# Testing Anomalies in Multiple and Multilevel Inheritance

**Thesis Report**

Submitted in partial fulfillment of the requirements

for the award of degree of

**Master of Engineering**

**In**

**Computer Engineering**


Submitted by

**Shubpreet kaur**

**Roll No. 800932021**

Under the supervision of:

**Ms. Shivani Goel**

Assistant Professor

CSED, TU, Patiala



COMPUTER SCIENCE AND ENGINEERING DEPARTEMENT

THAPAR UNIVERSITY

PATIALA—147004


**May 2011**

# Certificate

I hereby certify that the matter which is being presented in the thesis report titled," **Testing Anomalies in Multiple and Multilevel Inheritance**" in the partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Ms. Shivani Goel and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Shubpreet Kaur

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Ms. Shivani Goel

Assistant Professor, CSED

**Countersigned by**

**(Dr. Maninder Singh)**

Head

Computer Science and Engineering Department

Thapar University

Patiala

**(Dr. S. K. Mohapatra)**

Dean (Academic Affairs)

Thapar University

Patiala

**Dedicated**

**To**

**My Guide and My Parents**

***"Guru Gobind Dau khade, kaike Langoo pai***

***Balihari Guru apne, jin Govind diyo milaye"***

# Acknowledgment

*I wish to express my deep gratitude to Ms. Shivani Goel, Assistant Professor in Computer Science & Engineering Department, Thapar University,Patiala for providing his uncanny guidance and support throughout the thesis.*

*I am thankful to Dr.Maninder Singh, Head, Computer Science & Engineering Department and Dr.(Mrs).Seema Bawa, Dean of Student Affairs, Thapar University, Patiala for the motivation and inspiration that triggered me for the thesis work.*

*I would also like to thank all the staff members and my classmates who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of the thesis.*

*Last but not the least, I express my heartfelt thanks to my parents, my friends and well-wishers for co-operation, which they were always ready to extend.*

*Shubpreet kaur*

*800932021*

*M.E.(Computer Engineering)-2nd year*

*Computer Science & Engineering Department*

*Thapar University*

*Patiala -147004*

# Abstract

Software testing is an important phase of software development process that can be easily missed by software developers because of their limited time to complete the project. Thus, Software testing is a widely used and accepted approach for verification and validation of a software system. One of the major challenges in software testing is the generation of test cases that satisfy the given competence criterion. Testing in an object oriented manner significantly increases software reusability, extendibility, interoperability, and reliability. Object oriented testing has to deal with the new problems to deal with the new problems introduced by the new features of Object oriented systems such as inheritance, polymorphism, method overloading, method overriding etc. Object oriented testing technique for testing software units that has great potential for improving the quality of testing and to assure the high reliability of software. In this thesis, we will focus on the features of object oriented systems that create many binding anomalies during static and dynamic binding. This shows that a value of the variable changes with the change in the object if there is given a wrong function call with the object of other class. To detect such anomalies an approach is discussed that will detect static and dynamic anomalies in multiple and multilevel inheritance.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1 Introduction

*According to **Fewster, Graham** ,"Testing is a skill. While this may come as a surprise to some people it is a simple fact." [32]*

Testing is very important part of software development. Almost 80% software fails because of not testing properly. Software testing is a widely used and accepted approach for verification and validation of a software system, and it can be regarded as the ultimate review of its specification, design, and implementation. Testing is applied to generate modes of operation on the final product that show whether it is conforming to its original requirements specification, and to support the confidence in its safe and correct operation [3]. Generally testing is performed on code, but if the software can be tested in the earlier phases then most of the errors can be eliminated and can be stopped from propagating to next phase. The software testing document, consists of event, action, input, output, expected result, and actual result. Clinically defined a test case is an input and an expected result [15]. Testing based merely on source code documents shows that the tested program does what it does, but not what it is supposed to do. Thus there is a need to explore testing possibilities in earlier phases. One of the major challenges in software testing is the generation of test cases that satisfy the given competence criterion [17].

## 1.1 Testing Methods

There are various testing methods. Some of them are discussed below:-

- **White box testing** strategy deals with the internal logic and structure of the code. White box testing is also called as glass, structural, open box or clear box testing. The tests written based on the white box testing strategy incorporate coverage of the code written, branches, paths, statements and internal logic of the code etc. In order to implement white box testing, the tester has to deal with the code and hence is needed to possess knowledge of coding and logic i.e. internal working of the code. White box test also needs the tester to look into the code and find out which unit/ statement/ chunk of the code is malfunctioning [4]. It mainly aims at testing whether we are getting required output with given output or not.

- **Black box testing** as the name suggests gives only the external view of the software. This type of testing involves, testing either functional or non-functional aspects of the software, without any sort of reference to the internal structure of the software [8].
- **Gray box testing** technique refers to the technique of testing a system with limited knowledge of the internals of the system. Gray box testers have the access to detailed design documents with information beyond requirement document. The developer carries out unit testing in order to check if the particular module or unit of code is working fine.

## 1.2 Types of Testing

Testing can be done at various levels as discussed below:-

- **Unit testing:** The developer carries out unit testing in order to check if the particular module or unit of code is working fine. The Unit Testing comes at the very basic level as it is carried out as and when the unit of the code is developed or a particular functionality is built [3].
- **Integration testing:** Testing in which modules are combined and tested as a group. Modules are typically code modules, individual applications, client and server applications on a network, etc. Integration Testing follows unit testing and precedes system testing.
- **System testing:** This testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic [4].
- **Regression testing** Similar in scope to a functional test, a regression test allows a consistent, repeatable validation of each new release of a product or Web site. Such testing ensures reported product defects have been corrected for each new release and that no new quality problems were introduced in the maintenance process. Though regression testing can be performed manually an automated test suite is often used to reduce the time and resources needed to perform the required testing [17].

## 1.3 Test Case

A test case has components that describe an input, action or event and an expected response, to determine if a feature of an application is working correctly [13].

### 1.3.1 Need of a test case

The basic objective of writing test cases is **to validate the testing coverage of the application.** CMMI companies strictly follow test cases standards. So writing test cases brings some sort of standardization and minimizes the ad-hoc approach in testing [21].

### 1.3.2 Fields in test cases

Fields in a test case are:-

**Test case id:** Unique for each test case

**Unit to test:** What to be verified?

**Test data:** Variables and their values

**Expected value:** What should be the output if ok?

**Actual value:** What is the actual output?

**Status:** Pass if the actual and expected values are same else fails.

**Test case example**

The following test case tests the output of the function call u.input() which is entering the username, p.input() which is entering the password details and t.display() is displaying the thanks message if username and password matches.

Table 1.1 Test case example

| Test case ID | Sequence call/Steps | Expected value | Actual value | Status |
|---|---|---|---|---|
| 1 | u.input()->p.input()->t.display() | username=*abc,password=*38,thanks for entering username and pwd. | username=*abc,password=*38,thanks for entering username and pwd. | pass |

## 1.4 Object Oriented Programming (OOP)

**Object-oriented programming** is a programming paradigm using "objects" which are the data structures consisting of data fields and methods together with their interactions to design applications and computer programs.

### 1.4.1 Features of Object Oriented Programming

- **Class**- A class is a template for an object, a user-defined data type that contains variables, properties of an object. A class defines abstract characteristics of a thing (object), including its characteristics (its attributes, fields or properties) and the things it can do (**behaviors, methods, operations or features**). One might say that a class is a blueprint or factory that describes the nature of something. For example, the class `Dog` would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark and sit (behaviors). Classes provide modularity and structure in an object-oriented computer program.
- **Object-** An object is the basic building block of object-oriented programming. Programmers developing a system model create object classes to represent each component of a system. Those generic classes are then used to create specific instances of each object for use in the program. Objects may interact with each other through the use of methods. An object doesn't exist until an instance of the class has been created. A class is just a definition. When the object is physically created, space for that object is allocated in RAM. Multiple objects can be created from one class.
- **Encapsulation-** Encapsulation in OOP-terminology means that an object has something like a shell around it protecting the attributes or status variables from being accessed from outside of the object. They belong to the object and to nobody else. Only the methods of the object have access to them, can read and modify their values. The methods, providing the services of the object, are the interface to the outside world.
- **Polymorphism-** Polymorphism in OOP-terminology means that a certain message may be implemented by different objects in different ways. A good example is the `display'-message sent to a number, a string of characters or a list.

It is always the message 'display yourself!'. The object must know itself what has to be done and how, nobody cares as long as the object appears on the screen.

- **Method Overriding-** Method overriding in Object Oriented Programming , is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super classes or parent classes. The implementation in the subclass overrides (replaces) the implementation in the super class by providing a method that has same name, same parameters or signature, and same return type as the method in the parent class.
- **Inheritance-** Inheritance is the process by which new classes called *derived* classes are created from existing classes called *base* classes. The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class [25].

**Benefits of inheritance**

- Reusability of code
- Saves Time and Effort
- Extendibility
- Information Hiding
- Increases Program Structure which results in greater reliability**.**

### 1.4.2 Types of inheritance

- **Single inheritance-** In which derived classes have only one base class as shown in figure 1.1(a).

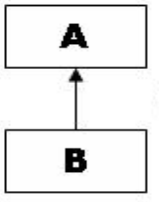


Figure 1.1(a) Single inheritance

**Example**- Vehicle is a base class and car is a derived class and car is using the inherited features of Vehicle such as Register()shown in figure 1.1(b).

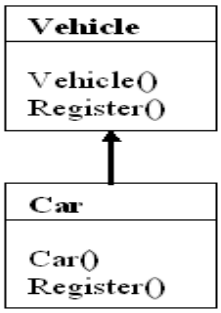


Figure 1.1(b) Example Single inheritance

- **Multiple inheritance-** It refers to a feature of some Object-oriented programming languages like C++ etc. in which a class can inherit behaviors and features from more than one super class as shown in Figure 1.2(a).

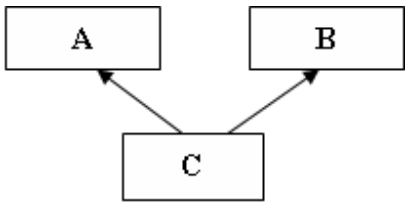


Figure 1.2(a) Multiple inheritance

**Example**- In figure 1.2(b) List and Ordered are base classes and OrderedList is a derived class which is inheriting the features of List and Ordered class. Java does not support multiple inheritance.

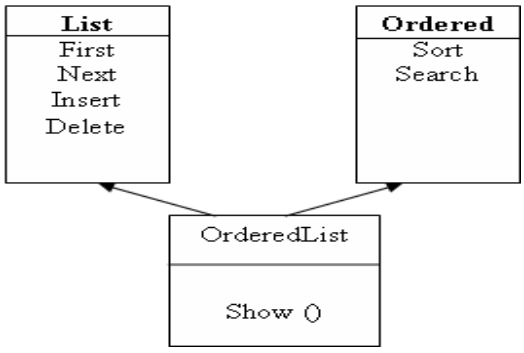


Figure 1.2(b) Example Multiple inheritance

- **Multilevel Inheritance-** It is the enhancement of the concept of inheritance. When a subclass is derived from a derived class then this mechanism is known as

the multilevel inheritance. The derived class is called the subclass or child class for its parent class and this parent class works as the child class for it's just above (parent) class. Figure 1.3(a) shows multilevel inheritance can go up to any number of levels.

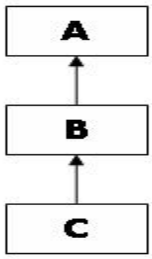


Figure 1.3(a) Multilevel inheritance

**Example-** Vehicle can be an abstract class and Bus is inheriting features of Vehicle and A/C Bus is a derived class of Bus which is inheriting the features of Vehicle as shown in figure 1.3(b).

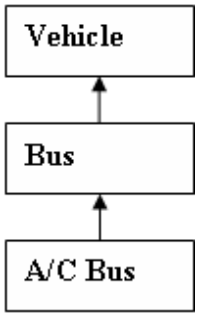


Figure 1.3(b) Example Multilevel inheritance

### 1.4.3 Access Modifiers

Access Modifiers are keywords used to specify the declared accessibility of a member of a type.

- **Public** is visible to everyone. A public member can be accessed using an instance of a class, by a class's internal code, and by any descendants of a class.
- **Private** is hidden and usable only by the class itself. No code using a class instance can access a private member directly and neither can a descendant class.
- **Protected** members are similar to private ones in that they are accessible only by the containing class. Protected members also may be used by a descendant class.

Members that are likely to be needed by a descendant class should be marked protected [16].

### 1.4.4 Types of Bindings

- **Static Binding or Early binding** means that our code directly interacts with the object, by directly calling its methods. Since the compiler knows the object's data type ahead of time, it can directly compile code to invoke the methods on the object.
- **Dynamic Binding or Late binding** means that our code interacts with an object dynamically at run-time. This provides a great deal of flexibility since our code literally doesn't care what type of object it is interacting with as long as the object supports the methods to be called [19].

In this chapter, what software testing is and why it is necessary is discussed. Next are various testing methods such as black box testing, white box testing and gray box testing. Then is a Test case which we start making at requirement analysis so as to give reliable and good quality software with no or least level of bugs. Its template is given which tells the required attributes and an example how to write test cases. After that Object Oriented Programming its various features including inheritance are explained. The use of Access modifiers and its types such as public, private and protected which tells the inheritance scope of one class with the other. Types of bindings as static and dynamic binding which is compile time and run time binding respectively are also discussed.

# Chapter 2 Literature Survey

*According to **Duke of Wellington,** "The business of life is to endeavor to find out what you don't know from what you do; that's what I called 'guessing what was on the other side of the hill" [30]*

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Since it is crucial to software quality and widely deployed by programmers and testers, software testing still remains an art, due to limited understanding of the principles of software [30] .Object oriented approach for programming is the need of the hour. Many researchers are trying to make it error free. The work done in this area till date is summarized here. Andrew has used class diagrams, test adequacy criteria which are defined for use in dynamic testing. A test criterion is defined by using collaboration diagram [1]. Rountev has presented a general approach for adapting whole program class analyses to operate on program fragments in polymorphism [2]. Alexander described the syntactic patterns for each OO fault type say the software contains an anomaly and possibly a fault [23]. There have been some conflicts in ideas, concepts, and opinions among researchers regarding Object Oriented Programming [27]. Robert V. Binder focused on test case design - heuristic and formal techniques to develop test cases from object-oriented representations and implementations, testability - factors in controllability and observability [23].Various issues and problems that are associated with testing polymorphic behavior were discussed by Saini [26] as discussed below.

