A Preliminary Testability Model for Object-Oriented Software

Bruce W. N. Lo and Haifeng Shi
School of Multimedia and Information Technology
Southern Cross University, Lismore, NSW Australia 2480
blo@scu.edu.au

Abstract

In software quality assurance, two approaches have been used to improve the effectiveness of fault detection. One attempts to find more efficient testing strategies; the other tries to use the design characteristics of the software itself to increase the probability of a fault revealing itself if it does exist. The latter which combines the best feature of fault prevention and fault detection, is also known as the testability approach to software quality assurance. This paper examines the factors that affect software testability in object-oriented software and proposes a preliminary framework for the evaluation of software testability metrics. The ultimate aim is to formulate a set of guidelines in object-oriented design to improve software quality by increasing their testability.

1. Introduction

In software industry where competing demands often outstrip the ability of the industry to supply them, software quality assurance has sometimes taken second place. As software grows more complex and begins increasingly to replace human decision-makers in every facet of our society, software quality and reliability requires careful attention. When human lives and huge fortunes are dependent on automated systems software quality assurance can no longer be treated lightly.

Traditionally, verification and validation (V&V) is the last defense against software failures caused by faulty software development. But with complex software systems, complete reliance on software testing often leads to unacceptably high costs for V&V. Other means must be found to address the efficiency problem of software testing.

Two approaches to software quality assurance have been used: fault prevention and fault detection. Software fault prevention aims to apply quality assurance at every stage of the systems development process, particularly the early analysis and design stages, to prevent faults from occurring. Naturally fault prevention is the better approach, because if a fault is prevented, there is no need to spend effort to correct it. It also reduces the likelihood of new faults being introduced during the correction process. But total fault prevention is not achievable [1]. The software development process is still essentially a human activity, often conducted in a team situation. It is not possible to guarantee flawless performance in complex applications environments. Therefore in addition to prevention, software fault detection methods are needed to locate and correct the remaining (inadvertent) faults. But complete testing of software is just as not tenable as total fault prevention because exhaustive testing will consume enormous amount of resource even for a medium size program.

Thus the aim of effective software testing is to reduce testing cost in a reliability-driven process and to increase the reliability of software in a resource-limited process. There are two ways to achieve this goal: devising more effective software testing strategies and designing the software in such a way that a fault is more likely to manifest itself if the software is faulty. The latter is the *testability* approach to software development.

IEEE Standard Glossary of Software Engineering Terminology defines testability as “the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met” [2].

In this paper, we use the definition proposed by Voas & Miller [3], which focuses on the design of software to increase “the probability that it will fail on its next execution during testing if the software includes a fault”. Thus testability is the ease with which a program reveals its potential faults. The higher the testability of a program, the easier potential faults in the program will reveal themselves; the lower the testability, the more likely the faults will be hidden from detection.

The testability approach combines the best aspects of the fault prevention and fault detection methods. On the one hand, it requires careful design of the software, thus improving fault prevention; on the other, it increases the probability that faults will reveal themselves, thus making software fault detection easier and less costly.

A number of researchers [4] [5] [6] [7] [8] [9] had written about the testability approach, notably among them are the series by Voas & Millier [10] [11] [12] [13]. These researchers concentrated their study of testability in the context of conventional structured design. However, there much work yet to be done. Firstly, a more systematic understanding of the theoretical basis of the testability approach is needed. Secondly, as a quality assurance technique the testability approach will need to be examined in the context of object-oriented (OO) software development. Although OO technology has now been widely accepted by the software industry, few research studies have been devoted to explore the concepts of testability in OO systems [14] [15].

Inherent in the testability approach is the need to consider software design characteristics so that the software has the ability to reveal faults if such faults exist. Thus testability must take into consideration the specific design features of the OO software. Object-orientation is characterised by object classes, inheritance, encapsulation, and polymorphism [16] [17]. While these are the features that enhance the usefulness of OO software, they also present some unique challenges to software fault detection.

