MESUREMENT OF SOFTWARE TESTABILITY

Hari Om Sharan, Rajeev Kumar, Garima Singh, Mohammad Haroon

Department of Computer Science,
College of Engineering, Teerthanker Mahaveer University, Moradabad (U.P.), India
Email: rajeevphd@hotmail.com, sharan.hariom@gmail.com, garimasingh.0606@gmail.com

ABSTRACT: Building high quality and testable software is an essential requirement for software system. Software testability is a critical aspect during the software development life cycle. Software that is easily testable is known as testable software. Testability is an essential or distinctive aspect that is acquainted with the objective of predicting efforts needed for testing the program. Designing testability is a very important issue in software engineering. It is suggested to design software with high degree of testability. A program with high degree of testability illustrate that a selected testing criterion could be achieved with less effort and the existing faults can be revealed more easily during testing. This paper gives the concept of software testability, previously defined by The IEEE standard Glossary, our measurement for testability and complexity and also shares our thought and understanding about the testability in the object oriented system. [Hari Om Sharan, Rajeev Kumar, Garima Singh, Mohammad Haroon. Mesurement of Software Testability. Stem Cell 2011;2(1):11-19]. (ISSN 1545-4570).

Keywords: Software Testing, Software Testability, Simplicity, Complexity.

INTRODUCTION

Development in object –oriented (OO) languages and methodologies have helped in the design of better and modular software, thereby reducing the complexity and software development methodologies reduce the incidence of error, but the need to test the software remains. OO has a unique architecture, and features like inheritance and dynamic binding introduce new kind of errors. As a result, some of the issues involved in the testing of OO software are different from the issue evolved in conventional software testing. However, conventional software testing techniques are not adequate to handle all the testing issue of OO software. New tools and techniques are required or existing ones need to be adapted, to test OO software more effectively.

As software applications grow more complex and become a necessity in almost everyday activities, more emphasis has been placed on software quality and reliability. Effective testing is therefore required to achieve adequate levels of software quality and reliability. However, we are facing a dilemma: software systems are growing in complexity and testing resources are by definition limited. To maximize the impact of testing, we need to design systems so that their testability is optimal. Software testability is an external software attribute that evaluates the complexity and the effort required for software testing. Software testability has been defined and described in literature from different point of views.

Testable software is one that can be tested easily, systematically and without following any ad-hoc measures. Testable software need to possess two characteristics i.e. observability and controllability. During testing, there is a need to observe internal details of execution to ensure the correctness of processing and to diagnose errors. Observable software makes it feasible for tester to observe the internal behavior of software, to the required degree of detail.

Compared to structural development, object oriented design is a comparatively new technology. The metrics, which were useful for evaluating structural development, may perhaps not affect the design using OO language. As for example, the "Lines of Code" metric is used in structural development whereas it is not so much used in object oriented design. Very few existing metrics (so called traditional metrics) can measure object oriented design properly.

One study estimated corrective maintenance cost saving of 42% by using object oriented metrics. There are many object oriented metrics models available and several authors have proposed ways to measure object oriented design. The motivation of this thesis is to give an overview of object oriented software testability.

WHY TESTING

Testing is important for error detection and continuous software evolution:

- The potential impact of software errors on business, human life, and environment grows as software controls more and more critical functionality within technical products and business processes. Unfortunately, software development is an error prone process. Testing is the most widely used technique to detect errors.

- If user requirements change frequently, it is important for the software developer that the software system can be adapted and extended easily. New functionality added to a system should not break existing functionality. Regression testing is one technique to assure, that existing functionality remains intact after implementation changes. Without the ability to perform regression tests quickly and easily after implementation changes, the risk of undetected errors in the new software release increases. Testing is therefore an enabling factor for continuous and rapid software evolution.

Software Testing

Programmers are human beings. Human beings are prone to make errors during most of their activities, and software development is no exception. Thus the need arises for verification of the products of software development. Software testing is the practice of running a piece of software in order to verify that it works as expected.