## 2.1 Single Inheritance

### 2.1.1 Problems in Single inheritance with polymorphism and method overriding

In single inheritance there are two classes X and Y with variables and functions as shown in Figure 2.1. Suppose that an instance of Y is used instead of instance of X. It can be seen that the overriding method Y.o() has a different definition set than X.o()

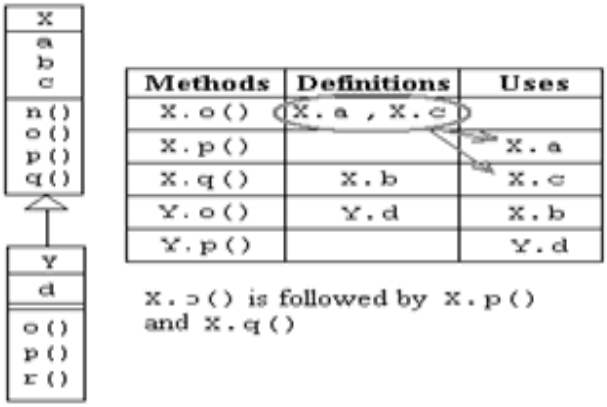


Figure 2.1 Definition use pairs resulting from X.o() followed by X.p() and X.q() [25]

There are two classes X and Y and calling sequence is given in the order shown below in figure 2.2.

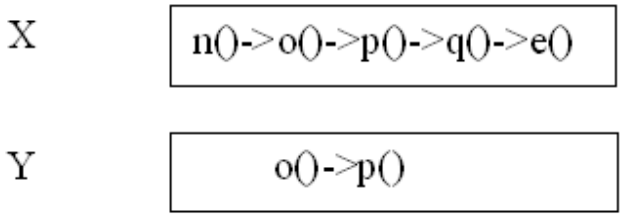


Figure 2.2 Order of sequence calls of Class X and Y

If the sequence of method calls shown in figure 2.3 were made in the context expecting an X but through an instance of Y, no data flow anomaly would exist. But if the call to X.o() is followed by Y.o() using instance of Y, an anomaly would exist.

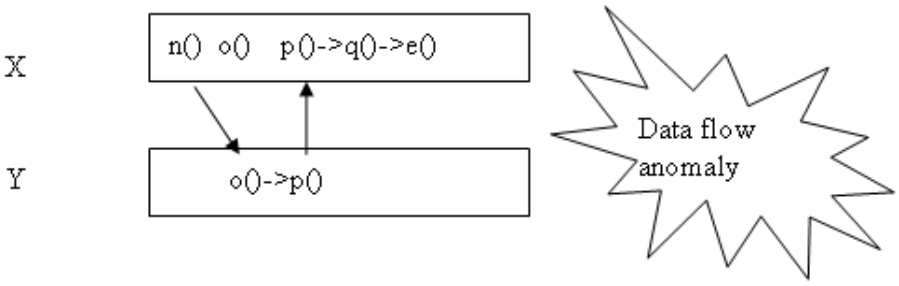


Figure 2.3 Method n() called through instance of Y and data anomaly in p() and q()

### 2.1.2 Some Definitions

In order to design an algorithm for detecting anomalies. Some terms are used which are defined below:-

**Type Family:** It is a set of classes that share a common behavior with respect to a base class A (*family (A)*). Each descendent of A is a member B of A's family. If B is in A's family, polymorphism means that any instance of B may be freely used wherever an instance of A is expected. Every class A defines a type family, and that type family includes at least A. Let *A* is a class in some type family.

**Used by:** A state variable $v \varepsilon A$ is *used by* some method $m \varepsilon A$ if *v* is used in some expression in *m*.

**Defined by:** A state variable $v \varepsilon A$ is *defined by* some method $m \varepsilon A$ if *m* assigns first legal value to *v*.

**Dependency of Methods:** Two methods $m,n \varepsilon A$, said that *m* is *dependent on n* if *m* uses a state variable $v \varepsilon A$ which is defined by *n*.

**Sequence diagram:** The sequence diagram can be a better way to understand the flow of control in program when initiation of one method execution initiates execution of another method. To be able to develop some algorithm which may work at design level for the above discussed problem and its sequence diagram is represented [6].

The sequence diagram of the example given in figure 2.1. Method n() is called on some object of X which in turn calls method o(), which calls p() and finally p() calls q() over the same object. This is what the intention of programmer was when he calls method n() over object of X using object reference of class X. The problem comes when object reference of X is associated with an object of Y which is a subclass of X and it overrides methods o() and p(). But since any object may be bind with the reference of its ancestor's type in a type family and this fact is unknown till run time. To be able to test it at design level an algorithm is given, which takes the sequence diagrams and dependency relation of methods as input and finds out the possibility of data anomaly because of polymorphism and method overriding [26].

### 2.1.3 Test Algorithm

Let A is some class of a type family TF, i.e. $A \in TF$ which has $n$ methods $m_i \in A$, for all $i$=1 to $n$ and $S$ is the set of all sequence diagrams.

**Inputs:** Sequence diagrams involving objects of some type class. Dependency relation of all the methods for all classes in the same type family.

**Output:** Whether there is possibility of data anomaly, if yes then it returns TRUE otherwise FALSE.

**Algorithm:**

1. **initialize** result = FALSE
2. **for** every class $A \in$ TF **do**
3. **for** all methods $m \in A$ **do**
4. **for** each state variable $v \in A$ used by $m_i$ ***do***
5. **if** $v_j$ is defined by $m_k \in A$ for some $k$ between 1 to $n$
6. **and if** $m_k$ is overridden by some class $B \in$ TF and $B$ is descendent of $A$ **then**
7. **for** each sequence diagram $S_l \in$ S **do**
8. **if** $S_l$ contains object of $A$ and there exists a call to $m_i$ from $m_k$ in $S_l$ **then**
10. result=TRUE
11. **return** result

**Explanation:** In object oriented paradigm the communication between objects takes place by message sending. A sender object sends a message to request a receiver object to fulfill a particular service [26]. A message sending sequence is a sequence of messages flying between objects in an interaction under a specific scenario. Required is to test the message sending sequence. The context in which a particular message sequence is initiated must be tested. The above algorithm is applied on some type family TF. It takes each of the classes $A$ one by one and if it finds any variable $v_i$ which is used by some method $m_i$ in that class. Then it checks whether that variable is defined by any other method $m_k$ and if so then it becomes important to check whether method $m_k$ is overridden by any descendent class $B$ because if it happens to occur then at run time the overridden method will execute even if some method of $A$ is called using the context of $B$'s object.

Now it is required to check all such sequence diagrams which contain an object of *A* and are making a call to method $m_i$ from method $m_k$. If such sequence occurs then polymorphism will create problem at run time and data anomaly would occur.

## 2.2 Multiple Inheritance

- Multiple Inheritance is the ability of a class to have more than one base class (super class)as shown in figure 2.4.
- Multiple inheritance complicates a programming language significantly, is hard to implement and is expensive to run [28]

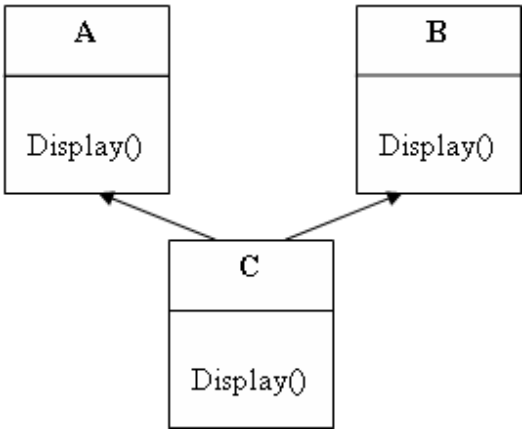


Figure 2.4 Multiple inheritance with no ambiguity

In figure 2.4 if the object of class C is created as obj and obj.Display() will display the method of itself. Output - “called with C object”. It will not override. It will not be checking in the other classes.

If the object of class C as obj and obj.Display() is existing in both the base classes will create **ambiguity** as shown in figure 2.5.

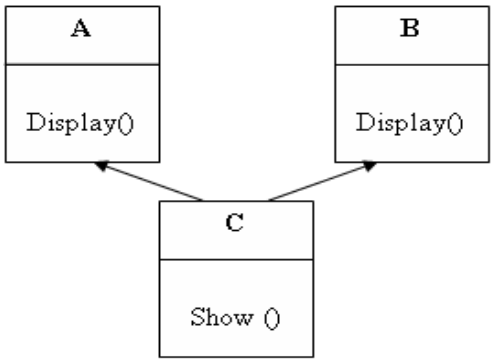


Figure 2.5 Multiple inheritance with ambiguity in Display()

### 2.2.1 Multiple inclusions

In this a class can have any number of base classes.

**Example** class A : B1, B2, B3, B4, B5, B6 { ... }.

It is illegal to specify the same class twice in a list of base classes.

Example class A : B, B { ... }; // error

It would be illegal and unambiguous [28]

## 2.3 Multilevel inheritance

Like single and multiple inheritance data flow anomalies occur in multilevel inheritance. It has shown 3 levels of classes with the overridden functions and defining that each overridden function has its own specific properties of its class. Thus wrong calling to a function will call other function which would return garbage values if it would not be defined or if it would have its dependency on other class. Figure 2.6 showing A::h is defining two variables and B::h is defining one variable. So instead of calling A::h, B::h is called then it would be a run time error [20].

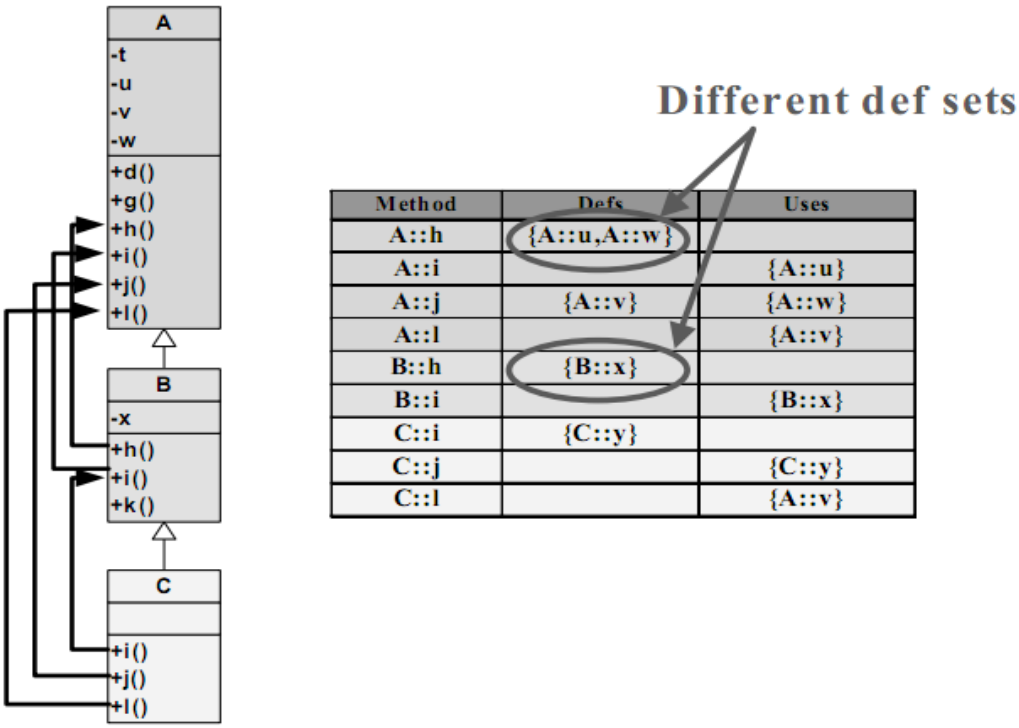


Figure 2.6 Hierarchy and definition uses table of multilevel inheritance [20]

Suppose the calling sequence given below in figure 2.7

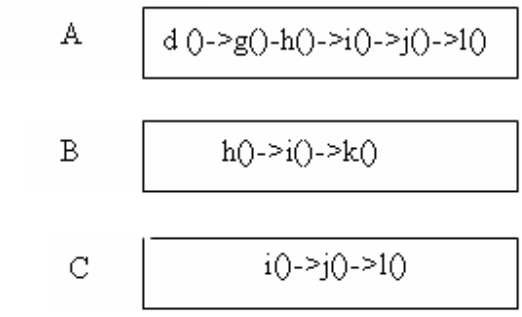


Figure 2.7 Sequence calls in multilevel inheritance

Data flow anomaly occurs; example after calling A::i A::j is to be called but C::j is wrongly called that would give data flow anomaly as shown in figure 2.8.

