unit Testing Effort of Classes	2012	Logistic Methods	Yes	

24	Voas and Miller	Improving the Software Development Process using Testability Research	1992	SDLC	No	**SDLC**
25	R. V. Binder	Design for Testability in Object Oriented Systems	1994	Method is not Mentioned	No	
26	Jimenez et al.	Design for Testability	2005	DFT	No	
27	Mulo	Design for Testability in Software Systems	2007	Architecture Specification	No	

After an exhaustive review, it is evident that testability measurement should be done at design phase of development life cycle. To measure testability at design phase it is important to identify commonly accepted testability factors. Now in the next Section i.e. Section 2.3 we will discuss about the testability factors.

2.3 TESTABILITY FACTORS

It is evident from exhaustive literature survey that there is an opposition among practitioners in taking into consideration the testability factors for measuring testability of object oriented software in general and at design phase. A consolidated table for the testability factors identified by various experts is concluded in Table 2.2. It is clearly highlighted from the table that Observability, controllability, Flexibility, Traceability, Understandability and Modifiability are the commonly accepted testability factors.

Table 2.2: Testability Factors Consider by Various Experts

Testability Factors ➔ **Author/Study⬇**	Observability	Controllability	Flexibility	Traceability	Understandability	Modifiability	Fault locality	Simplicity	Complexity	Development process	Separation of concerns
Binder (1994)	✓	✓	✓	✓		✓				✓	
Bach (1999)		✓			✓			✓			
Jungmayr (2002)	✓		✓	✓		✓	✓		✓		✓
Wang (2003)		✓	✓		✓			✓			
Baudry et al. (2005)			✓						✓		
Zaho et al. (2006)	✓		✓		✓	✓					
E Mulo (2007)	✓	✓		✓		✓				✓	
Dino Esposito (2008)		✓				✓		✓			
Khatri et al. (2011)		✓	✓			✓					
Malla P. et al. (2012)				✓	✓	✓					

P. Nikfard (2013)		✓				✓					
Joshi et al. (2014)	✓		✓								

An effort has been made to recognize the testability factors that truly affect testability measurement at design phase. Modifiability and Flexibility are the key testability factors that truly affect software testability measurement and fulfill the quality criteria, Modifiability quality criteria is understandability, traceability, self descriptiveness and Flexibility quality criteria is simplicity and complexity. Therefore, without any loss of generality, it comes into view realistic to include Modifiability and Flexibility for testability measurement at design phase.

2.4 LITERATURE SURVEY ON MODIFIABILITY

The critical review of the related work on the topic can be summarized as follows:

In the study done by Kiewkanya et al. (2011) proposed a new metrics to assess modifiability of object oriented design. These metrics can be measured with the help of class diagrams. Proposed metrics viz. Mod_{Gen}, Mod_{Agg}, Mod_{Com}, $\text{Mod}_{\text{CAssoc}}$, $\text{Mod}_{\text{AssocC}}$, Mod_{Dep} and Mod_{Real} developed by considering the number and the kind of relationships among classes. In this study, for comparing modifiability of software with different sizes, author introduced the concept of average and modifiability will be estimated in the average case. To validate the new proposed metrics, a Pearson correlation analysis between modifiability estimated by the proposed metrics and human beings' intuition was performed. The result shows that $\text{AvgMod}_{\text{Sys}}$ and Modifiability score have a negative correlation.

Study done by Lulu et al. (2009) applied Wood's task complexity model to propose a general analytical model that describes the characteristics of maintenance tasks and the analytical dimensions of modifiability independent of the individual maintainers. In this study three analytical dimensions described task complexity viz. component, coordinate, dynamic and total complexity is determined by all three dimensions. By applying task analysis theory, the proposed model captures the complexity inherent in the maintenance task independent of individual characteristics of maintainers. In this work, author theoretically justify the proposed model provides greater insight into the relationship between internal software attributes (e.g. structural measures) and external attributes of maintenance process (e.g. effort).

Bengtsson et al. (2004) proposed scenario based software architecture analysis methods that focus exclusively on modifiability. It consists of five major steps:

1. Set goal: determine the aim of the analysis
2. Describe software architecture: give a description of the relevant parts of the software architecture
3. Elicit scenarios: find the set of relevant scenarios
4. Evaluate scenarios: determine the effect of the set of scenarios
5. Interpret the results: draw conclusions from the analysis results

ALMA distinguishes the following goals that can be pursued in software architecture analysis of modifiability: maintenance prediction, risk assessment and software architecture comparison. Study draw up an explanation of the software architecture. This explanation will be used in the assessment of the scenarios. Unfortunately, it contains insufficient detail to perform an impact analysis for each of the scenarios and draw conclusions. Moreover, this approach is only suitable for client server architectures.