The purpose of this paper is to examine factors that affect the testability of OO software through a study of the unique characteristics of OO systems and propose a framework to measure some OO testability metrics.

The remaining part of this article is divided into three sections. Section 2 introduces the theoretical basis for software testability, based on the *fault-failure* model. Section 3 investigates the factors in OO software that affect the testability of the system. These factors fall into three groups: structure factors, communication factors and inheritance factors. Each of them contributes to the software testability in a unique way. Section 4 attempts to summarise the testability approach in the context of OO systems and discusses possible future research directions.

2. How does software fault manifest itself?

Testability measures the probability that potential faults in a software will reveal themselves under software testing. Software faults and software failures are not the same. Faults are static while failures are the results of dynamic execution. What we observe is failure. Faults are not directly observable until they are located in the code or design specification. Not all faults necessarily lead to software failures. To understand the process of fault manifestation, a number of questions must be considered. Under what conditions does a fault in a program reveal itself? How does a particular input set trigger a fault? How does an erroneous internal data state manifest itself in the output space? To understand the mechanism of software testability, we introduce the fault-failure model of software testing [1] which is illustrated in Figure 1.

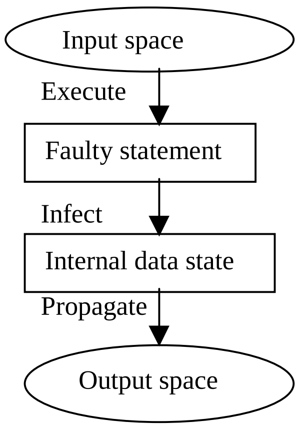

Figure 1: Fault-failure model

The fault-failure model suggests that it is necessary to satisfy three conditions for a fault to reveal itself. Firstly, the code segment in which a fault exists must be executed. Secondly, at least one data state inside the program module must be infected by the faulty segment. Finally, the infected data state must be propagated to the program's output, for it to be observed. This *execute-infect-propagate* action chain is essential for the observation or detection of software faults.

If a segment in a program module contains a fault but is not executed, the fault will not affect the program output, so testing procedures will not detect its presence. If that segment is rarely executed in the module, the chance for it to reveal the potential faults will be low, resulting in a low testability for this program segment.

However, "execution" of the faulty segment alone is not sufficient for fault detection. Consider the following faulty program statement

*X=(-B-SQR(B*B-2*A*C))/2/A*

which is designed to solved the quadratic equation $ax^2 + bx + c = 0$. But if we use this statement to solve an equation with $c=0$, we will still get the right answer, since the fault inside the executed statement does not infect the internal data state of X in this case.

If for a given program segment, only a small number of cases among the set of test cases in the input space, will infect the internal data state, the testability of this part of the program will be low. But if almost every time the faulty segment is executed an internal data-state is infected, the testability of this part of the program will be high. Of course, data state infection is possible only if the faulty segment is executed in the first place. Therefore the two events are stochastically dependent.

There is a third condition to be satisfied before a fault can be observed externally. Does the incorrect internal data state propagates to the output space? The likelihood of propagation also contributes to software testability. For instance, if the quadratic equation example above contains the faulty segment (where *Z* is the output variable):

```
X1=-B-SQR(B*B-2*A*C)/2/A
X2=-B+SQR(B*B-4*A*C)/2/A
IF X2>Y THEN Z=X2 ELSE Z=X1
```

Suppose for a particular set of input data, C does not equal to 0. The statement that contains a fault will be executed and the internal data state X1 will be infected by this fault. But if X2>Y, the output variable Z will not be affected by infected data state. The incorrect result cannot be detected in the program's output. Therefore in addition to have a faulty segment executed and one of the internal data states affected, the incorrect data state must also be propagated to the output space.

In the fault-failure model, we expect a fault to manifest itself, only if all three conditions: *execute, infect,* and *propagate* are satisfied. During program testing, the more frequently a code segment containing a fault is executed, the more likely the executed fault will infect the internal data state, and the easier the infected data state propagates itself to the output space, the higher will be the testability of this code segment.