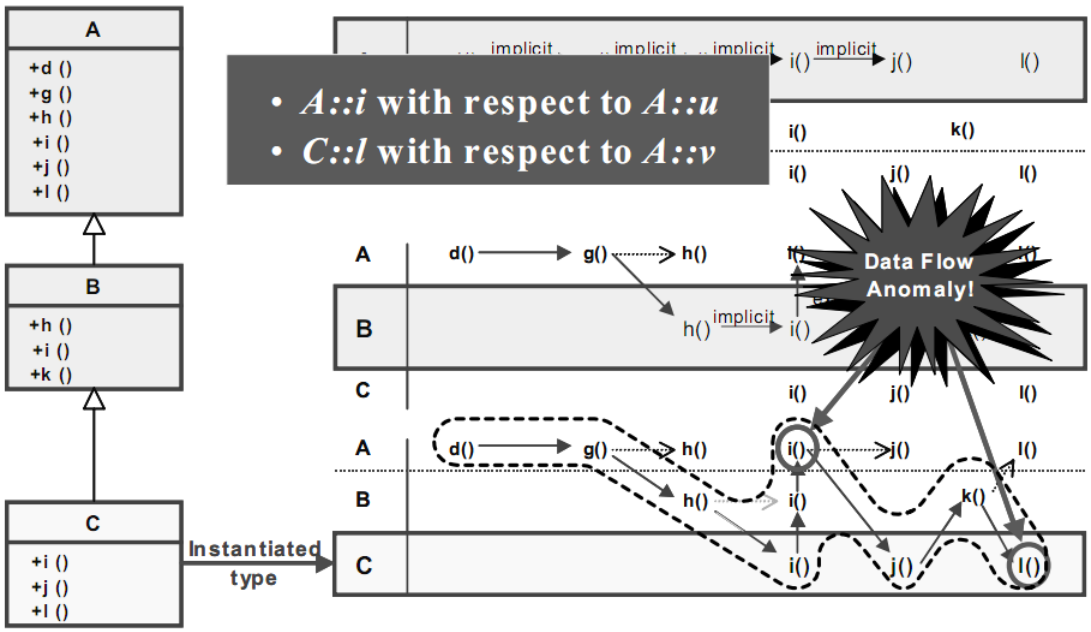


Figure 2.8 Data flow anomaly in multilevel inheritance [20]

Inheritance and polymorphism provide simplest ways of reuse in object oriented systems. Christian had focused on object oriented systems with integration testing as object oriented technology is too complex and it is a recursive and iterative process from analysis, design, implementation and testing view [4]. There are many modules having interactions which increases the complexity and makes integration testing difficult and creates problem with generation of test cases [5]. Alexander described the syntactic patterns for each object oriented fault type saying that the software contains an anomaly and possibly a fault [22]. Pressman suggested that integration of the system as a whole and then testing is a big bang approach in which all the units are taken together and tested together which is doomed to failure [6]. So incremental integration i.e. collaborate two units is used and then apply integration testing. Fault based testing is discussed by him which identifies the method calls, objects which has high likelihood of uncovering plausible faults [19].

# Chapter 3 Problem Statement

*According to **Tolkien,** "If we fail, we fall. If we succeed - then we will face the next task." [32]*

Various anomalies occur during compile time and run time in object oriented systems. They are discussed below:-

- **Compile time binding**: It is known as static binding. As shown in figure 3.1 on compiling a program it will give error in multiple inheritance if the same function name exits in various base classes but doesn't exist in the derived class. If the same function name exists in the derived class then it will call its function only i.e. it will not give call to functions that exists in the base classes so no ambiguity exists with anomaly removed.
- **Run time binding:** It is known as dynamic binding. It is binding of a member function with an object at. Anomalies occur in both multiple and multilevel inheritance. They are difficult to trap in large systems having large number of classes and objects. It doesn't give any error while compiling. But if the output is not as required i.e. it is giving garbage value and code is correct. Problem is the incorrect use of object while calling methods and functions because value is retained object only. Wrong function disturbs the whole system and the error is difficult to find.

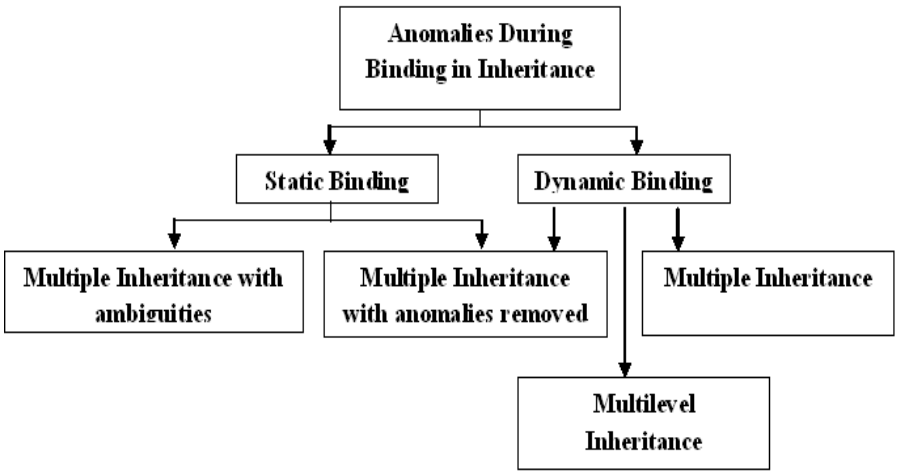


Figure 3.1 Anomalies during Binding in Multiple and Multilevel Inheritance

Ambiguity is a compile time error that occurs during static binding. And returning garbage value is a run time error that occurs during dynamic binding.

Various approaches discussed in Chapter 2 have identified anomalies during testing Object Oriented Programming using single inheritance [26]. None of the approaches identified anomalies during multiple and multilevel inheritance. Many anomalies exists and increases many folds as level of inheritance increases. Need is to design an algorithm for testing anomalies in multiple and multilevel inheritance.

# Chapter 4 Proposed Methodologies

*According to **Terry Pratchett,** "Winners never talk about glorious victories. That's because they are the ones who see what the battlefield looks like afterwards. It's only the losers who have glorious victories."[32].*

A solution is proposed to detect the anomalies discussed for both the multiple and multilevel inheritance.

## 4.1 Algorithm for Anomaly Detection of multiple inheritance

An algorithm is designed for detecting the anomalies in multiple inheritance. For this various definitions used in the algorithm are discussed.

The terms and definitions are same as discussed in chapter 2 [26].

### 4.1.1 Algorithm for static anomaly detection in multiple inheritance

*1.**initialize** result=false*

*2.**for** every class ε TF **do***

*3.**for** all methods $m_i$ ε A and A is the child class or leaf node **do***

*4.**if** there exists multiple parents for some p between 2 to n of child class **then***

*5.**if** each parent class is publicly inherited **then***

*6.**and** if method $m_k$ of parent class is also defined by some other parent class ε TF and child is the descendent of parent but the child class does not contain the method $m_k$ and there exists a call from child class object to method of $m_i$ parent class **then***

*7.result=true*

*8.**end** for*

*9.**end** for*

*10. **end** for*

*11. **if** there exists a call to method with the object specified with the class name **then***

*12. **return** result*

**Explanation**: Let's suppose initially there is no error in the program. So, for that results is initialized as false. Then every class is being checked that belongs to the same type family starting from child class. To check for all the methods defined in the child class. Here child class is the leaf node. If there exists multiple parents (2 or more than 2) of that child class then there can be the chances of ambiguity if both the parents have the same method present in both the classes but not defined in the child class. If child class also has the same method name defined then there will be no ambiguity at all because the call will go only to the method present in the child class. Other case is to check whether the parent's classes are publically inherited or not. If the all parent classes are publically inherited then there will be the chance of an error. Suppose there are two parent classes of one child class. One class is publically inherited and other is privately inherited then in that case there will be no error but in that case there can occur the problem of dynamic anomaly that if inherited the parent class privately and its data members are being used in the child class. Then instead of inheriting the original value the garbage value will be taken. For checking an ambiguity for the existence of multiple parents and whether they are publicly inherited. Then checking for the methods if they are present in all the parent classes but not in the child class. And child is the descendent of parent and calling the method of parent class with the object of child class. Then there exists the static ambiguity which is a compile time error. And the result changes from false to true that there is an ambiguity. If the method is called with the specified object with the class name and method, then there will be no ambiguity if same method name is defined in the other parent classes.

### 4.1.2 Algorithm for dynamic anomaly detection in multiple inheritance

1. ***initialize*** *result=false*
2. ***for*** *every class ε TF* ***do***
3. ***for*** *all methods* $m_i$ *ε A and A is the child class or leaf node* ***do***
4. ***if*** *there exists multiple parents for some p between 1 to n of child class* ***then***
5. ***for*** *parent class is inherited for some p between 1 to n* ***do***
6. ***for*** *each state variable* $\mathbf{v_j}$ *ε parent used by* $m_i$ ***do***
7. ***if*** $v_j$ *is defined by* $m_k$ *ε parent for some k between 1 to m*

8. ***and*** *if $m_k$ is overridden by child class ε TF and child is the descendent of parent* ***then***
9. ***if*** *there exists a call with child class object from $m_i$ to $m_k$ of parent* ***then***
10. *result=true*
11. ***end*** *for*
12. ***end*** *for*
13. ***end*** *for*
14. ***if*** *there exists a call to method with the object specifying the class name* ***then***
15. *result =false*
16. ***return*** *result*

**Explanation:** Here also suppose initially there is no error in the program. So, for that result is initialized as false. Then checking for every class that belongs to the same type family starting from child class. To check for all the methods defined in the child class. Here child class is the leaf node. If there exists multiple parents (2 or more than 2) of that child class then there can be the chances of anomaly that whether the parent class is publicly inherited or not. Anomaly exists because of the wrong object call is given. So for that checking the state variable that belongs to child class methods. So for this checking the state variable used by the parent and the variable $v_j$ is defined by method $m_k$ of parent class and this method is overridden by child class and there exists a call with child class object from $m_i$ to $m_k$ of parent class. If the result returns true i.e. there exists data flow anomaly. And if there exists a call to method with the object specifying the class name there would be no data flow anomaly and the result is false.

## 4.2 Algorithm for Anomaly Detection for multilevel inheritance

An algorithm is designed for detecting the anomalies in multilevel inheritance. For this various definitions used in the algorithm are discussed. Definitions are same as discussed by Saini [26].

### 4.2.1 Algorithm for anomaly detection in multilevel inheritance

1. ***initialize*** *result=false*
2. ***for*** *every class ε TF* ***do***

3. ***for*** *all methods* $m_i \varepsilon A$ ***do***
4. ***for*** *each state variable* $\mathbf{v_j} \varepsilon$ *parent used by* $m_i$ ***do***
5. ***if*** $v_j$ *is defined by* $m_k \varepsilon$ *parent for some k between 1 to n*
6. ***and*** *if* $m_k$ *is overridden by some class child* $\varepsilon$ *TF and child is descendent of parent* ***then***
7. ***if*** *child class contains state variable of parent class and there exists a call to* $m_i$ *from* $m_k$ ***then***
8. *result=true*
9. ***if*** *child class contains sub class* ***then***
10. ***set*** *child=subclass*
11. ***end*** *for*
12. ***end*** *for*
13. ***set*** *parent=parent +1*
14. ***end*** *for*
15. ***return*** *result*

**Explanation:** Let's suppose initially there is no bug in the program. So, for that result is initialized as false. Check for every class starting from parent class that belongs to a type family. In that class checking all the methods defined the parent class. Check for each state variable that is defined in the different methods of parent class. Check whether that method is overridden by its child if it exists. Then in those methods check for the state variable. If that state variable is used then there exists some data flow anomaly, the result becomes true. If there exists sub class of that child class i.e. the grand child, then swap grand child with a child and parent child relationship is there. And will check for the anomalies again and this continues till last level is reached After completing all the levels parent is incremented i.e. come one level down and make level 2 a parent and then the process continues from step2 for finding anomalies in the parent child relationship.

In proposed methodologies an approach for detecting anomalies in multiple inheritance with static and dynamic error and an approach for detecting anomalies in multilevel inheritance is discussed.

# Chapter 5 Results and Findings

*According to* ***Robert A. Heinlein,*** *"Everything is theoretically impossible, until it is done." [31]*

## 5.1 Single Inheritance

In this section results of single inheritance are discussed. Implementation of single inheritance is shown at first.

### 5.1.1 Implementation for single inheritance

There occur anomalies in single inheritance. Firstly variable part is discussed which shows whether variable is accessible with the required object or not example base class object cannot access derived class variable. Below are few cases given showing the possible cases of accessing a variable.

**CASE 1**: **When we try to access public variable of base class by base class or by derived class OBJECT then:**

Now in this case we do not get any error for this case we can access every public variable

**CASE 2: When we try to access private variable of base class by derived class OBJECT**:

In this we have 2 private variable which is not used in DERIVED class those are "e, f". So when we try to variable "e" by derived class object then the following error will come as indicated in figure 5.1.

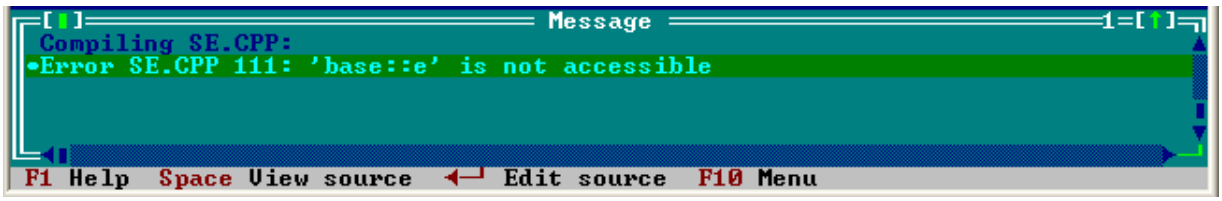


Figure 5.1 Private member of base class not accessible by derived class

**CASE 3**: **When we try to access public variable of base class by derived class OBJECT which is PRIVATE in derived class**

Figure 5.2 shows the variables which are public in base class but are PRIVATE in Derived class are “g, h”. When we try to access them we will get the error and we cannot access them as those are private member of derived class.


Figure 5.2 Public member of base class with private inheritance not accessible by derived class

**CASE 4: When we try to access private variable of base class by derived class OBJECT which are PUBLIC in derived class.**

In this case we **do not get the error** as those variables are public in derived class. The variables which are Private in base class but Public in derived class are: - “b, d”. These variables can be accessed by derived class object.

**CASE 5**: **When we try to access private variable/public variable of derived class (which is not a member of base class) by base class OBJECT**

We will get an error as we cannot access the variable of derived class which is not a member of base class (private or public). The error will be like this when we try to access one of the variables which are not a member of base class “n, o, p, and q “We run the case for “n” and for “q” as shown in figure 5.3.


