Analysis of Factors Affecting Testing in Object oriented systems

Mrs. Sujata Khatri [1] Dr. R. S. Chhillar[2] Mrs. Arti Sangwan[3]

[1] D.D.U College, University Of Delhi, Delhi, India
[2] Department of Computer Science & Application, M.D.U, Rohtak, India
[3]Maharaja Agrasen College, University Of Delhi, Delhi, India

ABSTRACT: Software testing is an important software quality assurance activity to ensure that the benefits of Object oriented programming will be realized. Testing object oriented systems is little bit challenging as complexity shifted from functions and procedures as in traditional procedural systems to the interconnections among its components. Object oriented development has presented a numerous variety of new challenges due to its features like **encapsulation**, **inheritance, polymorphism** and **dynamic binding**. Earlier the faults used to be in the software units, whereas the problem now is primarily in the way in which we connect the software. Development is writing the code, testing is finding out whether or not the code runs the way you expect it to. A major challenge to the software developers remains how to reduce the cost while improving the quality of software testing. Software testing is difficult and expensive, and testing object oriented system is even more difficult. A tester often needs to spend significant time in developing lengthy testing code to ensure that system under test is reasonably well tested. Substantial research has been carried out in object oriented analysis, design .However relatively less attention has been paid to testing of object oriented programs. This paper describes the various features of object oriented programming and how they effect testing of object oriented systems.

KEYWORDS: Encapsulation, Inheritance, Abstract Classes, Polymorphism, Coupling, Cohesion.

1. INTRODUCTION: Software testing is an important software quality assurance activity to ensure that the benefits of object oriented programming will be realized. The objective of software testing is to uncover as many errors as possible with a minimum effort and cost. A successful test should show that a program contains bugs rather than showing that the program works. Since software testing consumes 40-50 percent of the development costs, how to reduce its cost and improve its quality has always been a big challenge. OO software testing has to deal with new problems introduced by the powerful features of OO languages. Features (such as encapsulation, inheritance, polymorphism, and dynamic binding) provide visible benefits in software development by increasing software reliability, reusability, extensibility and interoperability. Software testing is necessary to realize these benefits by uncovering as many programming faults as possible at a minimum cost. However OO features introduce new testing challenges that cannot be easily addressed with the previous techniques used in traditional approaches and require definitions of new techniques, new problems that require new solutions. Therefore requirements for testing object-oriented programs differ from those for testing conventional programs. Most of the interactions that occur among program units take place through complex state interactions due to inheritance and polymorphism. . **Encapsulation** makes it difficult to understand object interactions and prepare test cases to test such interactions. **Inheritance** does not guarantee that a method that is tested to be "correct" in the context of the super class will work "correctly" in the context of the sub class. **Polymorphism** creates an attribute of an object may refer to more than one type of data, and an operation may have more than one implementation results in lack of controllability. **Dynamic** binding introduces undecidability concern making testing more difficult because the exact data type and implementation cannot be determined statically. **Abstract classes** cannot be instantiated and thus pose challenges for execution base testing. In this paper we discuss the six critical features of object oriented systems, namely- encapsulation, inheritance, polymorphism, dynamic binding, abstract classes, coupling and cohesion. Further we consider these features separately to identify the different problems each of them can introduce as far as testing is considered. . The paper is organized as follows: Section 1.1 of the paper deals with of work done in the area of problems. Section 2 of the paper describes the various factors affecting testing in object oriented systems. Section 3 of the paper describes the results and findings. Conclusion of the paper and future research direction has been given in section 4. References are listed in section 5.