Since the testability is the likelihood of a fault revealing itself, it is a probability function. Its value depends on three stochastically dependent events. Therefore we may write the testability, $T(s)$, of a software segment as the product of three probabilities:

$$T(s) = R_e(s) \bullet R_i(s) \bullet R_p(s)$$

$R_e(s)$ is the probability that a certain faulty program segment will be executed. If for a particular testing strategy, N is the total number of test cases selected, among which n will cause the specific segment to execute, then $R_e(s)=n/N$. We shall call $R_e(s)$ the *execution rate* of the code segment under consideration.

$R_i(s)$ is the probability of an internal data state being infected given that a faulty program segment has been executed. If out of n times that a particular faulty program segment is executed, the internal data state is infected m times, then $R_i(s)=m/n$.

$R_p(s)$ is the probability of incorrect result(s) being propagated to the output space given that a certain data state has been infected. If out of the m times that internal states are infected, t of them will propagate to the program output, then $R_p(s)=t/m$.

However, the parameters, N, n, m and t are neither directly accessible nor reflecting the internal design of a program from which the concept of testability is derived. Therefore the challenge is to rewrite the testability in a form so that it is susceptible to analysis. We choose to express $T(s)$ as the product of two probabilities:

$$T(s) = R_e(s) \bullet R_{fp}(s)$$

$R_{fp}(s)$ is the probability of propagating the incorrect result(s) to the output space after the statement is executed. We shall call it the *fault propagation rate* of the code segment under consideration.

This formula reflects the two key features of execution and propagation, which determine the testability of a program segment. Its elements are accessible through the method of sensitive analysis and reflect the internal structure of the software. This equation, which represents testability as a product of execution rate and propagation rate, will form the basis in our study to explore the testability factors in OO software development.

3. What factors affect testability in object-oriented software systems?

A close examination of the characteristics of OO software systems shows that factors which contribute to software testability may be grouped into three classes: structure factors, communication factors, and inheritance factors.

Structure factors reflect the testability of coding structure of the program segment; communication factors measure the testability of OO software by the degree of coupling among classes and objects; and inheritance factors contribute to the testability via the inheritance features of OO design. Each of these three groups of factors will now be examined individually. Our aim is to propose a set of testability metrics to complement other OO software metrics [18] [19].

3.1 Structure factors

The structure of an OO program is characterised by the structure of its classes and methods. The structure factors that affect the testability of an OO program are reflected by the structural characteristics of its classes and methods. Focusing on at the classes structure, we can further differentiate three subfactors which contribute to testability. These are: internal testability of a method (reflecting the internal structure of a method), number of methods in a class, and cohesion among methods within a given class.

3.1.1 Internal testability of a method

Associated with every method inside a class, there is an internal testability. Based on the fault-failure model discussed above, this testability may be expressed as a product of the execution rate and the fault propagation rate associated with that method. We propose the following procedure to estimate their values.

(a) Execution rate of a method. The execution rate of a method is determined by the execution rate of individual statement inside the method. In order to determine the execution rate of each statement, we need to consider both the total execution number of a method and the number of times this particular statement is executed during the total execution time. Since we are concerned with the internal structure of a method, we use the number of paths (or test cases) needed in a specific white box testing strategy as the total execution number. More precisely, we define the total execution number as the minimum number of test cases to provide a total test coverage, if such coverage is possible. Similarly, the number of times a statement is executed during the total execution times is also based on this principle. Obviously such a definition is test strategy related. Our definition is based upon the control-flow-oriented test strategy. Since this is a well researched area in software testing, the number of total test cases and the number of test cases passing through a specific statement can be determined by a flowgraph analysis derived from that method.

The flowgragh model is based on Fenton-Whitty's flowgraph theory [20]. The Fenton-Whitty theory shows how a program can be represented by a flowgraph, and how the flowgraph can be decomposed into primitive flowgraphs by repeated sequencing and nesting of the primitive flowgraphs. Figure 2 provides some examples of primitive flowgraphs.