Figure 5.3 Variable of derived class is not class not accessible by base class

Screen shots of simple inheritance of variable accessibility with an object are shown in figure 5.4. Similarly is the case in which accessibility of variable with different functions. Its screen shots are shown are shown in figure 5.5


Figure 5.4 Screen shots of accessing variable with an object in single inheritance

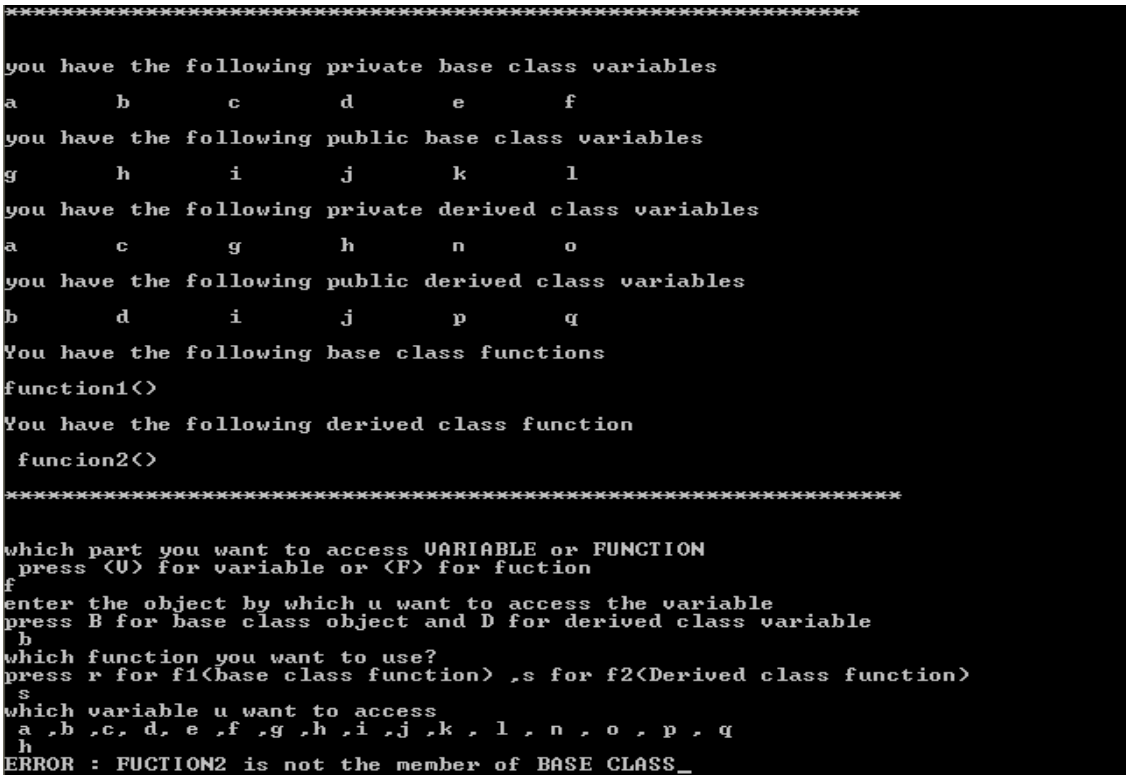


Figure 5.5 Screen shots of accessing variable with a function in single inheritance

In table 5.1 showing the accessibility of public and private members.

Table 5.1 Accessibility of public and private members.

| Object | Base public<br>Derived public<br>"i , j" | Base public<br>Derived private<br>"g , h" | Base private<br>Derived private<br>"a , c" | Base private<br>Derived public<br>"b ,d" | Base private<br>"a ,b ,c ,d ,e ,f" | Base public<br>"g ,h, l, j, k ,l" | Derived Private<br>"a, c, g, h, o, n" | Derived public<br>"b, d ,l, j, p, q" |
|---|---|---|---|---|---|---|---|---|
| Base Class | Yes (no error) | Yes (no error) | Error Case 1 | Error Case 1 | Error Case 1 | Yes (No error) | "a,c"– error<br>Case 1<br>"o, n"- error<br>Case-7<br>"g,h"– no error | "b,d"- error<br>Case 1<br>"p,q"- error<br>Case - 7<br>"i,j"-no error |
| Derived class | Yes (no error) | Error Case 8 | Error Case 8 | Yes (no error) | "e,f" error<br>case 3<br>"a,c" error<br>Case 8<br>"b,d" No error | Yes (no error) | Error Case 8 | Yes (no error) |

### 5.1.2 Program to implement single inheritance in C++/

**C**ode for detecting anomalies in single inheritance is shown in figure 5.6.

```
1. #include<iostream.h>
2. #include<conio.h>
3. class X
4. {public:int a,b,c;
5. public:o()
6.    { a=10;
7.      c=20;}
8.      p()
9.    { cout<<"\n a="<<a;  }
10.    q()
11.    { b=15;
12.     cout<<"\n c="<<c;
13.      }};
14. class Y:public X
15. {public:int d;
16. public:o()
17.      {      d=25;
18.      cout<<"\n b="<<b; }
19.      p()
20.      { cout<<"\n d="<<d;
21.      }};
22. void main()
23.{Xx1;
24. Yy1;
25.x1.o();
26.x1.p();
27.y1.o();
28.y1.p();
29.y1q();}
```

Figure 5.6 Code for detecting anomalies in single inheritance

By running the code no anomaly is found i.e. it will successfully run the code without static and dynamic errors. But if we change its sequence i.e. instead of calling x1.o() we are calling y1.o() and not changing the remaining sequence calls. We will not get static error but we will get dynamic error i.e. output will be garbage values because of the dependency of the one module to the other.

### 5.1.3 Test Results for Single inheritance

In Table 5.2 are the tests results of single inheritance showing there result.

Table 5.2 Test cases for single inheritance

| Test case ID | Sequence Call | Expected value | Actual value | Status |
|---|---|---|---|---|
| 1. | x.o->x.p->x.q->y.o->y.p | A=10,b=15,c=20,d=25 | A=10, b=gv, c=20,d=25 | fail |
| 2. | x.o->x.p->x.q->y.p->y.o | A=10,b=15,c=20,d=25 | A=10, b=gv, c=20,d=gv | fail |
| 3. | x.o->x.q->x.p->y.o->y.p | A=10,b=15,c=20,d=25 | A=10, b=gv, c=20,d=25 | fail |
| 4. | x.p->x.o->x.q->y.o->y.p | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=20,d=25 | fail |
| 5. | x.p->x.q->x.o->y.o->y.p | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=gv,d=25 | fail |
| 6. | x.q->x.o->x.p->y.o->y.p | A=10,b=15,c=20,d=25 | A=10, b=gv, c=gv,d=25 | fail |
| 7. | x.q->x.p->x.o->y.o->y.p | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=gv,d=25 | fail |
| 8. | x.o->x.q->x.p->y.p->y.o | A=10,b=15,c=20,d=25 | A=10, b=gv, c=20,d=gv | fail |
| 9. | x.p->x.o->x.q->y.p->y.o | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=20,d=gv | fail |
| 10. | x.p->x.q->x.o->y.p->y.o | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=gv,d=gv | fail |
| 11. | x.q->x.o->x.p->y.p->y.o | A=10,b=15,c=20,d=25 | A=10, b=gv, c=gv,d=gv | fail |
| 12. | x.q->x.p->x.o->y.p->y.o | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=gv,d=gv | fail |
| 13. | y.o->y.p->x.o->x.p->x.q | A=10,b=15,c=20,d=25 | A=10, b=gv, c=20,d=25 | fail |
| 14. | y.p->y.o->x.o->x.p->x.q | A=10,b=15,c=20,d=25 | A=10, b=gv, c=20,d=gv | fail |

| | | | | |
|---|---|---|---|---|
| 15. | y.o->y.p->x.o->x.q->x.p | A=10,b=15,c=20,d=25 | A=10, b=gv, c=20,d=25 | fail |
| 16. | y.o->y.p->x.p->x.o->x.q | A=10,b=15,c=20,d=25 | A=gv, b=15, c=20,d=25 | fail |
| 17. | y.o->y.p->x.p->x.q->x.o | A=10,b=15,c=20,d=25 | A=gv, b=15, c=gv,d=gv | fail |
| 18. | y.o->y.p->x.q->x.o->x.p | A=10,b=15,c=20,d=25 | A=10, b=gv, c=gv,d=25 | fail |
| 19. | y.o->y.p->x.q->x.p->x.o | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=gv,d=25 | fail |
| 20. | y.p->y.o->x.o->x.q->x.p | A=10,b=15,c=20,d=25 | A=10, b=gv, c=20,d=gv | fail |
| 21. | y.p->y.o->x.p->x.o->x.q | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=20,d=gv | fail |
| 22. | y.p->y.o->x.p->x.q->x.o | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=gv,d=gv | fail |
| 23. | y.p->y.o->x.q->x.o->x.p | A=10,b=15,c=20,d=25 | A=10, b=gv, c=gv,d=gv | fail |
| 24. | y.p->y.o->x.q->x.p->x.o | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=gv,d=gv | fail |
| 25. | x.o->y.p->x.q->y.o->x.p | A=10,b=15,c=20,d=25 | A=10, b=gv, c=20,d=gv | fail |
| 26. | x.o->y.p->y.q->y.o->y.p | A=10,b=15,c=20,d=25 | A=10, b=15, c=gv,d=gv | fail |
| 27. | x.o->x.p->y.q->y.o->x.p | A=10,b=15,c=20,d=25 | A=10, b=15, c=gv,d=25 | fail |
| 28. | y.o->x.p->x.q->y.o->x.p | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=gv,d=gv | fail |
| 29. | y.o->x.p->y.q->x.o->y.p | A=10,b=15,c=20,d=25 | A=gv, b=gv, c=gv,d=25 | fail |
| 30. | x.o->x.p->y.o->y.p->y.q | A=10,b=15,c=20,d=25 | A=10,b=15,c=20, d=25 | pass |

## 5.2 Multiple Inheritance

In multiple inheritance both the static and dynamic errors are there. Firstly start discussing with the dynamic errors and then static errors.

### 5.2.1 Implementation details for multiple inheritance.

In figure 5.7 an implementation example showing accessibility in multiple inheritance.

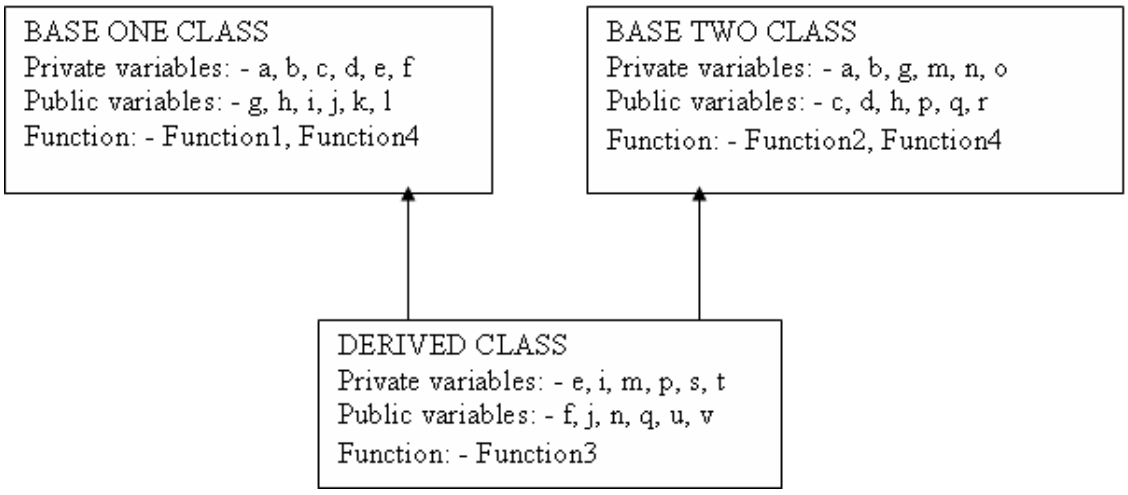


Figure 5.7 Implementation example multiple inheritance

#### 5.2.1.1 Accessing variable with an object

After testing the various errors occurred and few are explained below.

**CASE 1: When we try to access the variables which are present in both the base (parent) classes by derived class object.**

There are six private variables: a, b, c, d. when we try to access these variables by the derived class object the following error occurred as shown in figure 5.8.


Figure 5.8 Ambiguity in multiple inheritance

**CASE 2: When we try to access the variables which are defined in other classes (base two and derived classes) but not defined in base one class**

There are variables : m , n , o , p , q , r , s , t , u , v  which are not the member of base class or  not defined in base class , when we try to access those variables by base class object the following error occurred shown in figure 5.9.