It is evident from exhaustive literature survey on modifiability that no comprehensive modifiability measurement model exists to measure modifiability of object oriented software at design phase of development life cycle.

2.5 LITERATURE SURVEY ON FLEXIBILITY

The critical review of the related work on the topic can be summarized as follows:

In the study done by Xiaoguang & Bo (2013) highlighted flexibility is an essential request and is also a way that must be taken during the establishment process of ERP. Study proposed flexibility estimation model of ERP system depend on fuzzy-analytic- network-process (FANP). The local weight of criterion and index is derived by fuzzy preference programming (FPP) technique. An unweighted super matrix depend on the network configuration of index system is developed, and the limit super matrix is generated. In this study flexibility level of ERP system can be estimated by the weights and scores of ERP. A numerical example is given by the proposed method, and the result is shown that it can deal with this kind of problem. However, this approach was not validated and applicable only establishment process of ERP.

Work done by Jingchun et al. (2011) proposed non linear model to measure software flexibility taking into account the relationship between operational control force and deformation of the software. In this study software flexibility is calculated by the second order cone programming (SOCP) approach. Second order cone program (SOCP) a linear function is minimized over the intersection of an affine set and the product of second order (quadratic) cones. Furthermore, no quantitative flexibility measurement model has been presented in this study.

In the study done by Limin Shen & Shangping Ren (2010) introduced two new concepts that are flexible change and Flexible Point (FXP). Flexible Point associated with, a set of flexibility indices i.e. flexible degree, flexible distance, flexible capacity and flexible force. Further study categorized flexible points into five different types, namely potential FXP, current FXP, available FXP, required FXP and used FXP and four different levels that are Self Adaptive FXP, High level User FXP, developer Level User FXP and Low level User FXP,. In this study, author discussed the associations and differences between Flexible Point (FXP) and FXP impact on the software development process and quality. Elementary metric for software flexibility such as flexible force, flexible degree, flexible capacity and flexible distance are proposed. Computing process of software flexibility based on flexible points (FXP) is represented in this study. But study failed to provide a formal proof for the evaluation of flexible force and study need further improve function point method to suit the measurement of flexible distance.

In the study done by Amnon et al. (2006) to measure software flexibility in precise terms, author introduced the notion of evolution complexity and demonstrated how it can be used to measure the flexibility of

(a) Programming paradigms that are Procedural Programs vs. Object Oriented
(b) Design architectural styles (Filters and Pipes, Abstract Data and Shared Data Type)
(c) Software design patterns (Abstract and Visitor Factory)

Study also demonstrated how development complexity can be used to select the flexible design policy and proposed development metric costs and recommended that flexibility can be computed as the complexity of executing particular development steps. In this work author highlighted the complexity of evolving implementations of five recognized programming paradigms, architectural styles, and design patterns, and demonstrated that evolution complexity corroborates intuitions and established observations on the flexibility of these design policies. As a result, authors provided only theoretical explanation for above stated theories In particular; the benefits from the measurements proposed are the following:

1) Development complexity can be used to corroborate and quantify informal claims on the flexibility of particular programming languages, architectural patterns and design patterns.
2) Development complexity can be used to measure flexibility with varying degrees of accuracy.
3) Development complexity can be used to select the design policy, given the class of the most likely shifts to the problem.

In conclusion, the guidelines provided by the authors are not sufficient for both architectural styles, and design patterns.

It is evident from exhaustive literature survey on flexibility that no comprehensive flexibility measurement model exists to measure flexibility of object oriented software at design phase of development life cycle.

2.6 OBJECT ORIENTED DESIGN PROPERTIES

Object oriented design overcomes the negative aspect of procedure oriented design. Object oriented design treats data as an important element in the program development and does not permit it to move freely within the system. The prominent object oriented design properties are: Inheritance, Coupling, Cohesion and Encapsulation. Object oriented software Properties that have affirmative impact on testability measurement has been recognized and consolidated chart for the same is given in Table 2.3.

Table 2.3: Object Oriented Design Properties Contributing in Testability Measurement: A Critical Look

Design Parameters ➔	Cohesion	Coupling	Encapsulation	Inheritance	Abstraction
Author/Study 🡻					
Gregor et al. (1996)		✓	✓	✓	
Bruce & Haifeng (1998)	✓	✓		✓	
Pettichord B. (2002)	✓	✓		✓	
Baudry et al. (2002)		✓	✓		✓
M Bruntik (2004)	✓			✓	✓
S. Mouchawrab (2005)	✓	✓		✓	
E Mulo (2007)		✓	✓	✓	
Khatri et al. (2011)	✓			✓	✓
Phogat et al. (2011)	✓		✓		✓

P. Malla et al. (2012)	✓	✓	✓	✓	
Nikfard et al. (2013)		✓	✓	✓	

2.7 MAPPING DESIGN PROPERTIES TO TESTABILITY FACTORS

Many experts tried to establish a relationship between object oriented design properties and testability factors. A consolidated view of the same is given in Table 2.4. After an in depth evaluation of available literature on the topic, the relation between object oriented design properties and testability factors, shown in Fig. 2.1 has been established.