The errors made by programmers have the potential of introducing faults in the program. Typically faults are confined to a single program statement, but more complex and distributed faults can occur too. Faults in a program have the capability of causing the program to fail. Failure happens when the program produces an output that is different from the expected output. In short, programmers make errors and introduce faults in their programs, which become prone to failure. The terms we use here are defined more thoroughly by the IEEE [3].

Software testing occurs during multiple phases of the construction of a software system. Typically the software development methodology determines both the kind of testing, and the phase(s) during which testing is done.

The following overview of software testing is based on the Software Engineering Body of Knowledge (SWEBOK) [2].First, we look at the level at which testing can take place.

Unit Testing is concerned with verifying the behavior of the smallest isolated components of the system. Typically, this kind of testing is performed by developers or maintainers and involves using knowledge of the code itself. In practice, it is often hard to test components in isolation. Components often tend to rely on others to perform their function.

Integration Testing is focused at the verification of the interactions between the components of the system. The components are typically subjected to unit testing before integration testing starts. A strategy that determines the order in which components should be combined usually follows from the architecture of the system.

System Testing occurs at the level of the system as a whole. On the one hand, the system can be validated against the non-functional requirements, such as performance, security, reliability or interactions with external systems. On the other hand, the functionality implemented by the system can be compared to its specification.

Second, testing can have several objectives. Of course, the base objective of testing is verification of the developed code; however, the reference to be used for verification can be different.

Acceptance Testing is done to verify that the system implements the customer's requirements correctly. Usually the testing is done by (future) users of the system. In addition to verifying whether or not the required functionality is present in the system, (future) users are also likely to be concerned about the user-interface and performance characteristics.

Functional Testing is done to determine if the system has correctly implemented the specification of functionality. Typically, a team separate to the development or maintenance teams would perform this task.

Reliability Evaluation is sometimes done by executing test cases obtained from a typical operational profile for the system. The rate of failure observed during such a test session can then be used to derive statistical measures of the reliability of the system.

Regression Testing is performed to make sure that a modification of a certain part of the system has not inadvertently broken other parts of the system. For example, a regression test could entail the execution of every unit test. Larger projects will likely require a more selective approach if regression testing is to remain viable.

Finally, we discuss the ways in which test cases can be selected.

White-Box Testing refers to the creation of test cases by exploiting knowledge of the implementation (i.e. the source code) of the system under test. Therefore, white-box techniques are typically applied by the same developers that wrote the code.

Several aspects of the source code can be targeted by white-box techniques. For example, possible techniques are based on the control- flow, data-flow or call behavior of the code being tested. Observing the effects of modifications made to certain parts of the code, so-called mutation analysis can also be classified as a white-box technique.

Black-box Testing is the opposite of white-box testing, in the sense that no knowledge of the implementation is used to generate test cases. Instead, black-box testing focuses on the input/output behavior of the system. This approach enables people without knowledge of the internals of a system to apply these techniques.

Many black-box techniques take the specification of the system as a starting point. The specification should provide information about the domains of inputs and outputs of the system, and describe the implemented functionality. Using this information, the tester should be able to generate input/output pairs that represent correct executions of the system. In other words, for every pair, the system should result in the specified output value when given the specified input value. Clearly, one such pair exactly represents a test case.

Software Testability

A software system's testability is defined by the ISO model [10] as "attributes of software that bear on the effort needed to validate the software product." In other words, the testability of a software system is indicative of the amount of effort needed to test the system.

What software testability is?

Software testability is an external software attribute that evaluates the complexity and the effort required for software testing. Software testability has been defined and described in literature from different point of views. The IEEE Standard Glossary defines testability as the degree to which a system or component facilitates the establishment of test criteria and performance of tests to determine whether those criteria have been met [3].

Testable software is one that can be tested easily, systematically and externally at the user interface level without any ad-hoc measure [4] [5].Testability is an important attribute to the maintainability of software. Testable software is easy and less costly to maintain and testability represents an important software quality characteristics. Testable software need to possess two characteristics i.e. observability and controllability.