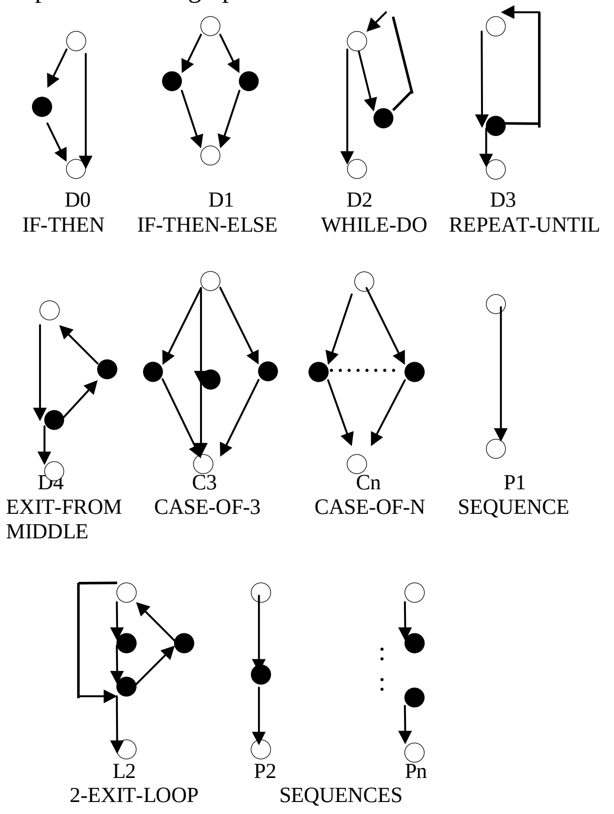

Figure 2: Primitive Flowgraphs

The decomposition of a flowgraph into primary flowgraphs is unique. We can express this decomposition as a tree where the nodes of the tree are the primitive flowgraphs and the edges represent the action of sequencing and nesting. An example of the decomposition of a flowgraph is illustrated in Figure 3.

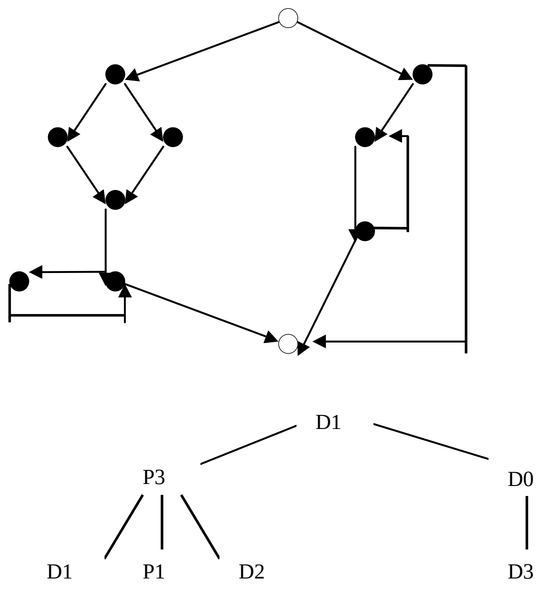

Figure 3: A flowgraph and its decomposition

Suppose we use the branch testing strategy. The decomposition tree enables us to determine the number of paths (or test cases) needed to cover all edges in a primitive flowgraph and also to determine the number of paths needed to cover all edges of the entire tree. From these we can work out the execution rate of a primitive flowgraph associated with a particular program segment. Space limitation does not allow a full description of the computational procedure. For further details, readers are referred to the reference [21]. Below is a brief summary of the approach.

Suppose flowgraph *M*, of a method is decomposed into *k* primitive flowgraphs P_i, $i=1\ldots k$, then

$$M = P_1 \cup P_2 \cup P_3 \ldots\ldots \cup P_k$$

The execution rate, $e(P_i)$, of each primitive flowgraph, P_i associated with a program segment within a method, is given by the ratio

$$e(P_i) = n(P_i) / n(M)$$

where $n(P_i)$ is the number of paths (or test cases) needed to cover all edges in the primitive flowgraph P_i; and $n(M)$ is the number of paths needed to cover all edges in the entire flowgraph *M* associated with the method.