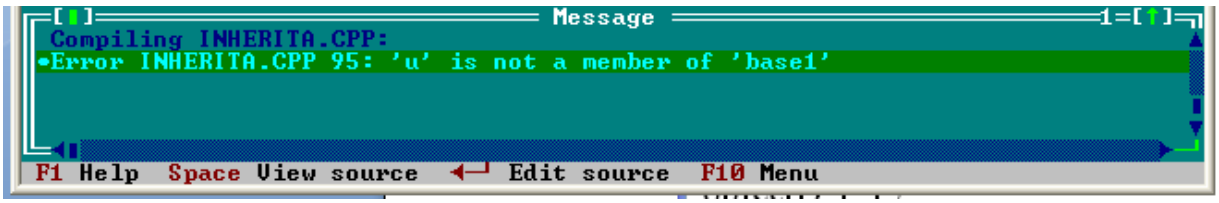


Figure 5.9 Base class accessing members which are not defined in it

The screen shots of the above cases are shown below in figure 5.10

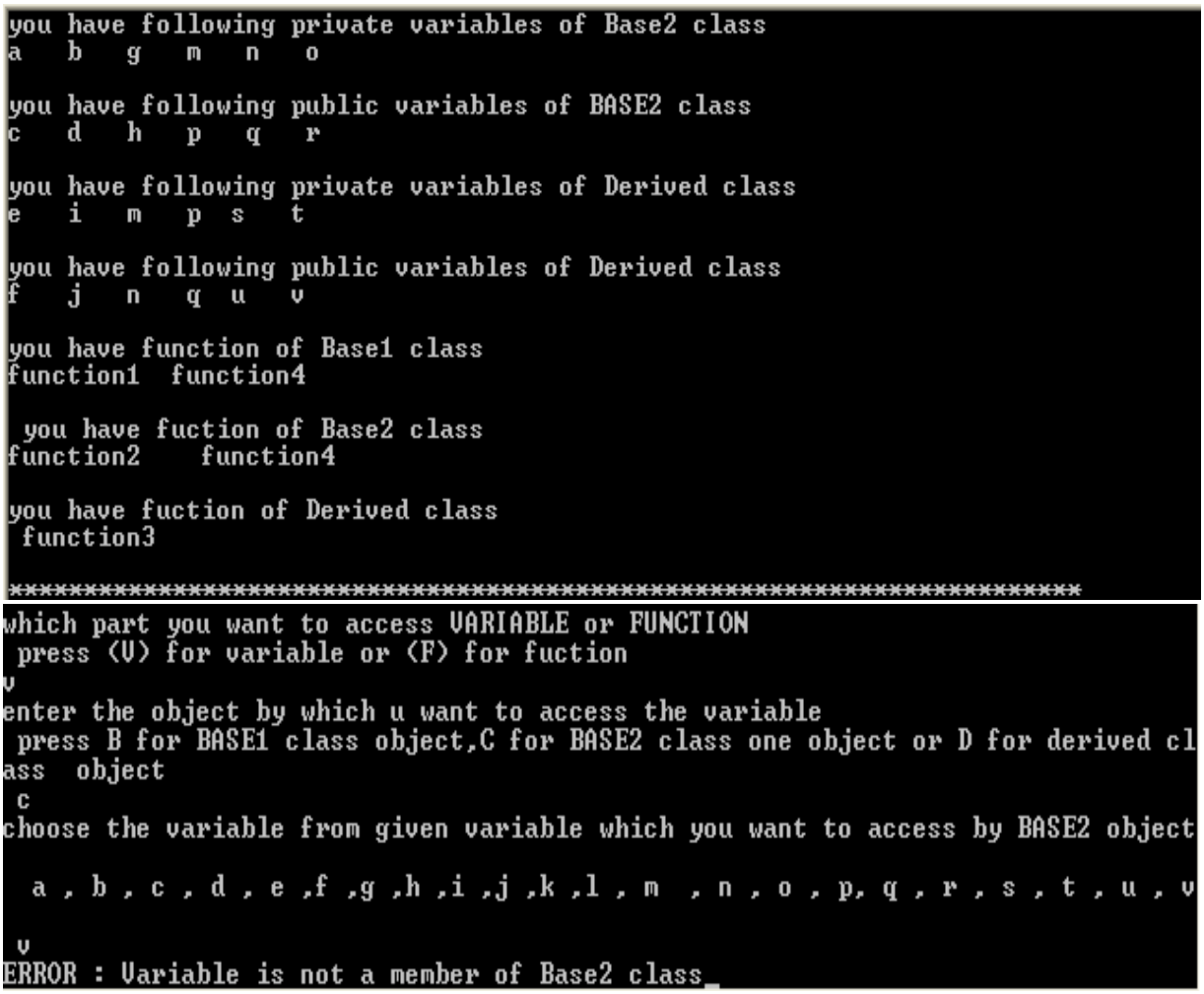


Figure 5.10 Screen shots accessing variable with an object

#### 5.2.1.2 Accessing Function with objects

Here we are experimenting with the three functions **Function1**, **Function2, function3.** Function1 belongs to the base class, Function2 belongs to the derived one class and function3 belongs to derived 2 class. In this part we will access all the variables of all classes one by one and with different class objects (base object and both derived class object). Different cases will be obtained and different error which will be classified below.

**CASE 1: When a base one class object is accessing a function 1, function4 and function1, function 4 is accessing the variables which are not defined in base one class**

The variables m, n, o, p, q, r, s, t, u, v are those variables which are not defined in base class. When function 1 which is defined in base class want to access these variables the following error will be occurred as indicated in figure 5.11.

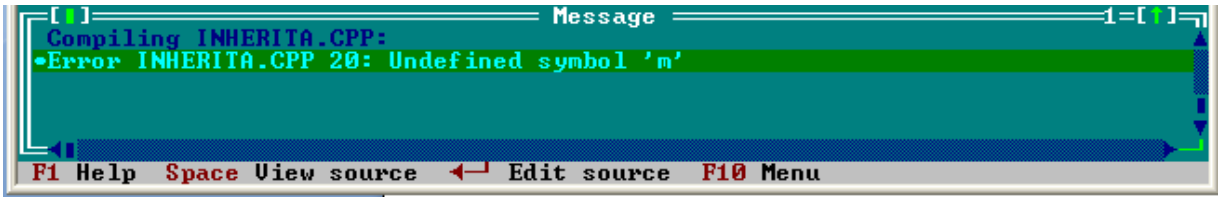


Figure 5.11 Base class functions accessing variables which are not defined in it

**CASE 2: When a Derived class objects want to access the variables via function3 which are present in both the base classes.**

Figure 5.12 indicates that there are variables a, b, c, g, h which are not defined in derived class and present in both the base classes. When derived class object want to access one of these variables via function3 then the following error will come.

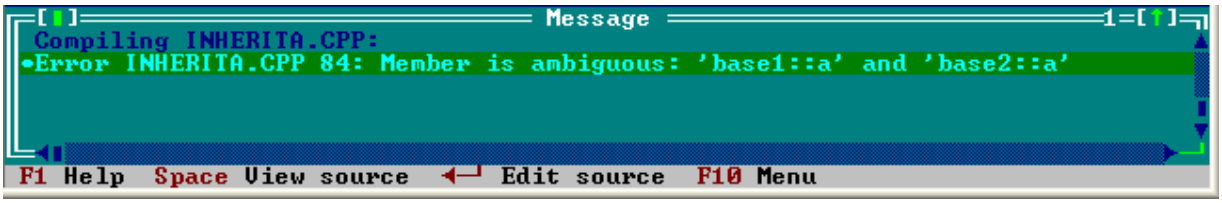


Figure 5.12 Derived class object wants to access variables which are present in both base classes.

The screen shots of accessing variables using functions in multiple inheritance are shown in figure 5.13.

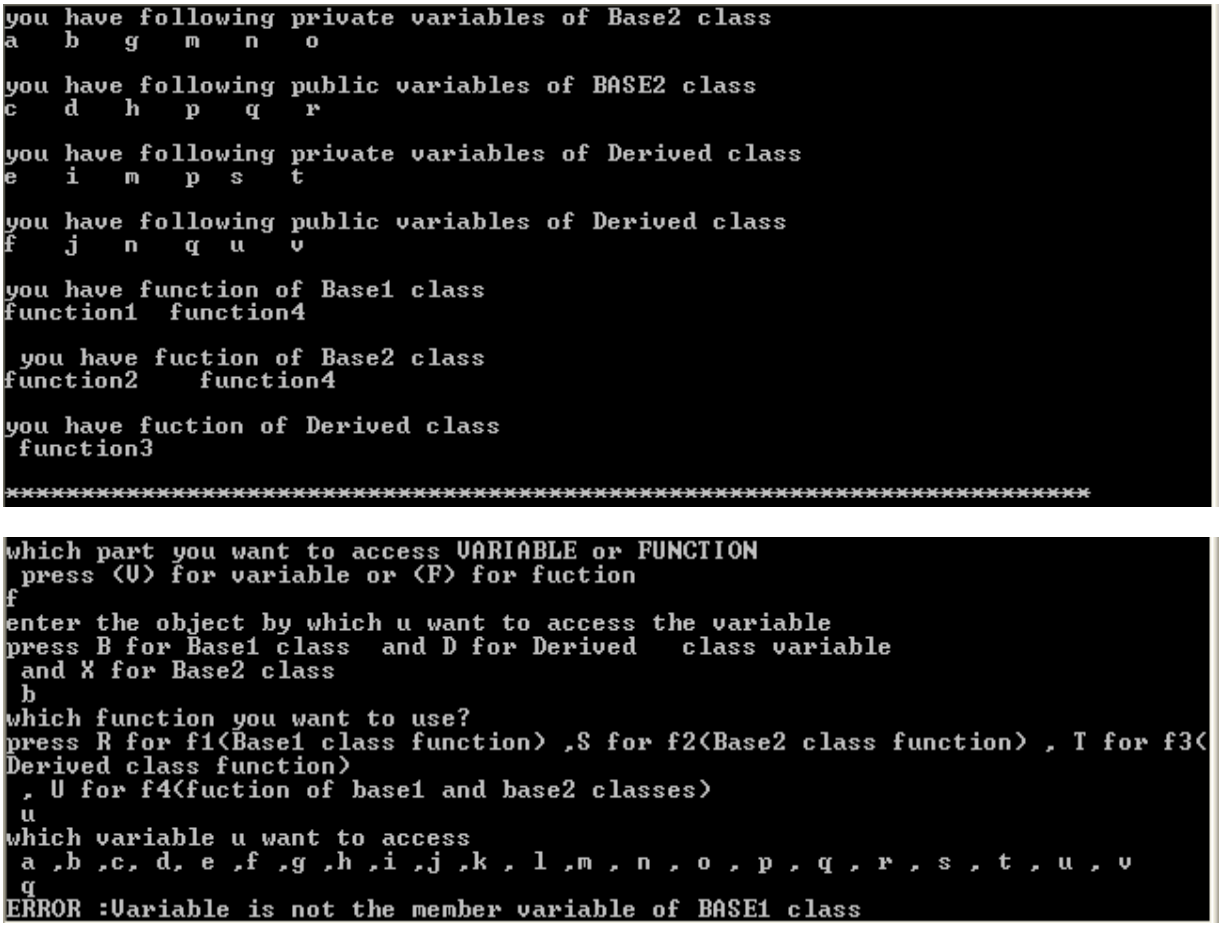


Figure 5.13 Screen shots of accessing a variable using function

### 5.2.2 Findings in Multiple inheritance

Multiple Inheritance is the ability of a class to have more than one base class (super class). The relationship between classes during multiple inheritance is shown in Figure 5.14.

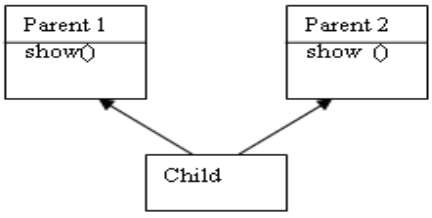


Figure 5.14 Multiple inheritance with static ambiguity

So calling the same method name of parent class will create compile time ambiguity [27].

### 5.2.3 Multiple Inheritance with Ambiguities in Dynamic (i.e. during Run-time) Binding

Consider the following program in figure 5.15 using multiple inheritance where the anomalies are present at run time:

```
1.  #include<iostream.h>
2.  #include<conio.h>
3.  class A
4.  {public: int a;
5.  void show()
6.  {a=10;
7.  cout<<"\n a="<<a;}};
8.  class B
9.  {public: int b;
10. void show()
11. {b=20;
12. cout<<"\n b="<<b;
13. }};
14. class C:public A,B
15. {public:int c;
16. void show()
17. {cout<<"\n c="<<a+b;
18. }};
19. void main()
20. {clrscr();
21. C ob;
22. ob.show();}
```

Figure 5.15 Code for detecting dynamic anomalies in multiple inheritance

In the above code, there is no ambiguity (i.e. no static error but a run-time error). There is a call to show() function which is present in all the three classes. Result will be garbage because class B is taken as private by default in line number14 and the value of variable b in class C will be taken garbage. Value of A will be correct as it is publicly derived. So, it will give run-time error.

### 5.2.4 Multiple Inheritance with ambiguities during Static (i.e. Compile time) Binding

Consider the following program in figure 5.16 using multiple inheritance where the anomalies are present at compile time:

```
1.  #include<iostream.h>
2.  #include<conio.h>
3.  class A
4.  {public:int a;
5.  void show()
6.  {a=10;
7.  cout<<"\n a="<<a;
8.  }};
9.  class B
10. {public:int b;
11. void show()
12. {b=20;
13. cout<<"\n b="<<b;
14. }};
15. class C:public A,public B
16. {public:int c;
17. void show1()
18. {cout<<"\n c="<<a+b;
19. }};
20. void main()
21. {clrscr();
22. C ob;
23. ob.show();
24. }
```

Figure 5.16 Code for detecting static anomalies in multiple inheritance.

In the above code, there is a static error because show() is present in both the parent classes A,B and there is show1() present in the child class C. Now it become ambiguous to which show() of the parent classes should be executed because now there is show1() in the child instead of show(). So, there is a static ambiguity at line 13 and the code will not be executed.

### 5.2.5 Multiple Inheritance with Anomalies Removed

Consider the following program in figure 5.17 using multiple inheritance where the anomalies are removed:

```
1.  #include<iostream.h>
2.  #include<conio.h>
3.  class A
4.  {public: int a;
5.  void show()
6.  {a=10;
7.  cout<<"\n a="<<a;
8.  }};
9.  class B
10. {public: int b;
11. void show()
12. {b=20;
13. cout<<"\n b="<<b;
14. }};
15. class C:public A, public B
16. {public: int c;
17. void show()
18. {cout<<"\n c="<<a+b;
19. }};
20. void main()
21. {clrscr();
22. C ob;
23. ob.A::show(); // calling class A show
                    1.  // function
24. ob.B::show();  // calling class B
                    1.  // show function
25. ob.C::show(); // successful giving the
                    1.  // result
26. }
```

Figure 5.17 Code in multiple inheritance with anomalies removed

In the above code, there is no error. Both the run-time and static anomalies are removed because of specifying when and to which class show() function to be called. Line number23,24,25 specifies the solution to the anomalies. Here show() is present in all the classes and no show1() function in any class. Now show() is called by specifying the object of specified class. So, it is the successful running of the code with no errors.

### 5.2.6 Test Results for multiple inheritance

Now, there is one case study in which there is one base class as library having data members as bid, rno and member function as get(), other base class as canteen having data member function as get1() and one derived class as student having data members as sname and member function as display() as shown in Figure 5.18.