Observability: In the process of testing, there is a need to observe the internal details of software execution, to ascertain correctness of processing and to diagnose errors discovered during this process. Observable software makes it feasible for the tester to observe the internal behavior of the software, to the required degree of details.

Controllability: During software testing, some conditions like disk full, network link failure etc. are difficult to test. Controllable software makes it possible to initialize the software to desired states, prior to the execution of various tests.

Other Definitions:

1. The relative ease and expense of revealing software faults [6].
2. A set of attributes that bear on the effort needed for validating the modified software [7].

A software system is testable if 1) its components can be tested separately, 2) test cases can be identified in a systematic manner and repeated, and 3) the test result can be observed [8].

Importance of Software Testability

Testability is important for software *testers and programmers* because it helps them to keep the test effort under control. Additionally it is relevant to customers *as* well; customers benefit from higher product quality and faster fixing of errors occurring at the customer site when testability features like built-in-tests, automatic failure reports, and built-in diagnostic capabilities provide better and faster information to the developers about the cause of failures which accelerates problem fixing.

Several software development and testing experts pointed out large systems the importance of testability and design for testability, especially in the context of large systems:

"During the design of new systems we do not have only to answer the question 'can we build it?' but also the question 'can we test it? 'Good testability of systems is becoming more and more important." [1].

"The absence of design for testability in large systems can greatly reduce testing effectiveness." [6]

"Design for testability, although rarely the first concern of smaller projects is of paramount importance when successfully constructing large and very large C++ systems." [11].

Factors contributing to importance of testability:

The importance of software testability for a particular software system increases with

- The size and complexity of the system,
- The risks for life and business if errors remain undetected
- The frequency of the test activities, and
- The life-time of the system (assuming that maintenance and regression testing are permanent tasks).

Fishbone of Testability

Software testability is a result of six factors:

- ➢ Characteristics of the representation
- ➢ Characteristics of the implementation
- ➢ Built in test capabilities
- ➢ The test suite (test cases and associated information)
- ➢ The test support environment
- ➢ The software process in which testing is conducted

These six factors are the spine of the testability fishbone.

The Fishbone in More Detail

Figure 1, adapted from the fish-bone figure in [3], give an overview of the facets that influence the test effort.

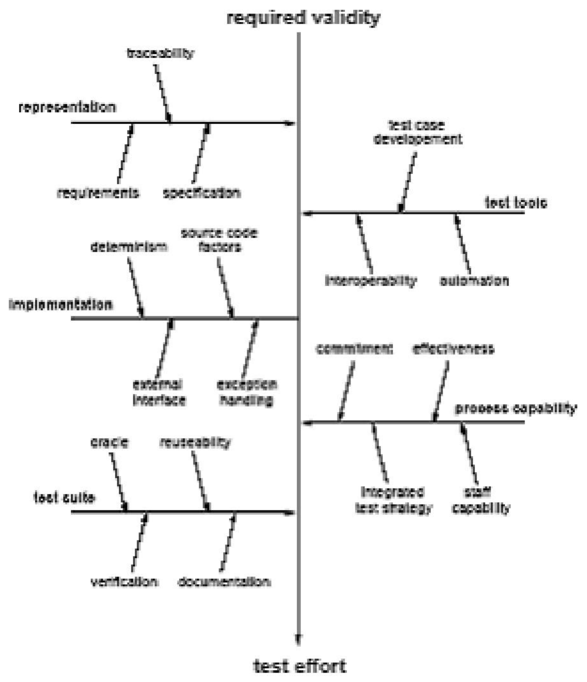

Fig 1.The Testability Fishbone [3]

A major input of the test effort picture is the degree of validity that the software is required to have. In general, software that is required to have a high degree of validity will need to be tested thoroughly before it can be claimed the requirement is met.