1.1 Related Work: Following sections briefly summarize some of the relevant efforts made by researchers in this area. The problem of identifying various factors affecting testing in object oriented system is extremely relevant and has been tackled in several papers. Most of the papers that addressed this problem proposed one or two factors. Some work investigated inheritance related problems and some investigated polymorphism related problems .Morris[1] discussed the challenges to testing object oriented systems and discussed different level of testing techniques . Geetha, Palanisamy [2] proposed a tool for inheritance related bugs in object oriented

software. John and Kolling [3] described the testing tools within the Blue programming environment which allow object-oriented programs to be thoroughly tested without writing a single line of new code. Smith and Robson [7] explored the problems created due to inheritance property of object oriented systems. Dinesh [4] presented a report strengthening the various issues and problems associated with polymorphism. Alexander, Jeff & Andreas Stefik [6] analysed techniques for analysing and testing polymorphic relationship that occur in object oriented systems. The purpose of this paper is to explore the various factors that make testing of object oriented software more complicated.

2. Factors affecting Testing in Object oriented systems:

2.1 Encapsulation/**Data Abstraction:** "Encapsulation is a technique for enforcing information hiding where the interface and implementation of a program unit are syntactically separated". This enables the programmer to hide design decisions within the implementation and to narrow the possible interdependencies with other components by means of interface. If a programmer changes only the implementation of unit leaving the interface same then he needs to retest that unit and any units that explicitly depend on it. Therefore if we modify the super class then it is necessary to retest all its subclasses because they depend on it in the sense that they inherit its methods. To make software easy to test we need to increase testability (ease of testing). Testability of software consists of two key factors: controllability and observability." The capability to control the inputs of a given program under test is controllability" and "observability is the ability to examine intermediate outputs of program i.e. how input is processed to get output". Encapsulation is a major obstacle for both controllability and observability. Data abstraction refers to the act of representing essential features without including the background details. Also due to data abstraction there is no visibility of the insight of objects. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. This data hiding makes it difficult for the tester to check what happens inside an object during testing. For example

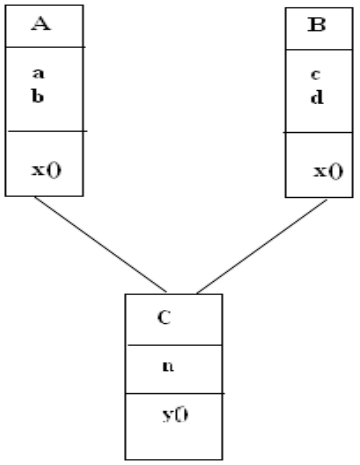
```
Class sample {
 Private:
     …
  Int a;
  Public:
 Void check sample status ( ) {
   …….
  If (a= =1)…..
       Else if (a….)
          }…..
          };
```

In this example attribute 'a' is not accessible as it is declared as private and the behavior is strongly dependent on it hence affecting controllability and observability. An encapsulated class is usually the focus of unit testing. Here the unit is a class, often a "bigger" or more complex thing than a procedural unit which may be a single function or sub-routine. Complicated by encapsulation, a unit test is more complicated and more effective in the overall system than with procedural unit tests.

2.2 Inheritance: Inheritance is one of the primary strengths of object-oriented programming. "Inheritance means properties defined for a class are inherited by its subclasses, unless it is otherwise stated". So actually it provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. For example if a need arises to include some additional features in a fully tested class, then one way is that we can modify the existing class and repeat the whole testing process again which means previous testing is redone along with testing of additional features .Under such situation inheritance helps a lot i.e. a derived class can be created by keeping the base class intact and derived class inherits features of base as well as additional features. However method that is tested to be "correct" in the context of the base class does not guaranteed that it will work "correctly" in the context of the derived class. Therefore it is precisely because of inheritance that we find problems arising with respect to testing. It also complicates inter-class testing as multiple classes are coupled through inheritance. Inheritance raises lots of issues from testing point of view; some common inheritance related problems are discussed below:

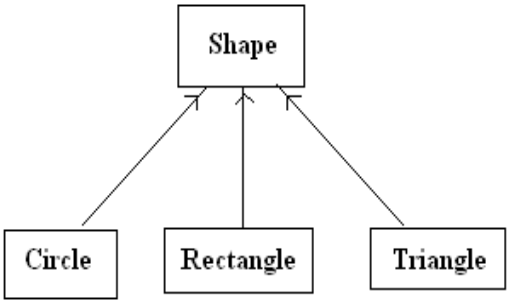
1. It is necessary to test whether a subclass specific constructor is correctly invoking constructor of the parent class. Because when both derived and base class contain constructor, the base constructor is executed first and then the then the constructor in derived class.
2. We need to know that whether we can trust features of classes we inherit from or need to retest all the features of derived class again which it inherits from the parent class. For example consider a class hierarchy in which an operation x is defined for the super class and it is inherited by a number of sub class. Each subclass uses operation x, but it is applied within the context of the private attributes and operation that have been defined for the subclass. Because the context in which operation x is used varies in subtle ways, it is necessary to test

operation x in the context of every subclass. This means testing of x in standalone fashion (i.e. conventional approach) is ineffective in Object oriented context.
3. Inheritance makes the subclasses dependent on the super class and a change in the super class will directly affect the subclasses that inherit from it means we have to retest all its subclasses. For e.g. if class A inherits class B means A is dependant on B, therefore changing B will also affect A. Hence it increases dependency among classes which results in low testability.
4. Testing is also affected by Depth in inheritance tree i.e. more the depth of a tree more dependency among the classes .All the subclasses which are inheriting the features of super class need to be tested again in their context.
5. Some Languages (like C++) support multiple inheritance which creates new challenges in testing object oriented systems. For e.g. in the figure given below class C is derived from two super classes A and B. Here function x () is not defined locally in C, hence it will inherits it from its parent classes .Invoking x () by using instance of class C will create ambiguity as which code will be used is not clear.



6. Suppose if there is a need of derived class to access the private members of base class then moving that member to the public access specifier will take away the purpose of data hiding and also placing the significant data in a class for all derived classes to use leaves the data open for corruption. Hence the better approach is to place it under protected access specifier because
Protected access specifier exposes the members to derived class only rather than exposing it everywhere an object is created.

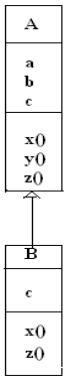
2.3 Polymorphism: Polymorphism means the ability to assume more than one form, both in terms of data and operations. It is the capability of an operation exhibiting different behavior in different instances. The behavior depends on types of data used. It allows an object reference to bind with objects of other classes. It allows that a standard interface may be created for group of objects. The object's action will depend on the message passed to the interface and different objects behave differently by receiving same message. Since the programmer is no longer concerned with the internal structure of objects, we can create complex program as he only needs to understand the interface to use objects. However polymorphism results in lack of controllability as actual binding of object reference is not known till run time. Some languages like C++ support Polymorphism in two ways i.e. static binding (function calls can be resolved during compile time as in traditional procedural language procedure calls are bound statically [5] and dynamic binding (the decision to which method is to be used is left till run time. Hence dynamic binding is closely related to Polymorphism. It introduces undecidability concern in program based testing as it can lead to messages sent to wrong object.




```
Void figure (Shape object) {
  ….
 Area=object. area ();
 …
```

In the above example, it is impossible to say at compile time which implementation of method area will be actually implemented.

To gain confidence in code containing in method calls on polymorphic entities, all the possible bindings need to be exercised but exhaustive testing of all possible combination of bindings may be impractical. Also if message is sent from one object to other object and if both objects are polymorphic in nature than number of possible combination depend on type of sender, type of receiver, type of parameters thus leads to a further increase in no. of test cases to be generated. To understand the problems created by polymorphism consider the figure given below. In the figure polymorphism can create many problem as method x () is redefined in derived class B hence it overrides the super class method. Hence in the presence of Polymorphism, the code associated with a polymorphic reference for function x () is known only at run time. If instance of A is used to call a method A.x () followed by A.y () and A.z (), the call will not create any problem as A.a and A.b used in A.y () and A.z () are already defined in A.x ().However if instance of B is used in place of instance of A then situation will be different as A.x () and B.x () has different definition sets i.e. A.a and A.b is defined in A.x () but not in B.x (). Therefore if call to A.x () is followed by A.y () using instance of B and data flow anomaly will occurs because A.b is used by A.z () which is not defined.