The execution rate $E(M)$ of the method can be defined as the average of the execution rates of all the individual primitive flowgraphs in *M*. Thus

$$E(M) = \tfrac{1}{k}\sum_{i=1}^{k} e(p_i)$$

where k is the number of primitive flowgraphs in method M. Therefore to find the execution rate of a method, we would first do a decomposition of flowgraph associated with that method into its primitive flowgraphs and then compute an average of the execution rates of all the primitives in that flowgraph.

(b). Propagation rate of a method. If every possible output of a method can be traced back to a unique input set, every fault will be propagated to the output space, resulting in a software failure. Under such circumstances, the propagation rate of the method is equal to 1. If the number of possible output decreases but the number of input is constant, the propagation rate will decrease. We propose to use the *domain-to-range ratio* (DRR) as a measure of the propagation rate of a method. The domain-to-range ratio, introduced by Voas and Miller [3] to measure the implicit information loss of a program function, is defined as the ratio of the cardinality of possible inputs to the cardinality of possible outputs.

The value of DRR is always greater or equal to 1. If DRR=1, then there is a one-to-one relationship between the input and the output. The method can retain all information from the input stage to the output stage. If DRR > 1, then there are at least some of information in its internal states cannot be transmitted to its output. This may be regarded as a collapse of some internal states, which results in a loss of information inside a method. Generally speaking as DRR increases the fault propagation rate of a method decreases.

Information loss can occur in at least two different ways. Firstly, the number of live variables in an executing method decreases as execution proceeds until finally only the surviving live variables are transmitted to the output variables. When an erroneous live variable become dead, unless its “incorrectness” is transfer into another live variable prior to becoming dead, this error will be invisible during software testing. Secondly, information loss also can occur when two different program states are presented to a particular subfunction in a program and that subfunction produces the same internal state in both cases. For example the function: $f(x) = x^2$ produces the same internal state for both inputs x = 5 and x = -5. If the negative value signals a problem with the software, execution of *f(x)* will erase that information in subsequent output. The incorrect data states will not be visible in the output. Consequently the probability of observing a failure during testing is reduced.

If information loss occurs somewhere inside a method, the probability of undetected faults in that method increases. Information loss in this context is related to testability. Therefore, we may define the propagation rate of a method M to be

$$P(M) = \frac{1}{DRR}$$

We may now combine the execution rate and the propagation rate according to the fault-failure model, to obtain the testability of method *M:*

3.1.2 Number of methods in a class

Having considered the contribution of the method

$$t(M) = E(M)P(M) = \frac{1}{k(DRR)}\sum_{i=1}^{1} e(p_i)$$

testability in a class, we turn now to the second factor, viz. the number of methods in a class. We assume that every method in a class contributes equally to the execution rate of a class. If there are M methods in a class then there is an *1/M* chance of a test case passing through a specific method. Therefore, the number of methods in a class is inversely proportional to the testability of a class. The larger the number of methods in a class, the lower the testability of the class. If the testability of a method M_i is $t(M_i)$, we may use the average value of all testabilities associated with the methods in a class, viz.

$$\frac{1}{M}\sum_{i=1}^{M} t(M_i)$$

as the contribution to the testability of a class by the testability of methods in that class, and define:

$$t(C) = \frac{1}{M^2}\sum_{i=1}^{m} t(M_i)$$

as the testability of the entire class. This expression takes into consideration both the contributions of the internal testability of the methods and the number of methods in that class.

3.1.3 Cohesion among methods

The cohesion among methods within a class is an indication of how closely those methods are related to local instance variables in that class. It measures the degree of similarity of methods within a class. The greater the number of similar methods, the more cohesive is the class.