For some software development projects the required degree of validity may be known explicitly, while for most others the software will simply be expected to `work'. For example, safety-critical systems are often required to meet very strict validity requirements; maximally allowable failure rates are typically stated explicitly. On the other hand, a word processor application will likely not be required to meet the same degree of validity.

Let's assume that a project intends to verify the validity of the software by means of testing. If the required degree of validity is specified, the goal of testing is clear; to evaluate whether or not the software meets the specified validity requirement. It will depend on the other aspects of the project how much effort will be required to complete the testing.

If the required degree of validity is not specified, the project will need to agree on some kind of testing criterion that indicates whether adequate testing has been performed. In the context of white box testing, such a criterion is typically called a code coverage criterion, because it indicates the extent to which a certain aspect of the code has been `covered' by testing.

In practice, the moment that testing is complete will typically is determined by the amount of effort a project is capable of spending on testing. The `spine' of Figure 1 would thus start at `available test effort', and point upwards to `resulting validity'.

What is Software Testability Measurement?

Generally speaking, *software testability measurement* refers to the activities and methods that study, analyze, and measure software testability during a software product life cycle. In the past, there were a number of research efforts addressing software testability measurement. Their focus was on how to measure software testability at the beginning of a software test phase. Once software is implemented, it is necessary to make an assessment to decide which software components are likely to be more difficult and time-consuming in testing due to their poor component testability. If such a measure could be applied at the beginning of a software testing phase, much more effective testing resources allocation and prioritizing could be possible.

As we understand, the objective of software testing is to confirm that the given software product meets the specified requirements by validating the function and nonfunctional requirements to uncover as many program problems and errors as possible during a software test process. 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.

How to Measure Software Testability

In the past few years, a number of methods have been proposed to measure and analyze the testability of software [16, 17, 18]. They can be classified into the following groups:

- ✓ Program-based measurement methods for software testability [17];
- ✓ Model-based measurement methods for software testability [17, 18];
- ✓ Dependability assessment methods for software testability [16].

Program Based Testability Measurement

Since a fault can lie anywhere in a program, all places in the source code are taken into consideration while estimating the program testability. J.-C. Lin et al. [17] proposed a program-based method to measure software testability by considering the single faults in a program. The faults are limited to single faults and are limited to faults of arithmetic expressions and predicates.

- **Arithmetic Expressions:** Limited to single changes to a location. It is similar to mutations in mutation testing;
- **Assignment Predicates:** An incorrect variable/constant substitution, for example, a variable substituted incorrectly for a constant, a constant substituted incorrectly for variable, or a wrong operator;
- **Boolean Predicates***:* A wrong variable/constant substitution, wrong equality/ inequality operator substitution, or exchanging operator *and* with operator *or*.

The basic idea of this approach is similar to software mutation testing. To check software testability at a location, a single fault is instrumented into the program at this location. The newly instrumented program is compiled and executed with an assumed input distribution. Then, three basic techniques (execution, infection, and propagation estimation methods) are used to compute the probability of failure that would occur when that location has a fault.

Model Based Testability Measurement

Another measurement approach of software testability is proposed based on a well-defined model: such as a data flow model [17]. This approach consists of three steps:

- ***Step #1:*** Normalizing a program before the testability measurement using a systematic tool. Normalizing a program can make the measurements of testability more precise and reasonable. A program, after being normalized, must have the same semantics as the original one. This is done mechanically. Two types of normalization are performed here. They are structure normalization and block normalization. In the structure normalization, the program’s control flow structure is reconstructed to make it regular to facilitate analyzing and property measuring.
- ***Step #2*****:** Identifying the testable elements of the targeted program based on its normalized data flow model. The elements include the number of noncomment lines, nodes, edges, p-uses, defs, uses, d-u paths (pairs), and dominating paths.

- ***Step #3***: Measuring the program testability based on data flow testing criteria. These data-flow testing criteria include: ALL-NODES, ALL-EDGES, ALL-P-USES, ALL-DEFS, ALL-USES, ALL-DU-PAIRS, and ALL-DOMINATING PATH.