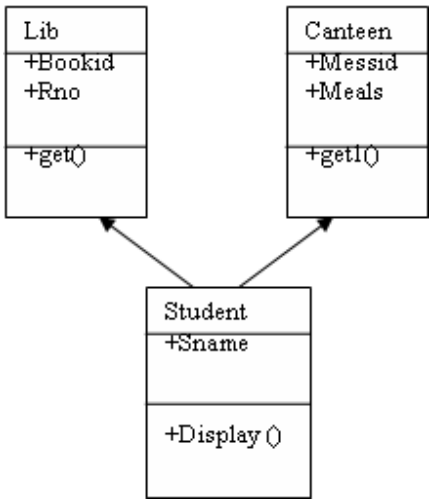


Figure 5.18 Example of multiple inheritance

In the case study of Student, Lib and Canteen of multiple inheritance discussed above. Now test cases will be generated. For that l an object of Library function, c an object of Canteen class and s an object of Student class is created. Bid is book id, rno is rollno, sname is student name. Values taken- bid=1,rno=22,messid=20,meals=bread,sname=anu.

#### 5.2.6.1 Unit Testing

In unit testing, there is testing of all the classes with their objects. And all classes are getting pass status because functions and data members are initialized in them and displayed in the same class. The case of display function in student class (which is derived of the get and get1 base classes). It is displaying the correct result of sname (i.e. student name) and other parameters as garbage because will be inheriting their features in the later part. The Test results are shown below in table 5.3.

Table 5.3 Test cases of unit testing in multiple inheritance

| Test case ID | Sequence call | Expected value | Actual value | Status |
|---|---|---|---|---|
| 1 | 1.get() | bid=1,rno=22 | bid=1,rno=22, | pass |
| 2 | c.get1() | messid=20,meals=bread | messid=20,meals=bread | pass |
| 3 | s.dispaly() | Rno=gv,sname=anu,messid=gv,bid=gv | Rno=gv,sname=anu,messid=gv,bid=gv | pass |

### 5.2.6.2 Integration Testing

In this only two cases are passing out of six because of calling two in correct sequence other in incorrect sequence which will give garbage value. Example as in table 5.4 calling Display() function first and Get() function later then obviously output will be the garbage value. It is a human mistake. But in case if handling a larger project in that case detecting such a mistake can be the difficult task. And if objects are not reduced or changed and calling sequence is with the previous object similar to unit testing then a lot of anomalies will be there.

Table 5.4 Test cases of integration testing in multiple inheritance

| Test case ID | Sequence call | Expected value | Actual value | Status |
|---|---|---|---|---|
| 1 | l.get()->s.display() | bid=1,rno=22,rno=22,sname=anu,messid=gv,bid=1 | bid=1,rno=22,mo=gv,sname=gv,messid=gv,bid=gv | fail |
| 2 | c.get1()->s.display() | messid=20,meals=bread,sname=anu,rno=22,messid=gv,bid=1 | messid=20,meals=bread,rno=22,sname=anu,messid=gv,bid=1 | fail |
| 3 | s.display()->l.get() | bid=1,rno=22,bid=1rno=22,sname=anu,messid=gv,bid=1 | bid=1,rno=22,bid=1,mo=gv,sname=gv,messid=gv,bid=gv | fail |
| 4 | s.display()->c.get1 | rno=22,sname=anu,messid=gv,bid=5,messid=20,meals=bread | rno=gv,sname=gv,messid=gv,bid=gv,messid=20,meals=bread | fail |
| 5 | s.get()->s.display() | bid=1,rno=22,rno=22,sname=anu,messid=gv,bid=5 | bid=1,rno=22,mo=22,sname=anu,messid=gv,bid=1 | pass |
| 6 | s.get1()->s.display() | messid=20,meals=bread,sname=anu,rno=22,messid=20,bid=1 | messid=20,meals=bread,sname=anu,rno=22,messid=20,bid=1 | pass |

### 5.2.6.3 System Testing

In system testing also, there are two test cases having pass status. Others are displaying garbage value because of wrong function call as shown in table 5.5.

Table 5.5 Test cases of system testing in multiple inheritance

| Test case ID | Sequence call | Expected value | Actual value | Status |
|---|---|---|---|---|
| 1 | s.get()->s.get1()->s.display() | bid=1,rno=22,bid=1,rno=22,sname=anu,messid=4,meals=bread,bid=1,bid=1,rno=22,sname=anu,messid=4,meals=bread | bid=1,rno=22,bid=1,mo=22,sname=anu,messid=4,meals=bread,bid=1,bid=1,mo=22,sname=anu,messid=4,meals=bread | pass |
| 2 | s.get1()->s.get()->s.display() | messid=4,meals=bread,bid=1,rno=22,bid=1,bid=1,rno=22,bid=1,rno=22,sname=anu,messid=4,bid=1 | bid=1,rno=22,bid=1,mo=22,sname=anu,messid=4,meals=bread,bid=1,bid=1,mo=22,sname=anu,messid=4,meals=bread | pass |
| 3 | s.display()->s.get()->s.get1() | messid=4,meals=bread,bid=1,rno=22,bid=5,bid=1,rno=22,bid=5,rno=22,sname=anu,messid=4,bid=1 | bid=1,rno=22,bid=1,mo=22,sname=anu,messid=4,meals=bread,bid=1,bid=1,mo=22,sname=anu,messid=4,meals=bread | fail |
| 4 | s.display()->s.get1()->s.get() | messid=4,meals=bread,bid=1,rno=22,bid=1,bid=1,rno=22,bid=5,rno=22,sname=anu,messid=4,bid=1 | bid=1,rno=22,bid=1,mo=22,sname=anu,messid=4,meals=bread,bid=1,bid=1,mo=22,sname=anu,messid=4,meals=bread | fail |
| 5 | s.get()->s.display()->s.get1() | messid=4,meals=bread,bid=1,rno=22,bid=1,bid=1,rno=22,bid=5,rno=22,sname=anu,messid=4,bid=1 | bid=1,rno=22,bid=1,mo=22,sname=anu,messid=4,meals=bread,bid=1,bid=1,mo=22,sname=anu,messid=4,meals=bread | fail |
| 6 | s.get1()->s.display()->s.get() | messid=4,meals=bread,bid=1,rno=22,bid=1,bid=1,rno=22,bid=1,rno=22,sname=anu,messid=4,bid=1 | bid=1,rno=22,bid=1,mo=22,sname=anu,messid=4,meals=bread,bid=1,bid=1,mo=22,sname=anu,messid=4,meals=bread | fail |

### 5.2.7 Anomalies identified from multiple inheritance

- An error comes in integration testing that is calling an object with base class object and then using other function with derived class object. Example is shown in Figure 5.19.

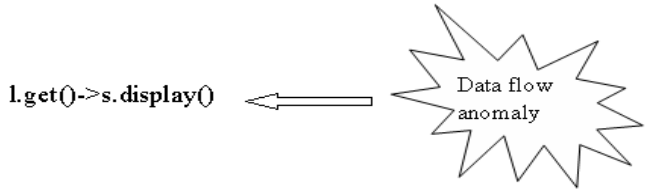


Figure.5.19 Data flow anomaly 1

- Figure. 20 shows a case in which base class does not calls a derived class but a derived class can always call a base class.

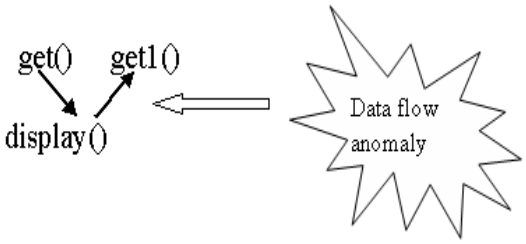


Figure 5.20 Data flow anomaly 2

- When a base class calls other base class in multiple inheritance , an anomaly occurs (Figure 5.21).

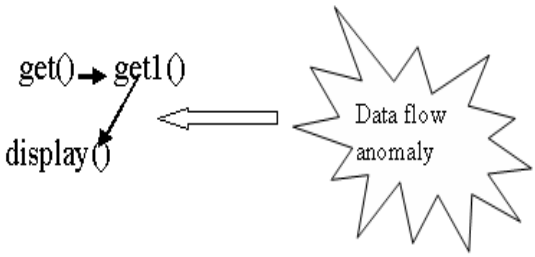


Figure.5.21 Data flow anomaly 3

- No anomaly: When both the base classes are inherited in derived class and they are calling with the required object or with the derived class object as stated in Figure 5.22.

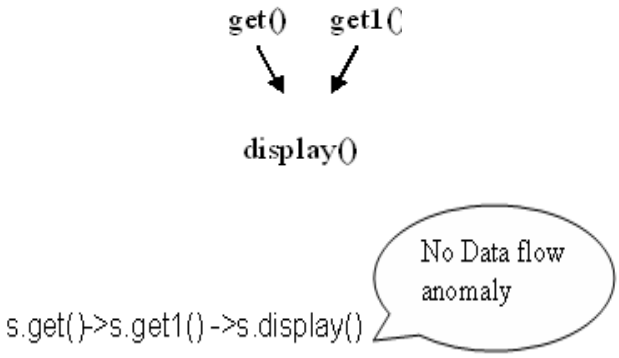


Figure.5.22 No Data flow anomaly

The above cases indicate that correct sequence call is essential for successful inheritance. A model of Do's and Don'ts in multiple inheritance is proposed here.

After defining the classes testing is required. Firstly start with unit testing, then with integration testing and then system testing so as to avoid various errors and anomalies.

## 5.3 Multilevel Inheritance

In multilevel inheritance, we have various levels which are defined in which anomaly comes with method overriding or by giving wrong function call.

### 5.3.1 Implementation details for multilevel inheritance

In figure 5.23 an implementation example showing accessibility in multilevel inheritance.

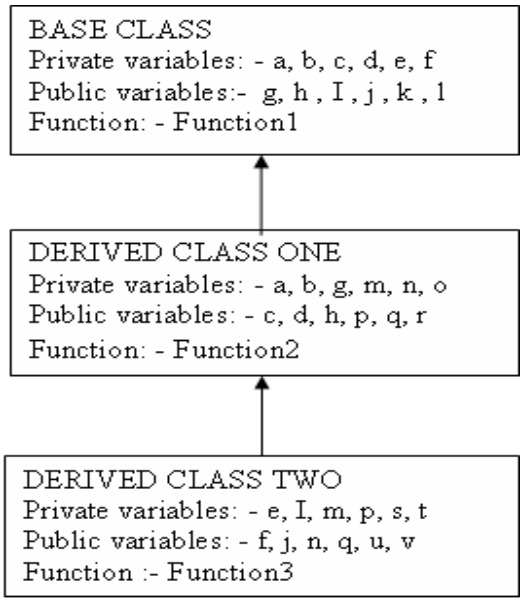


Figure 5.23 Multilevel inheritance example.

#### 5.3.1.1 Accessing variable with an object

After testing the few errors occurred and those are explained below.

**CASE 1: When we try to access the variables which are defined in other classes (derived one and derived two classes) but not defined in base class**

There are variables : m , n , o , p , q , r , s , t , u , v which are not the member of base class or not defined in base class, when we try to access those variables by base class object the following error in figure 5.24 occurred .


Figure 5.24 Base classes cannot access any other base class or derived class

**CASE 2: when we try to access the variables which are private member of base class by derived one class object.**

In this case when we try to access the private members of the base class which is not defined in derived one class with the derived one class object then the following error in figure 5.25 will occur.

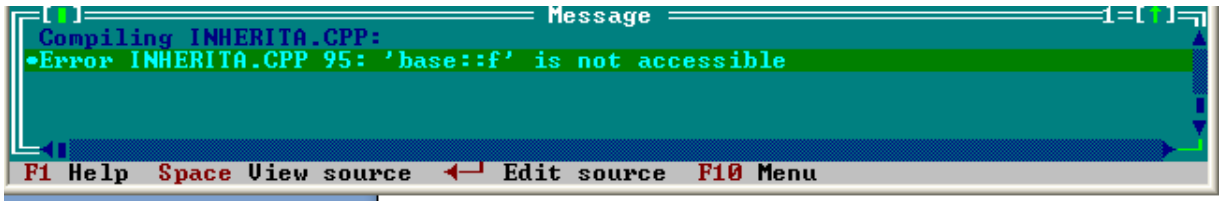


Figure 5.25 Derived class cannot access private variable of base class

The screen shots of the above cases are shown below in figure 5.26.

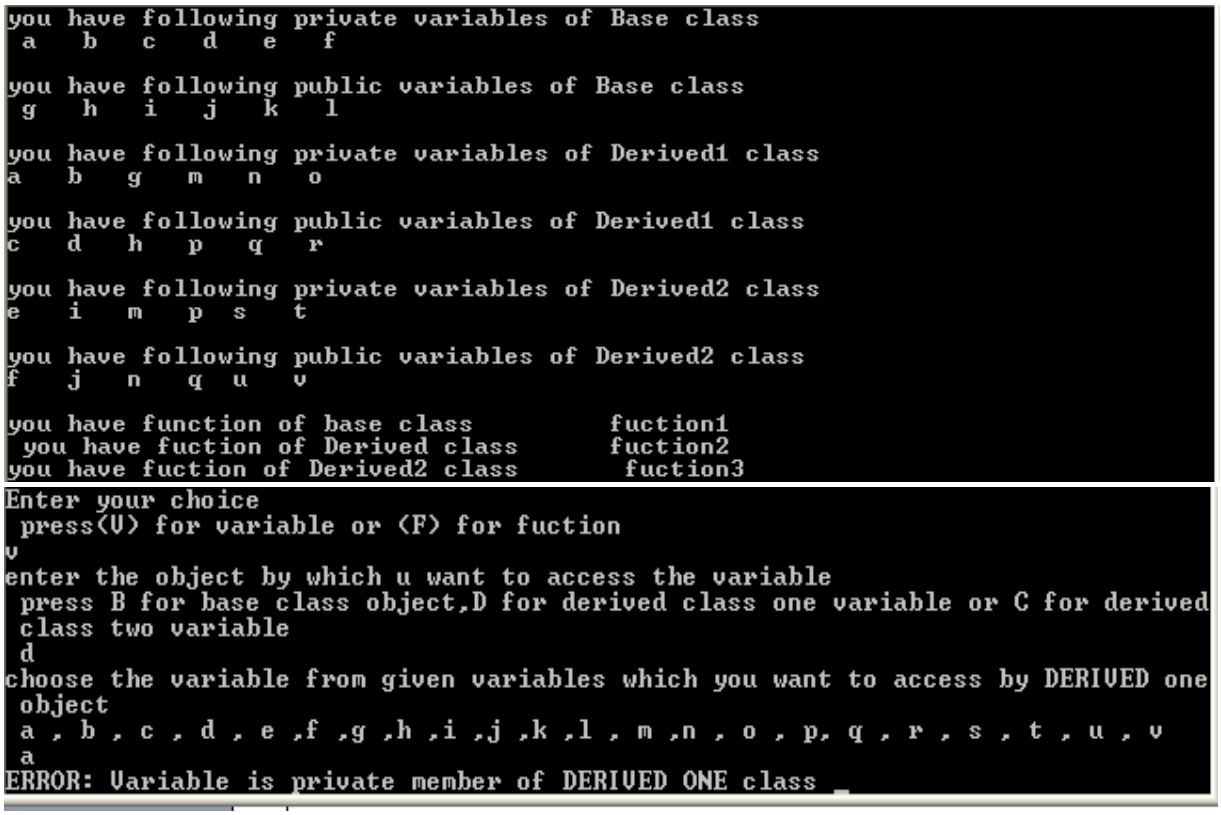


Figure 5.26 Screen shots of accessing variable with an object

### 5.3.1.2 Accessing function with objects

Here we are experimenting with the three functions **Function1**, **Function2, function3.** Function1 belongs to the base class, Function2 belongs to the derived one class and function3 belongs to derived2 class. In this part we will access all the variables of all classes one by one and with different class objects (base object and both derived class object). Different cases will be obtained and different error which will be classified below.