Cohesiveness among methods within a class is desirable, since it usually means decreasing the number of methods, the size of the class and hence the testing efforts. Low cohesion usually means poor design or poor organization of a class and thereby increasing the complexity of a class and the likelihood of errors. Therefore the more cohesive are the methods within a class, the higher the testability of that class. The lack of cohesion in methods results in lowering the testability of the class.

To measure the cohesion among methods in a class, we consider a class C with M methods denoted by M_i, $i=1….M$. Let I_i be the set of instance variables used by method M_i. There are M such sets I_1, I_2, …., I_M. Among these *M* sets, some of them will have common elements while others will be disjoint sets.

Let us consider the set of all ordered pairs of I_i's and partition them into two mutually exclusive classes *P* and *Q*, where *P* denotes the set of ordered pairs of I_i's that are disjoint and *Q* denotes the set of ordered pairs of I_i's that have common elements. Thus

$P = \{(I_i, I_j) \mid I_i \bullet I_j = \phi\}$

$Q = \{(I_i, I_j) \mid I_i \bullet I_j \neq \phi\}$.

Put in another word, *P* is the set of ordered pairs (I_i, I_j), whose intersection is null, while *Q* is the set of ordered pairs (I_i, I_j), whose intersection is nonempty. The larger the cardinality of *P*, the more cohesive are the methods, while the larger the cardinality of *Q*, the less cohesive are the methods. To measure the cohesion among methods we use the following metric

$Ch=(|PUQ| - |P|) / |PUQ|$

If all order pairs are disjoint, then $|PUQ| = |P|$ and $Ch = 0$, indicating there is a total lack of cohesion. On the other hand if all order pairs have common elements, then $|P| = 0$ and $Ch = 1$, indicating a high degree of cohesion.

Including this factor into the definition of the testability of a class, we obtain,

$$T(C) = \frac{|P \cup Q| - |P|}{|P \cup Q|} t(C)$$

3.2 Communication factors

The interaction between components within an OO program is actually the interaction or coupling between classes. There are two types of interactions between classes: through class inheritance and through direct methods calling. Here we shall only consider the contribution of the second to the communication factor, i.e. the coupling between classes through method calling. The other factors relating to the class inheritance factors will be discussed later.

Coupling by inheritance among classes is encouraged since it utilises the OO features of inheritance and encapsulation and enhances testability. However, coupling between classes through message passing is detrimental to modular design and prevents reuse, since this kind of coupling tends to complicate the relationship between classes, making the refinement and maintenance of the coupled classes difficult. It requires more effort in software testing because classes are entangled with each other. The larger number of classes a given class is coupled to, the greater the effort is needed to find software faults among the entangled classes if faults do exist. As a result the testability of the class is lower.

We shall define the coupling number of a class as the number of other classes to which it is coupled. A class is coupled to another when methods declared in one class use methods or instance variables defined by the other.

From the viewpoint of fault-failure model, coupling between classes through message passing will increase the number of test cases needed to cover a certain testing strategy and therefore decrease the execution rate of a class. Suppose the coupling number of a class *C* is *m*. If we test class *C* we must go through all *m* classes with which it couples. Thus *m* times more test cases are needed to cover the coupled class *C* compared to when there is no coupling. The coupling number is therefore inversely related to the testability factor of a class. The testability of

$$T(C) = \frac{1}{m} t(C)$$

a class *C* after considering the effect of coupling with other classes is given by:

where *m* is the coupling number of class *C* and *t(C)* is the testability of class before considering its coupling factor.

3.3 Inheritance factors

Three inheritance factors that can affect the testability of an OO program are: depth in the inheritance tree, the number of children, and the number of disjoint inheritance trees.

3.3.1 Depth in the inheritance tree

The inheritance structure of classes in a program can be represented as an inheritance tree. The superclasses are the root or nodes closer to the root than the one under consideration, while the subclasses are either the intermediate node of the tree or the leaves of the tree.