Though there is no correlation between the measurements and the number of faults, this approach can be used to check how easily software modules can be tested

C. Robach and Y. Le Traon [18] also used the data-flow model to measure program testability. Unlike the previous approach, their method is developed for co-designed systems.

Dependency Based Testability Measurement

Clearly, the two previous approaches need program source code and/or a program-based model to support software testability measurement. A. Bertolino and L. Strigni [16] proposed a black-box approach, where the software testability measurement is performed based on the dependency relationships between program inputs and outputs. The basic idea is to perform an oracle in a manual (or systematic) mode to decide whether a given program behave correctly on a given test. The oracle decides the test outcome by analyzing the behavior of the program against its specification. In particular, an input/ output (I/O) oracle only observes the input and the output of each test, and looks for failures. A program is correct with respect to its specification if it is correct on every input setting; otherwise the program is faulty. If the program generates an incorrect output, then the test has failed. If the oracle output is approved, then the test is successful.

REPRESENTATION:

Ideally there is more to a software system than its source code. According to various industry standards, documentation should cover the requirements the software needs to implement and the specification of the chosen solution. The quality of these documents has its bearing on the test effort.

Requirements capture the expectations of the customer, and thus are a crucial source of test cases that determine whether the implementation is correct and complete. From a testing viewpoint, good requirements are unambiguous and quantifiable.

A specification details the architecture and design of the solution that was selected to implement the requirements. Complete and current specifications describe the intended behavior of the implementation. Knowing the intended behavior is valuable if one wants to derive test cases that validate the implementation.

The separation of concerns inherent in modern software documentation raises the issue of traceability. A software system and its documentation are traceable if the relations between the components of the requirements and those of the specification, and those of the specification and implementation, are clear. In other words, it should be easy to point to the components involved in solving a certain requirement. Vice versa, it should be clear which requirement a certain component implements.

A non-traceable software system cannot be effectively tested, since relations between required, intended and current behaviors of the system cannot easily be identified.

Test Suite:

Aspects of the test suite itself also determine the effort required to test.

First, test cases should be created to allow for automated execution. It should be possible to compare observed output values to expect output values in an automated way, preferably by employing a mechanism called a test oracle. A test oracle is a simple abstraction of the mapping from valid input values to correct output values.

Second, reusing test suites for different revisions and configurations of the system under test must be possible. Test suites should thus be subject to configuration management along with the software itself.

Third, test cases that contain errors are as harmful as buggy code. If they are to be of any use, test suites had better be subject to a verification process of their own. Finally, test suites need documentation detailing the implemented tests, a test plan, test results of previous test runs and reports.

Test Tools:

The presence of appropriate test tools can alleviate many problems that originate in other parts of the `fish bone' figure. For example, easy-to-use tools will demand less of the staff responsible for testing. Test case definition in the presence of graphical user interfaces is another example where tooling can significantly reduce the required effort.

Obviously, testing benefits from automation of repetitive and error-prone tasks as much as any other activity does. A good set of test tools is capable of interoperating with related tools. For example, a test runner that encounters a failed test is capable of producing a trace which can subsequently be read be debugger or profiler tools, which in turn are linked with an editor, and so forth.

Process Capability:

The organizational structure, staff and resources supporting a certain activity are typically referred to collectively as a (business) process. Properties of the testing process obviously have great influence on the effort required to perform testing. Important factors include a commitment of the larger organization to support testing, through funding, empowerment of those responsible, and provision of capable staff.

In order for the process to perform effective testing, i.e. testing the right thing, requirements and specification should be taken as a starting point.

Heuristics of Software Testability

Heuristics of software testability are as follows [9].

Controllability: The better we can control it, the more the testing can be automated and optimized.

- A scriptable interface or test harness is available.
- Software and hardware states and variables can be controlled directly by the test engineer.
- Software modules, objects, or functional layers can be tested independently.

Observability: What you see is what can be tested.