**CASE 1: When a base class object is accessing a function 1 and function1 is accessing the variables which are not defined in base class**

In figure 5.27, the variables m, n, o, p, q, r, s, t, u, v are those variables which are not defined in base class. When function 1 which is defined in base class want to access these variables the following error will be occurred.

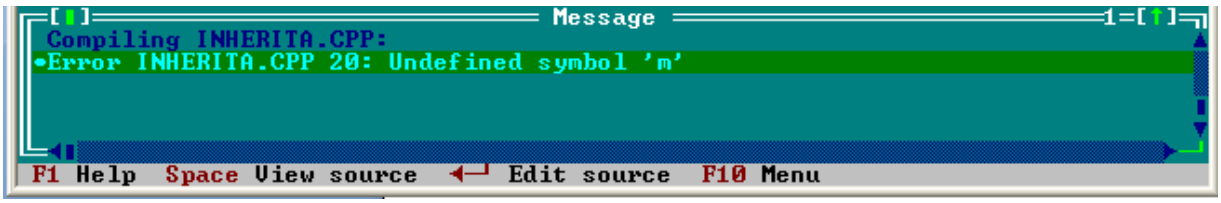


Figure 5.27 Base class function accessing variables which are not defined in it.

**CASE 2: When a derived one class object wants to access the private variables of base class via FUNCTION2**

The variables “e, f” are the variables which are not present in derived one class and are private member of base class. In figure 5.28, when the derived one class object want to access these two variables via FUNCTION 2 the following error will come**.**

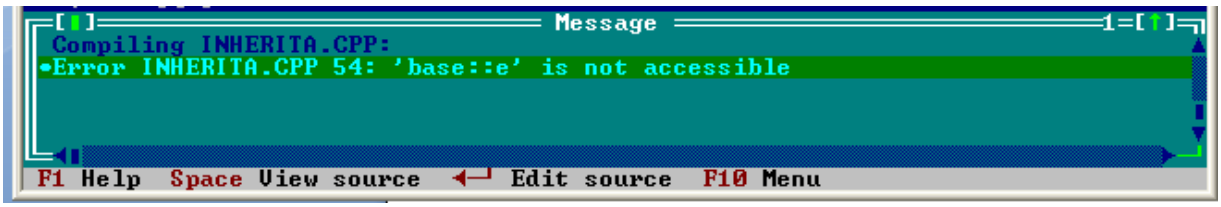


Figure 5.28 Derived class accessing private variable of base class

The screen shots of the above cases are shown below in figure 5.29.

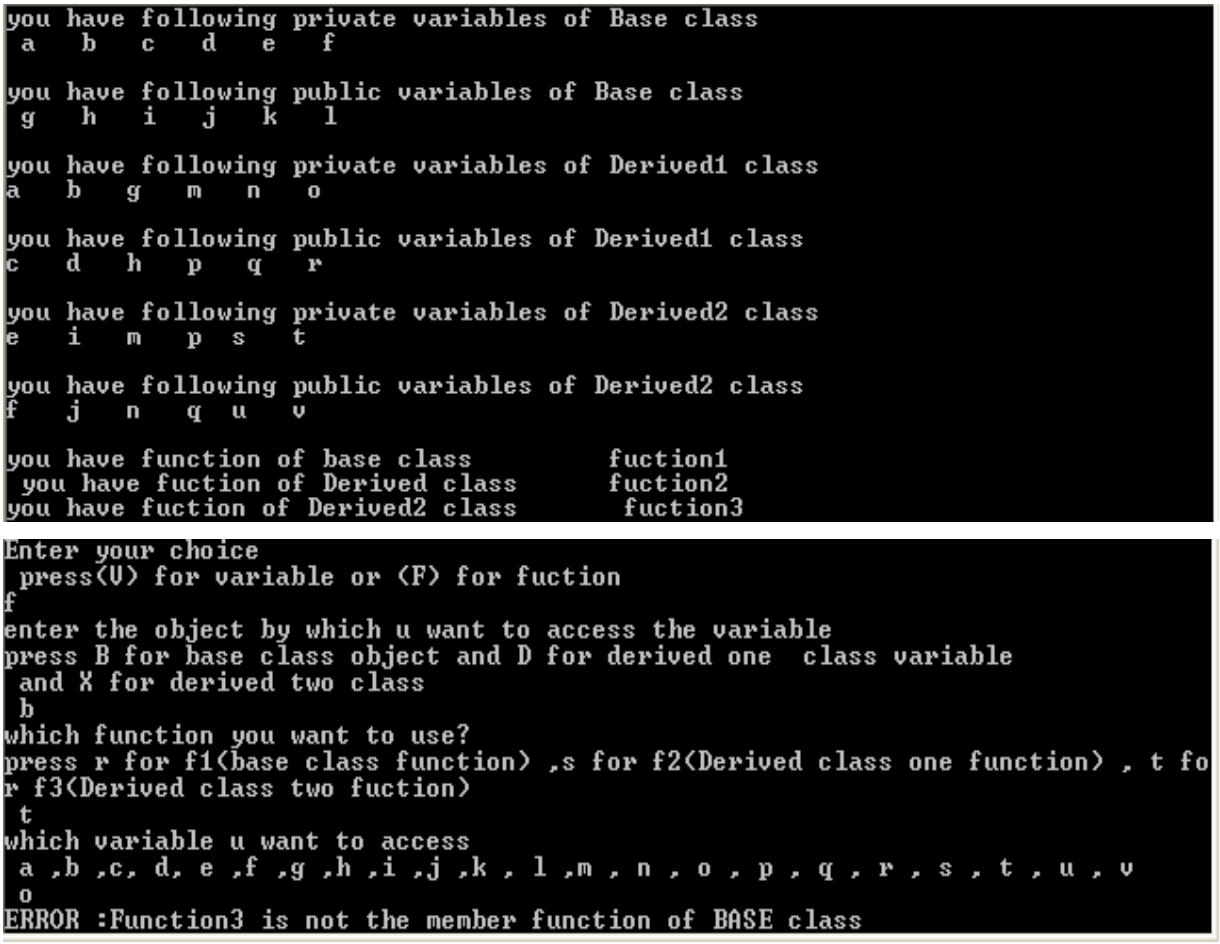


Figure 5.29 Screen shots of accessing variable using function

### 5.3.2 Program to implement multilevel inheritance in C++

Table 5.6 showing various private, public and functions of base and derived class. Then the cases are given showing the accessibility of variable and functions

In Figure 5.30, there are similar function names and three different objects. And there comes an anomaly when a function is called with different object name.

```
1.  #include<iostream.h>
2.  #include<conio.h>
3.  class First
4.  {public:int q,r,s;
5.  void a()
6.  {}
7.  void b()
8.  {q=10;
9.  s=20;}
10. void c()
11. {cout<<"\n q=10    "<<q;}
12. void d()
13. {r=30;
14. cout<<"\n s=20   "<<s;
15. }
16. void e()
17. {cout<<"\n r=30    "<<r;
18. }};
19. class Second:public First
20. {public:int t;
21. void b()
22. {t=50;
23. }
24. void c()
25. {cout<<"\n t=50   "<<t;
26. }};
27. class Third:public Second
28. {public:int u;
29. void c()
30. {u=60;
31. }
32. void d()
33. {cout<<"\n u=60    "<<u;
34. }};
35. void main()
36. {clrscr();
37. First F;
38. Second S;
39. Third T;
40. F.a();
41. F.b();
42. F.d();
43. F.d();
44. F.e();
45. S.b();
46. S.c();
47. T.c();
48. T.d();
49. S.c();
50. T.d();
51. F.d();
52. F.e();
53. };
```

Figure 5.30 Code for detecting anomalies in multilevel inheritance

.

### 5.3.3 Test Results for Multilevel inheritance

Here are the test cases shown in table 5.6 for the multilevel inheritance which is having the F, S, and T the objects of First, Second and Third class as discussed above. In order to test the proposed algorithm, various test cases were generated and executed and are summarized in table 5.6

Table 5.6 Test cases of multilevel inheritance

| Test Case ID | Sequence call | Expected() value | Actual value | Status |
|---|---|---|---|---|
| 1. | F.a()->F.b()->F.c()->F.d()->F.e() | q=10,s=20,r=30 | q=10,s=20,r=30 | pass |
| 2. | S.b()->S.c() | t=50 | t=50 | pass |
| 3. | T.c()->T.d() | u=60 | u=60 | pass |
| 4. | F.a()->F.b()->F.c()->F.d()->F.e()->S.b()->S.c() | q=10,s=20,r=30,t=50 | q=10,s=20,r=30,t=50 | pass |
| 5. | F.a()->F.b()->F.c()->F.d()->F.d()->F.e()->S.b()->S.c()->T.c()->T.d() | q=10,s=20,r=30,t=50,u=60 | q=10,s=20,r=30,t=50,u=60 | pass |
| 6. | F.a()->S.b()->T.c()->T.d() | u=60 | u=60 | pass |
| 7. | F.a()->F.b()->S.c()->T.c()->T.d() | t=50,u=60 | T=50,u=60 | pass |
| 8. | F.a()->F.b()->S.b()->T.c()->T.d() | u=60 | u=60 | pass |
| 9. | F.a()->F.b()->S.b()->S.c()->T.c()->T.d() | t=50,u=60 | T=50,u=60 | pass |
| 10. | F.a()->F.b()->S.b()->F.c()->F.d()->F.e() | S=20,r=30 | S=20,r=30 | pass |
| 11. | F.a()->F.b()->S.b()->S.c()->F.c()->F.d()->F.e() | T=50,q=10,r=30,s=20 | T=50,q=10,r=30,s=20 | pass |
| 12. | F.a()->S.b()->F.b()->F.c()->F.d()->F.e() | q=10,s=20,r=30 | q=10,s=20,r=30 | pass |
| 13. | F.a()->F.b()->S.c() | t=50 | T=gv | fail |
| 14. | F.a()->F.b()->S.c()->T.d() | t=50,u=60 | T=gv,u=gv | fail |
| 15. | F.a()->F.b()->S.c()->T.c()->T.d() | T=50,u=60 | T=gv,u=60 | fail |
| 16. | F.a()->F.b()->S.b()->S.c()->T.c()->T.d() | T=50,u=60 | T=50,u=60 | pass |
| 17. | F.a()->F.b()->S.c()->T.d() | T=50,u=60 | T=gv,u=60 | fail |
| 18. | F.a()->F.b()->F.c()->S.c()->T.d() | q=10,t=50,u=60 | q=10,t=gv,u=gv | fail |
| 19. | F.a()->F.b()->F.c()->S.c()->T.c()->T.d() | q=10,t=50,u=60 | q=10,t=gv,u=60 | fail |
| 20. | F.a()->S.b()->S.c()->F.c()->F.d()->F.e() | q=10,t=50,s=20,r=30 | q=gv,t=50,r=30,s=gv | fail |
| 21. | F.a()->S.b()->S.c()->T.c()->T.d()->F.e() | r=30,t=50,u=60 | r=gv,t=50,u=60 | fail |
| 22. | F.a()->F.b()->S.b()->S.c()->T.c()->T.d()->F.e() | r=30,t=50,u=60 | r=gv,t=50,u=60 | fail |
| 23. | F.a()->S.b()->S.c()->T.c()->T.d()->F.d()->F.e() | r=30,t=50,u=60,s=20 | r=30,t=50,u=60,s=gv | fail |
| 24. | F.a()->F.b()->S.b()->S.c()->T.c()->T.d()->F.d()>F.e() | r=30,t=50,u=60,s=20 | r=30,t=50,u=60,s=20 | pass |
| 25. | F.a()>F.b()->F.c()->S.b()->T.c()->S.c()->F.d()->F.e() | r=30,t=50,s=20,q=10 | r=30,t=50,s=20,q=10 | pass |
| 26. | F.a()>F.b()->F.c()->S.b()→T.c()>T.d()->F.d()->F.e() | r=30,u=60,s=20,q=10 | r=30,u=60,s=20,q=10 | pass |
| 27. | F.a()->S.b()->S.c() ->F.d()->F.e() | r=30,t=50,s=20 | r=30,t=50, s=gv | fail |
| 28. | F.a()->F.b()->T.c()->T.d()->F.d()->F.e() | u=60,s=20,r=30 | U=60,s=20,r=30 | pass |
| 29. | F.a()->F.b()->T.c()->T.d()->S.c()->S.d() | u=60,s=20,r=30 | u=60,s=gv,r=30 | fail |
| 30. | F.a()->S.b()->F.c()-F.d() | q=10=,s=20 | Q=gv,s=gv | fail |
| 31. | F.a()->S.b()->T.c()->S.c()->F.c()->T.d() | t=50,q=10,u=60 | T=50,q=gv,u=10 | fail |

### 5.3.4 Findings in multilevel inheritance

Figure 5.31 shows the class diagram of multilevel inheritance. It shows the various state various and methods present in respective classes [20].