The depth of a class in an inheritance tree affects the testability of the class. Usually, the deeper the class in an inheritance tree, the greater the number of methods it will inherit making it more complex to test and maintain. If a class is near the root of an inheritance tree, there are more chances it will be tested since every time a test case goes through its offspring, it will also go through the class itself. Therefore the nearer the class to the root of an inheritance tree, the higher the testability of a class while the deeper the class is in an inheritance tree, the lower its testability.

We define the depth of a class in an inheritance tree to be *(m-1)/n*, where *n* is the length of paths from the furthest leaf to the root and *m* is the length of paths from the node representing this class to the root of the tree. In cases involving multiple inheritance, *m* will be the maximum length from the node of the class to the root of the tree.

In the fault-failure model, the testability *T(C)* of a class *C* with *(m-1)/n* depth of inheritance is

$$T(C) = (1 - \frac{m-1}{n})t(C)$$

where *t(C)* is the testability of the class before its inheritance factor of depth in inheritance tree is considered.

3.3.2 Number of children

The number of children of a class is defined as the number of immediate subclasses subordinated to the class in the class hierarchy. From the view of inheritance trees, the number of children of a class *C* is the number of nodes which are directly below and linking to node *C*.

Usually the number of children of a class reflects its reuse. The greater the number of children, the greater the reuse. From the viewpoint of the fault-failure model, all test cases that go through the children of a class will also go through this class. Therefore the greater number of children a class possesses, the higher the percentage of test cases will go through this class, rendering a higher execution rate for this class resulting in higher testability.

Suppose a class *C* has *m* children. Generally speaking, a test case going through its children has the opportunity of going through the class itself. The execution rate of a class with children should be higher than one with none. Therefore, if the testability of class *C* is *t(C)* before its number of children is considered, its testability now will be raised to

$$T(C) = (b - \frac{b-1}{m+1})t(C)$$

where b is a fixed parameter, whose value is slightly greater than 1. We note that the second and higher generation offspring of class *C* will also contribute to its testability. But they have been taken into consideration through the testability factor of the depth of that class in the inheritance tree.

3.3.3 Number of disjoint inheritance trees

To model real world phenomena, we often found that one single inheritance tree cannot adequately represent the complex relationships needed. The subclasses cannot be regarded as the offspring of a single superclass. Several disjoint superclasses are needed, each of which is the root of a disjoint inheritance tree.

The number of disjoint inheritance trees affects the design of testing efforts. The higher the number of disjoint inheritance trees, the more complex is the program and more resources are needed for software testing and maintenance lowering the its testability.

From the viewpoint of fault-failure model, the test cases needed to cover a certain kind of testing strategy will increase and the execution rate of a class (or a component) in a program will decrease as the number of disjoint inheritance trees grows. Suppose there are *m* disjoint inheritance trees in a program. Assuming the complexities of the inheritance trees are about the same, on the average the execution rate for a class within one of the inheritance tree will be *1/m* times smaller than that of a class with only one inheritance tree. The testability of the program after the number of disjoint inheritance trees is taken into consideration is given

$$T(P)=t(P) / m$$

where *t(P)* is the testability of program before the disjoint tree has been considered.

4. Summary

The purpose of this paper is to explore how to take into consideration factors that affect testability in OO software. We divide the factors into three groups: structure factors, communication factors and inheritance factors, each of them represents one facet of the characteristics of OOD. Each affects the testability of OOD in a particular way. Some factors are common to non-OO systems, but most of them are unique to OO software development.

The testability factors presented here are all related to the testability of classes and methods. They may be classified as intra- or inter- object factors. This 2 x 2 classification gives rise to four groups of factors, to which can be add one other program-level factors. Table 1 below summarises the relationships between the testability factors and the type of characteristics in OO design.

Table 1: A preliminary taxonomy of testability factors

Types of Factors	**Testability Factors**		
Intra-method	Execution Rate	Propagation Rate	
Inter-method	Cohesion		
Intra-class	No of methods	Depth in inheritance tree	No of children
Inter-class	No of coupling		
Program	No of disjoint inheritance trees		

In summary, the testability of an OO program is the composition of all these factors from program level down to method level. Based on the fault-failure model of software testing, this paper proposes a framework for the estimation of the total testability of an OO system from the individual testability factors of its components based on the OO development paradigm.