- Past system states and variables are visible or queriable (e.g., transaction logs).
- Distinct output is generated for each input.
- System states and variables are visible or queriable during execution.
- All factors affecting the output are visible.
- Incorrect output is easily identified.
- Internal errors are automatically detected and reported through self-testing mechanisms.

Availability: To test it, we have to get at it.

- The system has few bugs (bugs add analysis and reporting overhead to the test process).
- No bugs block the execution of tests.
- Product evolves in functional stages (allows simultaneous development and testing) source code is accessible

Simplicity: The simpler it is, the less there is to test.

- The design is self-consistent.
- Functional simplicity (e.g., the feature set is the minimum necessary to meet requirements)
- Structural simplicity (e.g., modules are cohesive and loosely coupled)
- Code simplicity (e.g. the code is not so convoluted that an outside inspector can't effectively review it)

Stability: The fewer the changes, the fewer the disruptions to testing.

- Changes to the software are infrequent.
- Changes to the software are controlled and communicated.
- Changes to the software do not invalidate automated tests.

Information: The more information we have, the smarter we will test.

- The design is similar to other products we already know.
- The technology on which the product is based is well understood.
- Dependencies between internal, external and shared components are well understood.
- The purpose of the software is well understood.
- The environment in which the software will be used is well understood.
- Technical documentation is accessible, accurate, well organized, specific and detailed.
- Software requirements are well understood.

Testability Measurement

Several techniques have been made for development of meaningful testability [4, 13, 14] but here we are using the testability measurement techniques of John McGregor and S. Srinivas [15].They mentioned that Testability of a method into the class depends upon the visibility component. Testability of method is

$$\acute{\eta}=\text{constant}*(\zeta)$$ Where ζ is the visibility component

Testability of the class is

$$\theta=\min(\acute{\eta})$$

The definition of the visibility component (VC) is

$$\zeta=\text{Possible Output}/\text{Possible Input}$$

Before doing implementation we are defining our input, output and constant for testability analysis work and also taking some assumption for this work.

Assumption:

1. Not consider system parameter
2. Consider only concrete class.

3. All method overloading and over ridding allow.
4. Not consider static method but treat public static void main as a starting point.
5. Not consider abstract method.

The input, output and constant for the java class will be as follows

Input:

1. All parameter into the class.
2. Parameters pass into the method signature.
3. All class method parameter of the parent class excluding system parameter.
4. All method of interface implementation.

Output:

1. The return value of the method
2. Any exception either checked or unchecked by the method
3. All implicit parameter & object attribute define in the class
4. Object reference in the method signature.

Constant

1. Final
2. Literal

Static final variable is also effectively used as a constant.

IMPLEMENTATION

Base Converter:

This program enables a user to convert from numbers of different bases to numbers of different bases. The number bases supported are decimal, binary, hexadecimal, octal, and a user defined base. This means that you can theoretically convert from any base to any base if you so choose.

This project has only one class. The testability analysis of this project is as follows:

S. No	Class No.	LOC	Testability	Complexity
1	1	589	10	15

Testability of Base Converter is = 1O

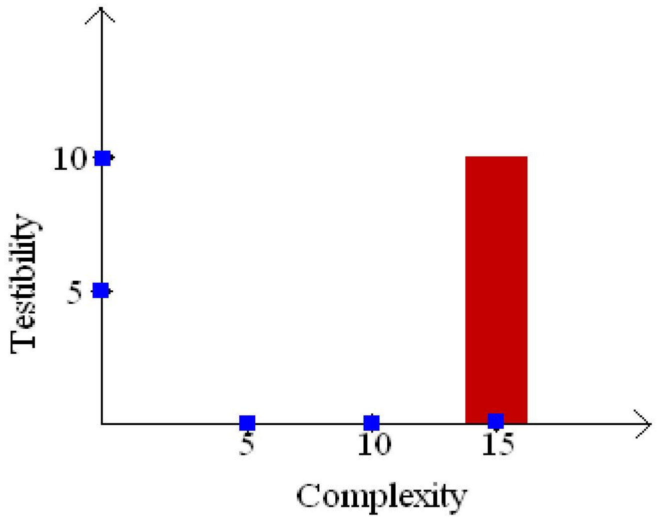

Fig 2. **Base converter testability and complexity graph**

Result:

After analysis of testability and complexity of the base converter we see that the result as shown in graph fig 2 is satisfied our working definition "Testability of a program is a degree of simplicity of the program".

