TESTABILITY OF SOFTWARE SYSTEM

Manu Phogat¹, Dharmender Kumar²

¹ Research Scholar, DCRUST Murthal, Haryana, India. 
Kunjian4181@gmail.com

² Associate Professor, CSE Deptt, DCRUST Murthal, Haryana, India. 
Dharmindia24@gmail.com

Abstract
This paper investigates factors of the testability of object-oriented software systems. The starting point is given by a study of the literature to obtain both an initial model of testability and existing OO metrics related to testability. The goal of this paper is to define and evaluate a set of metrics that can be used to assess the testability of the classes of an Object Oriented System.

1. Introduction
What is it that makes code hard to test? Why is one class easier to test than another? How can I tell that I’m writing a class that will be hard to test? What contributes to a class’ testability? How can quantify this notion? Software testability is affected by many different factors, including the required validity, the process and tools used, the representation of the requirements, and so on — in the next section we will survey what has been written on this topic so far. This paper investigates testability from the perspective of unit testing, where our units consist of the classes of an object-oriented software system. Our approach is to evaluate a set of object-oriented metrics with respect to their capabilities to predict the effort needed for testing. We choose this approach because metrics are a good driver for the investigation of aspects of software. The evaluation of metrics that are thought to have a bearing on the testing effort allows us, on the one hand, to gain insight into the factors of testability, and to obtain refined metrics on the other. Both results can subsequently be used in further studies.

1.1 Testability
The ISO defines testability as “attributes of software that bear on the effort needed to validate the software product” [2]. Binder [1] offers an analysis of the various factors that contribute to a system’s testability.

Major Factors

Testing Criterion. A major factor of the test effort picture is the degree of validity that the software is required to have. Based on the validity requirements, a project uses a testing criterion (or code coverage criterion) that specifies which parts of the software have to be tested. In effect, a testing criterion will establish a lower bound on the validity of the software, and an upper bound on the number of test cases required. A project will thus have to make a trade-off between the verification of validity on the one hand, and the required amount of testing on the other.

Documentation. There are many reasons why a software system should be accompanied by documentation of several kinds. With regards to testing, requirements and specifications are of prime importance, capturing required and specified behavior, respectively. These documents should be correct and complete, and clear links should exist between the concepts captured in the various documents.

Implementation. The implementation is the target of all testing, and thus the extent to which the implementation allows itself to be tested is a key factor of the testing effort. One important aspect of the implementation is determinism; it should be possible to consistently repeat tests. The major part of the implementation of an application consists of source code expressed in one or more programming languages. Factors of the source code that relate to the testability of the implementation, and thus the testing effort, are the topic of this paper.

Test Suite. Factors of the test suite itself also determine the effort required to test. Desirable features of test suites are correctness, automated execution and reuse of test cases. Similar to the system-under-test, 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.

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.

2. Related Work

A number of testability theories have been published in the literature. Voas et. al. [4] defines software testability as the probability that a piece of software will fail on its next execution during testing, provided it contains a fault. This fault sensitivity is obtained by multiplying the probabilities that (1) the location containing the fault is executed; (2) the fault corrupts the program’s state; and (3) the corrupted state gets propagated to the output. High fault sensitivity indicates high testability and vice versa.

Voas and Miller [4] present a different approach to fault sensitivity, in which semantic information contained in program specification and design documents is analyzed. An upper-bound on a component’s fault sensitivity is given by the amount of information loss occurring within the component. Information loss can appear in roughly two guises: Explicit information loss occurs because the values of variables local to the component may not be visible at the system level, and thus cannot be inspected during testing. Implicit information loss is a consequence of the domain/range ratio (DRR) of the component. The DRR of a component is given by the ratio of the cardinality of the input to the cardinality of the output.

McGregor et. al. [5] attempt to determine the testability of an object-oriented system. They introduce the “visibility component” measure (VC for short), which can be regarded as an adapted version of the DRR measure. The VC has been designed to be sensitive to object oriented features such as inheritance, encapsulation, collaboration and exceptions. Furthermore, a major goal of the VC is the capability to use it during the early phases of a development process. Calculation of the VC will thus require accurate and complete specification documents.

Freedman [6] proposes *“domain testability”*, based on the notions of observability and controllability as adopted in hardware testing. Observability captures the degree to which a component can be observed to generate the correct output for a given input. The notion of ‘controllability’ relates to the possibility of a component generating all values of its specified output domain. Adapting (the specification of) a component such that it becomes observable and controllable can be done by introducing extensions. Observable extensions add inputs to account for previously implicit states in the component. Controllable extensions modify the output domain such that all specified output values can be generated. Freedman proposes to measure the number of bits required to implement observable and controllable extensions to obtain an index of observability and controllability, and consequently a measure of testability.

3. Object-Oriented Technology:

A way to develop and package Software that draws heavily from common experience and the manner in which real world objects relate to each other.