Table 2.4: Mapping between Design Properties and Testability Factors based on Experts Consideration

Author/Study ↓	Testability Factors ➔ / Object Oriented Properties ↓	Controllability	Observability	Modifiability	Understandability	Traceability	Flexibility
Gregor et al. (1996)	Encapsulation	✓	✓			✓	✓
Baudry et al. (2002)		✓	✓			✓	✓
E Mulo (2007)				✓			
P. Malla et al. (2012)				✓			
Nikfard et al. (2013)		✓		✓		✓	✓

Bruce & Haifeng (1998)	Inheritance		✓		✓		✓
Pettichord B. (2002)				✓			✓
E Mulo (2007)			✓		✓		✓
Sujata et al. (2011)			✓	✓			✓
P. Malla et al. (2012)			✓	✓			
Nikfard et al. (2013)				✓	✓		✓
Bruce & et al. (1998)	Coupling	✓			✓	✓	✓
Pettichord B.(2002)				✓			
S. Mouchawrab (2005)		✓			✓	✓	✓
E Mulo (2007)		✓		✓	✓	✓	
P. Malla et al. (2012)		✓		✓	✓		✓
Bruce & et al. (1998)	Cohesion	✓			✓		✓
Pettichord B. (2002)		✓			✓		✓
Khatri et al. (2011)		✓					✓
Phogat et al. (2011)		✓			✓		