❖ Here the testability is 10 which is far greater than the testability of the OES(which is maximum 3)
❖ The reason of greater testability is that, if the number of constant (In the case of base converter project we have 10 constant) in the

class will increase than the testability is automatically increased.

❖ So we can say that, constants are main factor which increase the testability of the class.

❖ In the case of OES no constant is available so the testability is varies between 1-3.

Benefits of Software Testability

There are a lot of other characteristics of design that are related to testability. In particular the lists below are benefits to testability [10].

- Understandability
- Modifiability
- Availability
- Flexibility
- Maintainability
- Reliability
- Usability
- Changeability
- Fault Tolerance

References:

(1) Martin Pol, Tim Koomen, and Andreas Spillner. Management und Optimierung des Testprozesses: Praktischer Leitfaden für erfolgre- iches Software-Testen mit TPI und Tmap . dpunkt.verlag, April 2000 ISBN 3-932588-65-7.A. Bertolino. Software testing. In The Guide to the Software Engineering Body of Knowledge, chapter 5. IEEE Computer Society, 2001. Public draft version 1.00, available at http://www.swebok.org.

(2) "IEEE standard Glossary of Software Engineering Terminology,"ANSI/IEEE Standard 610-12-1990, IEEE Press, New York, 1990.

(3) R. S. Freedman, "Testability of Software Components", IEEE Transactions on Software Engineering, vol. 17, No. 6, June 1991, pp. 553-563.

(4) S. C. Gupta, M. K. Sinha: "Improving Software Testability by Observability and Controllability Measures", 13th World Computer Congress, IFIP, vol. 1, 1994, pp. 147-154.

(5) Binder, R.V., Design for Testability with Object-Oriented Systems. Communications of the ACM, 1994. 37(9): p. 87-101.

(6) ISO/IEC 9126-2. Software product quality-external metrics.

(7) Bernd kahlbrandt.SoftwareEngineering: Objektorientierte Software-Entwicklung mit der Unified modeling language.springer,1998.

(8) **James Bach**, "Heuristics of Software Testability", April 2003.

(9) Appendix D of Stefan Jungmayr's thesis **Improving testability of object-oriented systems.**

(10) John Lakos. Large -scale C++ software design. Addision Welsey, 1996, ISBN 0201633620.

(11) B. van Zeist, P. Hendriks, R. Paulussen, and J. Trienekens. Quality of Software Products. Software Engineering Research Center, Utrecht, the Netherlands, 1996.

(12) J. Voas, PIE: A dynamic failure-based technique, IEEE Transactions on Software Engineering 18 (8) (1992) 717–727.

(13) J. Voas, K. Miller, Software testability: The new verification, IEEE Software 12 (1995) 17–28.

(14) J. McGregor and S. Srinivas. A measure of testing effort. In Proceedings of the Conference on Object-Oriented Technologies, pages 129{142}. USENIX Association, June 1996.

(15) Bertolino, A., and L. Strigini, "On the Use of Testability Measurement for Dependability Assessment," IEEE Trans. on Software Engineering, Vol. 22, No. 2, February 1996, pp. 97–108.

(16) Lin, J.-C., I. Ho, and S.-W. Lin, "An Estimated Method for Software Testability Measurement," Proc. of 8th International Workshop on Software Technology and Engineering Practice (STEP '97), 1997.

(17) Robach, C., and Y. Le Traon, "Testability Analysis of Co-Designed Systems," Proc. of 4th Asian Test Symposium (ATS'95), 1995.

11/20/2011