Object-Oriented Systems: All programming languages, tools and methodologies that support Object-Oriented Technology. The main properties of object-oriented technology are following:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding

Benefits of object-oriented system:

The Advantage or benefits of object oriented system are following:

- The use of objects as basic modules assists the designer to model complex real-world systems (Model Complexity).
- The flexibility of object-oriented code allows a rapid response to changes in their requirements.
- The reuse of standard components reduces both the development time for new applications and the volume of code generated.
- The increased maintainability of software makes it more reliable and reduces maintenance costs.
- Improve Productivity
- Designed for change
4. 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.

4.1 Measurement of Software Testibility

In the past few years, a number of methods have been proposed to measure and analyze the testability of software [7, 8-9]. They can be classified into the following groups:

- Program-based measurement methods for software testability [7];
- Model-based measurement methods for software testability [8,9];
- Dependability assessment methods for software testability [7].

4.1.1 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. [8] 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.

4.1.2 Model Based Testability Measurement

Another measurement approach of software testability is proposed based on a well-defined model: such as a data flow model [8]. 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 non-comment 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 [9] also used the data-flow model to measure program testability. Unlike the previous approach, their method is developed for co-designed systems.
4.1.3 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 [7] 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.

5. Goal

In this paper working definition of testability is “Testability of a program is a degree of simplicity of the program”. In this work we are trying to understand the simplicity in the form of complexity. Means if system’s complexity is increase that means its simplicity decrease and the effort of testing (Testability) will increase.

5.1 Testability Measurement

Several techniques have been made for development of meaningful testability [10, 11, and 12] but here we are using the testability measurement techniques of John McGregor and S. Srinivas [13]. They mentioned that Testability of a method into the class depends upon the visibility component. Testability of method is

\[ TM = \text{constant} \times (VC) \]

Testability of the class is

\[ \theta = \min (TM) \]

The definition of the visibility component (VC) is

\[ VC = \frac{\text{Possible Input}}{\text{Possible Output}} \]

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

**Constant**
1. Final
2. Literal
3. Static final variable is also effectively used as a constant.

5.2 Complexity Measurement

Cyclomatic complexity is software metric (measurement). It was developed by Thomas J. McCabe [14] and is used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code. Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to the commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command.

The cyclomatic complexity of a flow graph is as follows

\[ M = E - N + 2P \]

Where

- \( M \) = Cyclomatic complexity
- \( E \) = Number of edges of the graph
- \( N \) = Number of nodes of the graph
- \( P \) = Number of connected components.

**Example:**
1. Vending Machine
   1. public class VendingMachine
2. { 
3. final private int COIN = 25;
4. final private int VALUE = 50;
5. private int totValue;
6. private int currValue;
7. private Dispenser d;
8. public VendingMachine()
9. { 
10. totValue = 0;
11. currValue = 0;
12. d = new Dispenser();
13. }
14. public void insert()
15. { 
16. currValue += COIN;
17. System.out.println("Current value = " + currValue );
18. }
19. public void return()
20. { 
21. if ( currValue == 0 )
22. System.err.println( "no coins to return" );
23. else 
24. { 
25. System.out.println("Take your coins");
26. currValue = 0;}
27. }
28. public void vend( int selection )
29. { 
30. int expense;
31. expense = d.dispense( currValue, selection );
32. totValue += expense;
33. currValue -= expense;
34. System.out.println( "Current value = " + currValue );
35. }
36. }

Step 1. Testability Analysis

<table>
<thead>
<tr>
<th>S. No</th>
<th>Method Name</th>
<th>VC (ζ)</th>
<th>TM (η)</th>
<th>Class Testability (θ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Vending Machine ()</td>
<td>3/3 = 1</td>
<td>2*1 = 2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>void insert()</td>
<td>3/3 = 1</td>
<td>2*1 = 2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>void return()</td>
<td>3/3 = 1</td>
<td>2*1 = 2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>void vend()</td>
<td>4/4 = 1</td>
<td>2*1 = 2</td>
<td></td>
</tr>
</tbody>
</table>

Step 2. Cyclometric Complexity

Control graph of vending machine

The cyclomatic complexity of a control flow graph is as follows

In control graph of vending machine the number of edges are 14 and number of nodes are 12, putting these values in the formula we will get the cyclomatic complexity 4.

\[ M = E - N + 2P \]

6. Conclusion

Software testability is an important factor during the software development life cycle. This paper gives the view that testability is the degree of the simplicity of the program and it will increase as the complexity of the program will increases and its complexity will depends on whole software development life cycle. It also emphasis is on the measurement of testability, its measuring technique and related work of the testability and in last we presents our approach of measuring testability with using some example.

After the study our conclusion is that testability is a quality factor that attempts to predict how much effort will be required for software testing and to estimate the difficulty of causing a fault to result in a failure (called revealing the fault).

7. References