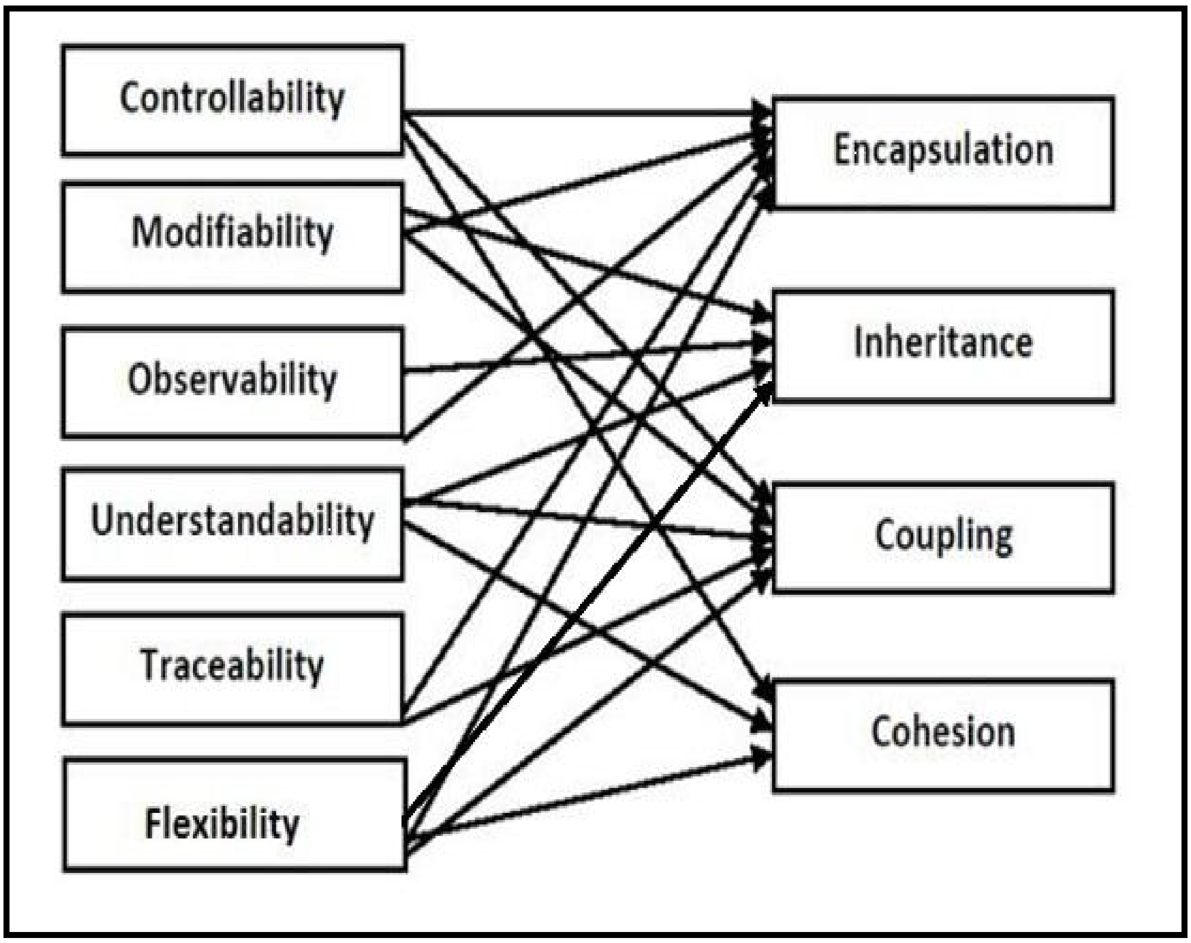

Fig. 2.1: Mapping between Design Properties and Testability Factors

2.8 QUALITY CRITERIA OF COMMONLY ACCEPTED TESTABILITY FACTORS

Criteria are the characteristics which classify the software quality factors stated by the experts Pizzi (2013), Shaheen et al. (2009) & Pettichord B. (2002). The criteria of the factors are the attributes of the software product or software production process by which the factor can be judged or characterized. The relationship between the commonly accepted testability factors and the quality criteria is listed below in Table 2.5.

Table 2.5: Commonly Accepted Testability Factors Quality Criteria

S.No.	**Factor**	**Quality Criteria**	**Mode**
1	Flexibility	▪ Structured ▪ Augment ability	Criteria of Boehm quality Mode
		▪ Generality ▪ Independence ▪ Self- documentation ▪ Modularity ▪ Software independence	Criteria of McCall quality Mode
		▪ Complexity ▪ Concision ▪ Consistency ▪ Generality ▪ Modularity ▪ Self-documentation ▪ Expandability ▪ Simplicity	Criteria of Ming-Chang Lee Mode
2	Traceability	▪ Correctness ▪ Documentation for other system Cross reference	Criteria of Boehm quality Mode
3	Understandability	▪ Consistency ▪ Structured ▪ Conciseness ▪ Self descriptiveness ▪ Legibility	Criteria of Boehm quality Mode

		▪ Consistency, ▪ Structure, ▪ Conciseness.	Criteria of Ming-Chang Lee Mode
4	Modifiability	▪ Structure ▪ Augment ability ▪ Understandability ▪ Traceability ▪ Self descriptiveness ▪ Adaptability	Criteria of Boehm quality Mode
5	Controllability	▪ Specific Design ▪ Expressiveness ▪ Understandability	Criteria of Bruce et al. Mode
6	Observability	▪ Expressiveness ▪ Specific Design ▪ Structured	Criteria of Bruce et al. Mode

There are following four major motivations for developing a list of criteria for commonly accepted testability factors:

- Criteria provide a more absolute and real definition of factors.
- Criteria common between factors help to show the interrelation among factors.
- Criteria allow assessment and review metrics to be developed with greater easiness.
- Criteria consent to identify that area of quality factors which may not be up to a predefined acceptable standard.

The analysis of the relationship between testability factors and quality criteria also suggests that modifiability and flexibility are the two testability factors that are significant for testability measurement and it encapsulates the majority quality criteria.

2.9 RELEVANT FINDINGS

The contextual findings of related work on software testability and the approaches available for its measurement may be summarized as follows:

- Testability is not an explicit focus in today's industrial software development projects. Hence, processes, guidelines and tools related to testability measurement are missing.
- Testability is primarily a design issues and it needs to be addressed at the design level. Early estimation of testability may help to improve design, and may finally produce quality software.
- In order to get reliable and correct measures of testability, it is advisable to identify the factors affecting testability. Though, getting a universally accepted set of testability factor is impossible, effort have been made to identify the key factors of testability for the same.
- Many approaches to measure software testability were proposed by the practitioners, but the empirical evaluation of these approaches is still missing.
- Testability can be estimated using the design artifacts and testability factors will also be finalized keeping in view their impact on the overall testability. However, a more systematic understanding of testability measurement is yet to be evolved.

2.10 SUMMARY

After a revision tour it comes into view that several approaches have been proposed in the available literature for measuring object oriented software testability. However, it is evident from the relevant literature that only khan et al. (2009) has validated the proposed model (MTMOOD), but this model measures software testability without including testability factors. A survey of the relevant literature also shows that maximum efforts have been put at the later stage of software development life cycle. However, the lack of testability at design phase may not be compensated during subsequent development life cycle. Researchers, practitioners and quality controllers emphasize on the need of having a systematic approach for testability measurement. They argue that testability can be measured at design phase by assessing the design level factors of testability. Hence, there is a potential to develop a systematic solution for testability measurement at design phase of development life cycle. Therefore, a comprehensive framework and related model to measure testability of object oriented software with the help of testability factors at design phase seems highly desirable and significant.

In the next chapter, we will discuss about Testability Measurement Framework.