Methods	Definition	Uses
A.x()	A.a , A.b	
A.y()		A.a
A.z()		A.b
B.x()	B.c	A.b
B.y()		

2.4 Coupling: Coupling is the strength of interconnection between the components. Since modules are decided during system design, hence coupling between modules is largely decided during designing. The more tightly coupled the modules are, the more dependent they are on each other and more difficult is to understand and test them. The degree of coupling between a module and another module depends on how much information is needed about the other module for understanding and modifying this module. In presence if object oriented systems coupling may be classified as subclass coupling and temporal coupling. Subclass coupling describes the relationship between a child and its parent. The child is connected to its parent, but the parent isn't connected to the child.Temporal coupling when two actions are bundled together into one module just because they happen to occur at the same time.However tightly coupled systems tend to exhibit the various problems related to testing.A change in one module usually forces a ripple effect of changes in other modules.Secondaly integration of modules might require more effort and time due to the increased inter-module dependency , also a particular module might be harder to test in isolation without dependent modules.

Coupling increases between two classes ***X***and Yif:

- *X* has an attribute that refers to Y causing global coupling.
- X calls on services of an object Y again causing global coupling.
- X has a method that references Y causing parametric coupling .
- X is a subclass of (or implements) class Ycausing inheritance coupling.

Coupling is always undesirable as it prevents the changes of components independently of the whole**.** Programmers need to understand potentially the whole system to be able to safely modify a single component making module might be harder to test in isolation without dependent modules. Coupling between the classes and their dependencies among classes is shown through the diagram. The arrows in figure indicate dependency among classes. The component under test is coupled with many other classes hence testing that component in isolation is almost impossible. If an attempt is made to test component under test in isolation then it would involve stubbing all the external classes, which is difficult or impossible.

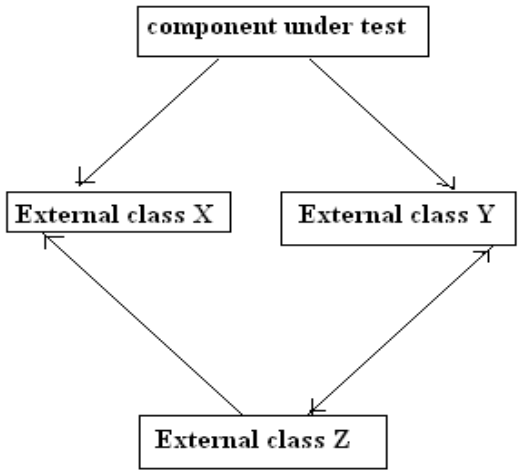


Figure showing component under test and its dependencies

2.5 Cohesion among methods: "**Cohesion** is a measure of how closely those methods are related to local instance variables within the class ".The class is said to have high cohesion, if the methods that serve the given class contribute to a single well-defined task and tend to be similar in many aspects. Modules with high cohesion tend to be preferable because high cohesion is associated with several desirable features of software including reliability, reusability, and understandability whereas low cohesion is associated with undesirable things such as being difficult to maintain, difficult to test, difficult to reuse, and even difficult to understand. A method is said to be less cohesive if the functionalities embedded in a class, accessed through its methods, have little in common. Methods carry out many different activities, often using unrelated sets of data. Low cohesion usually means poor design or poor organization of a class and thereby increasing the complexity of a class and increasing the testing effort. It results in making the module less understandable, difficult to maintain as changes in one require changes in all related module. 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 decreasing the testability of the class. Hence Testability of a class is directly proportional to cohesion. . Cohesion is often contrasted with coupling, high cohesion often correlates with loose coupling and low cohesion relates with strong coupling

2.6 Abstract Classes: Abstract class is the way to push up common implementation into a base class. Hence adding new objects are easier, because a lot of the common interfaces may already be implemented. These classes are designed only to act as a base class (to be inherited by other classes), defer the implementation of some features, and are typically used as a specification. However, since their features are not fully implemented, these classes cannot be instantiated and thus pose challenges for execution base testing. Only classes derived from the abstract class can be easily tested, but errors can be present also in the super class i.e. abstract class. Hence we need to write an abstract test class for every abstract class also. Also the common practice that is used to create a set of tests once, which can be run against all new objects they create from an abstract class, is questionable.