While some of the metric measures may require further refinement, the proposed approach gives us a unified and integrated model of OOD testability. It is our intention to apply this model to several real systems to test and verify the usefulness and validity of the individual testability factor metrics and the correctness and meaningfulness of the complete model. The final aim of our research is to propose a testability approach to OO software development which may lead to higher quality software or save development/maintenance effort or both.

6. References

[1] B. Beizer, *Software Testing Techniques*, 2nd edn, Van Nostrand Reinhold, New York, USA, 1992.

[2] IEEE *Standard Glossary of Software Engineering Terminology*, ANSI/IEEEE Standard 610.12-1990, IEEE Press, New York, 1990.

[3] J.M. Voas and K.W. Miller, Software testability: The new verification, *IEEE Software*, pp. 17-27, May 1995.

[4] R. Bache & M. Mullerbury, Measures of testability as a basis for quality assurance, *Software Engineering Journal*, pp. 86-92, March 1990.

[5] R.S. Freedman, Testability of software components, *IEEE Transactions on Software Engineering*, Vol. 17, No. 6, pp. 553-564, June 1991.

[6] John Bainbridge, Defining testability metrics axiomatically, *Software Testing, Verification and Reliability*, Vol. 4, pp. 63-80, 1994.

[7] W. Howden and H. Yudong, Software testability analysis, *ACM Transactions on Software Engineering and Methodology*, Vol. 4, pp. 36-64, January 1995.

[8] S.T. Chanson, A.A.F. Loureiro, S.T. Vuong, On the design for testability of communication software, *Proceedings of the 1993 International Test Conference*, pp. 190-199, 1993.

[9] K.R. Pattipati, S. Deb, M. Dontamsetty, A. Maitra, START: system testability analysis and research tool, *IEEE AES Systems Magazine*, pp. 13-20, Jan. 1991.

[10] J.M. Voas and K.W. Miller, Improving the software development process using testability research, *IEEE Software*, pp. 114-121, 1992.

[11] Jeffrey M. Voas and Keith W. Miller, Semantic metrics for software testability, *Journal of Systems and Software*, Vol. 20, pp. 207-216, 1993.

[12] Jeffrey M. Voas, Keith W. Miller and J. E. Payne, A comparison of a dynamic software testability metrics to static cyclomatic complexity, *Software Quality Management*, pp. 431-445, 1994.

[13] J. Voas, J. Payne, and K. Miller, Designing programs that are less likely to hide faults, *Journal of System and Software*, Vol. 26, pp. 93-100, January 1993.

[14] Robert V. Binder, Design for testability in object-oriented systems, *Communications of the ACM*, Vol. 37, No. 9, pp. 87-101, Sept. 1994.

[15] R.L. McGarvey, Object-oriented test development in ABBET, *IEEE Software*, pp. 243-255, 1994.

[16] R. Fichman & C. Kemerer, Object-oriented and conventional analysis and design methodologies: comparison and critique, *IEEE Computer*, Vol. 25, pp. 20-39, 1992.

[17] Andy Carmichael, *Object development methods*, SIGS Books, Inc., New York, USA, 1994.

[18] S.R. Chidamber & C.F. Kemerer, A metrics suite for object oriented design, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476-493, June 1994.

[19] M. Lorenz & J. Kidd, *Object-oriented software metrics: a practical guide*, P T R Prentice Hall, New Jersey, USA, 1994.

[20] S. Deb, K.R. Pattipati, V. Raghavan, M. Shakeri, and R. Shrestha, Multi-signal flow graphs: A novel approach for system testability analysis and fault diagnosis, *IEEE AES Systems Magazine*, pp. 14-25, May 1995.

[21] H. Shi and B. W. N. Lo, Design factors that affect testability in object-oriented software development, *Technical Report CIS97/1*, School of Multimedia & Information Technology, Southern Cross University, 1997.