3. Results and Findings: During this comparative analytical study we have found that object oriented systems may reduce some kind of errors however they increase the chances of others. Encapsulation requires that classes are only aware of their own properties, and are able to operate independently. However if you do not have access to source code then structural testing can be impossible. If you violate encapsulation for testing purposes, then the validity of test could be questionable. Inheritance make unit testing of a sub class impossible to do without the super classes methods/variables. Polymorphism causes repeatedly testing same methods. When inheritance combined with the power of polymorphism and dynamic binding, makes it difficult to detect faults that result from the integration of units. This is because the integration of classes and components is more challenging in object-oriented languages. The major problem towards integration testing is that it is tough to determine the order in which classes should be integrated and tested. Secondly it is tough to determine the coverage criteria. To test polymorphism every possible form of component requires a separate test. It is not so easy to find all such bindings hence increases chances of faults and making coverage goals tougher. However to overcome these hindrances while developing new software we have to assure that open closed principle is followed properly i.e." Software should be open for extension, but closed for modification". When requirements change, we can extend the behavior of existing modules by adding new code, not by changing old code that already works. Another convention is that as far as possible we should make all member variables private so that member variables of classes should be known only to the methods of the class that defines them. Member variables should never be known to any other class, including derived classes. Because if the member variables of a class change, every function that depends upon those variables must be changed. Coupling can be reduced by making use of instance variable rather than static variable as instance variables can be accessed only through appropriate object references rather whenever class name is available.

4. **Conclusion:** Object oriented features introduce new testing challenges that cannot be easily addressed with the previous techniques used in traditional approaches and require definitions of new techniques, new problems that require new solutions. Object-oriented source code is harder to test than procedure oriented due to the fragmentation of functionality that can result from inheritance, dynamic binding, where control flow and state control are distributed over several classes. Due to these hindrances, the static testing techniques are less effective as compared to dynamic testing techniques. Encapsulation affects controllability and observability as discussed above which reduce testability of systems. Inheritance and polymorphism increase the class dependency hence making integration testing more challenging. Hence developers need to put more emphasis on integration testing than on unit testing. Therefore to surpass the quality provided by conventional testing techniques, we need to devise more testing tools to bypass these obstacles. Another observation is that we need to invent more effective techniques to increase the testability of systems in order to make software testing easy.

5. References:

[1] Morris S. Johnson, Jr. A survey of testing techniques for Object oriented systems, Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative Research.

[2] B.G Geetha V. Palanisamy, K.Duraiswamy a Tool for Testing of Inheritance Related Bugs in Object oriented software, Journal of Computer Science 4(1): 59-65, 2008.

[3] John Rosenberg and Michael Kolling, Testing Object-Oriented Programs: making it Simple, Proceedings of the twenty-eighth SIGCSE' 97 technical symposium on Computer science.education

[4] Dinesh Kumar Saini Testing Polymorphism in Object oriented Systems for improving Software Quality, ACM SIGSOFT Software Engineering Notes Volume 34 Issue 2, and March 2009.

[5] Robert V. Binder Testing Object-oriented Systems: A Status Report published in April 1994 issue of American Programmer.

[6] Roger T. Alexander, Jeff Offutt, Andreas Stefik Testing coupling relationships in object oriented programs , Article first published online, 21 JAN 2010 DOI: 10.1002/stvr.417.

[7] M.Smith and D.Robson ," A Framework for testing object oriented programs , journal of Object-Oriented programming , june 1994, pp.45-53.

[8] Pressman, Roger S. Ph.D (1982). Software Engineering - A Practitioner's Approach - Fourth Edition.

[9] W. Stevens, G. Myers, L. Constantine, "Structured Design", IBM Systems Journal, 13 (2), 115-139,1974.