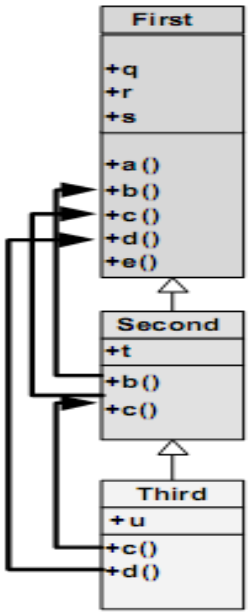


Figure 5.31 Class diagram of multiple inheritance [20]

The table 5.7 shows the state variable definition and uses of the methods for each class hierarchy i.e. which state variables are defined in which methods and used in which methods. Here a(),b(),c(),d(),e() are methods. q,r,s,t,u are state variables. F,S and T are the objects of the First, Second and Third class [20].

Table 5.7 Anomalies in multilevel inheritance

| Method | Definitions | Uses |
|---|---|---|
| F::b | {F::q,F::s} | |
| F::c | | F::q |
| F::d | {F::r} | {F::s} |
| S::b | {S::t} | |
| S::c | | {S::t} |
| T::c | {T::u} | |
| T::d | | {T::u} |

Figure 5.32 depicts the flow control of methods when c() is bound to be an instance of First. Suppose that a() calls b(), b() calls c(), c() calls d() and so on.

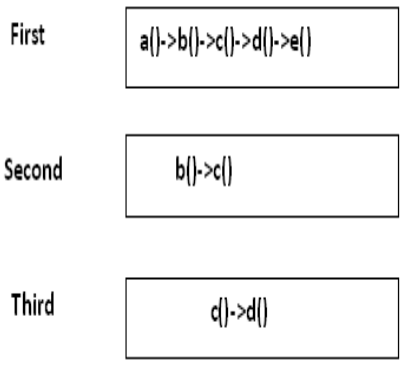


Figure 5.32 Sequence call of methods

Now suppose that calls to method F::b() precedes the call of F::c() and F::d(). Table 1 shows the definitions uses table that shows F::q and F::s are defined by F::b() and used by F::c() and F::d(). So here is no data flow anomaly because it is firstly defining the variables then using it.

Now suppose here is an innocuous call to S.b() instead of F.b(). Then data flow anomaly would exist because according to definitions uses S.b() has called to S::t. Here there is no problem but after S.b(), F::c() and F::d() are called. Then it creates data flow anomaly as shown in Figure 5.33 because F::c() uses F::q and F::d() uses F::s. They are used before they are defined [20].

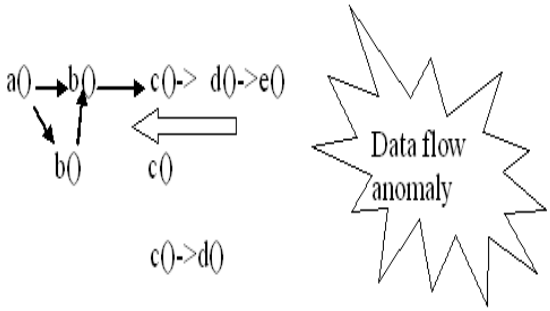


Figure 5.33 Data flow anomaly1

Now suppose F.a() is called then S.b() and then T.c() called S.c(), S.c() called F.c() and F.c() called T.d(). It creates data flow anomaly because according to definitions uses S.b() has called to S::t and T::c() is defining T::u then calling S::c. There is no problem but after S.c(), F::c() is called that creates data flow anomaly because F::c() uses F::q. This anomaly is shown in Figure 5.34.

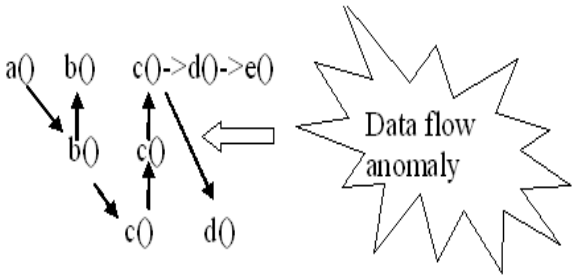


Figure 5.34 Data flow anomaly2

In results and findings various anomalies in single inheritance are found in which firstly a code is written and anomalies are detected by running the code from each possible way of calling functions and store its results in the test cases written showing the status pass or fail. Similarly code is written for multiple inheritance and detected its anomalies by unit testing, integration testing and system testing and storing its results in test cases written for them separately. And same for multilevel inheritance in which various overridden functions are there in various classes and results are written in test cases.

# Chapter 6 Conclusion

*According to **Carl Sagan**, "Knowing a great deal is not the same as being smart; intelligence is not information alone but also judgment, the manner in which information is collected and used." [31]*

There are two different types of errors occur such as static and dynamic in Object Oriented Programming. Static are the real time syntax errors that come in the programming code where as Dynamic errors are called anomalies which occur during run time and give garbage values in the result. It is a tedious task to trap such anomalies. Here an approach is discussed to detect anomalies in the object oriented systems so that the given software becomes error free, give reliability to the software and then software comes with the good quality. Various reasons for anomalies in object oriented systems are discussed below.

## 6.1 Causes of anomaly

1. Probability of anomalies increases with increase in number of objects and number of overridden functions and number of base classes
2. Anomaly occurs when there is a function call with wrong type of object.

The results are indicated in table 6.1. The design of test cases indicate that with the increase in number of objects and functions number of test cases increases which will result in giving wrong output if sequence call is not desired in single inheritance.

1. In Multiple inheritance, in unit testing, separate modules are being tested. And there is one function in each of the methods. So there is anomaly in the derived class which is dependent on base class.

2. In Multiple inheritance, with integration testing, number of objects are 3 and number of functions are 2 and there are 6 test cases and 4 have status as fail. Because numbers of functions are increasing and number of objects are not decreased.

3. In Multiple inheritance, with system testing, number of objects as 1 i.e. number of objects are reduced either there is increase in number of objects. So, the test cases generated is 6 out of which 4 has fail status.

Similarly with the Multilevel inheritance, number of objects is 3 and number of functions are 9, and there are 14 test cases of fail status out of 31.

Table 6.1 Case studies in inheritance

| Case study | No.of objects | No.of functions to be tested | No. of anomalies | No. of test cases | Reason for anomaly |
|---|---|---|---|---|---|
| Single inheritance | 3 | 5 | 29 | 30 | Wrong function call gives garbage values. |
| Multiple inheritance( unit testing) | 3 | 1per class | 1 | 3 | Inheritance dependency of functions is there. |
| Multiple inheritance( integration testing) | 3 | 2 | 4 | 6 | Function should be called with the desired object only because value is retained in the object not in function or class |
| Multiple inheritance( system testing) | 1 | 3 | 4 | 6 | If there is a dependency or inheritance, then we should call base class first then derived class. Or we should call functions in the required sequence. Here, we have less test cases and anomalies because we have reduced no. of objects to 1 instead of 3. |
| Multilevel inheritance | 3 | 9 | 14 | 31 | Sometimes Overridden function becomes problematic to handle if they are defined separately in each class. |

From the table 6.1, on comparing the single and multilevel inheritance both have number of objects as 3 but number of functions vary but it doesn't make effect over number of test cases because complexity increases with the number of objects than the number of function calls.

## 6.2 Solutions for handling anomaly

In case of multiple inheritance, if same function name in the base classes exists and in order to resolve the ambiguity then functions should call as **obj.A::display()** for A class display **Obj.B::display()** for B class display i.e. the name of the class should be added with member function to avoid anomaly.

Now, there are do's and don'ts for multiple inheritance shown below in figure 6.1

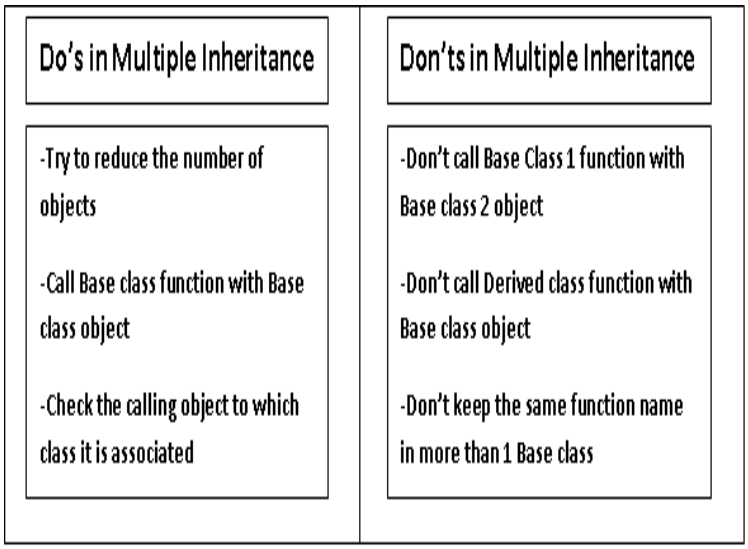


Figure 6.1 Do's and don'ts in multiple inheritance

In Multilevel inheritance complexity will increase if number of levels is increasing. And if the base is an abstract class and some functions will be added later on then also changing the base class properties can affect the derived class functionality because it leads to regression testing if there is a change or added one of the properties.

Regression testing attempts to mitigate two risks:

- A change that was intended to fix a bug failed.
- Some change had a side effect, unfixing an old bug or introducing a new bug.

In future, these algorithms can be embedded with VC++ compliers and with other programming languages which can help developers to get rid of such problems so that software and projects should become reliable, and efficient.

# References

[1] A. Andrews, R. France, S. Ghosh, and G. Craig, “Test Adequacy Criteria for UML Design Models”, Software Testing, Verification, and Reliability Journal, Vol. 13, No. 2, pp. 24-30, June 2003.

[2] Atanas. Rountev, Ana. Milanova, Barbara G. Ryder, “Fragment Class Analysis for Testing of Polymorphism in Java Software”, IEEE Transaction on Software Engineering, Vol. 30, No.6, pp.372-387, June 2004.

[3] Beizer. Boris, Software Testing Techniques, Second Edition, Van Nostrand Reinhold, 1990.

[4] Bucanac. Christian, “Object oriented testing report”, Software verification and validation, Version. 0.2, pp. 3-7, December 1998.

[5] Clay E. Williams “Software Testing and the UML” ,International Symposium on Software Reliability Engineering (ISSRE’99), Boca, Raton ,pp. 1-2,November 1999.

[6] F. Fraikin and T. Leonhardt, “SeDiTeC - Testing Based on Sequence Diagrams”, Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002), Edinburgh, Scotland, UK, pp. 23-27, September 2002.

[7] H. Y. Chen, T H. Tse, and T. Y. Chen, “TACCLE: A Methodology for Object-Oriented SoftwareTesting at the Class and Cluster Levels”, ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 4, pp. 56–109, January 2001.

[8] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen, “In Black and White: An Integrated Approach to Class-Level Testing of Object-Oriented Programs”, ACM Transactions on Software Engineering and Methodology, Vol. 7, No. 3, pp. 250–295, July 1998.

[9] H. Y. Chen, “An Approach for Object-Oriented Cluster-Level Tests Based on UML”, Proceeding of IEEE International Conference on Systems, Man, and Cybernetics, Hyatt Regency, Washington, D.C., USA, Vol. 2, pp. 1064-1068, 5-8 Oct. 2003.

[10] K. C. Tai and F. J. Daniels, “Test Order for Inter-Class Integration Testing of Object-Oriented Software”, Proceedings of the 21st Annual International Computer Software and Applications Conference (COMPSAC '97), Washington, DC, USA, pp. 602-607, August 13-15, 1997.

[11] L. Briand and Y. Labiche, “A UML-Based Approach to System Testing”, Journal of Software and System Modelling, Volume 1, Issue 1, pp. 19-30, Springer, 2002.

[12] L. Souter and L. L. Pollock, “OMEN: A Strategy for Testing Object-Oriented Software”, Proceedings of the International Symposium on Software Testing and Analysis, Portland, Oregon, United States, pp. 49–59, 2000.

[13] Lv Ge-Feng, Zou Bei-Ji, Zhou Hao-Yu, Sun Jia-Guang, “Research on Model of Automated Test Case Generation for Complicated Interactive Software,” Mini-Micro System ( in Chinese) , Vol.1, No. 27, pp.131-135, 2006.

[14] L. Zhao, “A new approach for software testability analysis”, In Proceeding of the 28th international Conference on Software Engineering, Shanghai, China, ICSE '06, ACM Press, New York, NY, pp.985-988, May 20 - 28, 2006.

[15] McGregor D., John, A practical guide to testing object oriented systems, 2001.

[16] Meyer, B., Object-Oriented Software Construction. Prentice Hall, second ed., pp. 136-169, Apr. 1997.

[17] M. Stannette, “A Bibliography of Object-Oriented Testing, with a Short General Survey of the Area”, Research Memorandum, http://www.dcs.shef.ac.uk/research/2002.

[18] Gordon Fraser, Franz Wotawa, “Mutant minimization for model-checker based test-case generation”, Testing: Academic and Industrial Conference Practice and Research Techniques –TAICPART-MUTATION 2007, pp. 161–168, 2007.

[19] Pressman, R.S. Software Engineering: A Practitioner’s Approach. Mc Graw Hill, second ed., 2005.

[20] R. Alexander and J. Offutt.,” Criteria for Testing Polymorphic Relationships”, In Proceedings of the 11th international Symposium on Software Reliability Engineering (ISSRE'00) ISSRE. IEEE Computer Society, Washington, DC, pp.15-23, October 08-11, 2000.

[21] R. Binder, “Testing object-oriented software: a survey”, Journal of Software Testing, Verification and Reliability, Vol. 6, No. 3-4, pp.125–252, 1996.

[22] Roger. T. Alexander, “Testing polymorphic relationships of object oriented programs”, A dissertion, pp23-31, Spring 2001.

[23] R. T. Alexander, J. Offutt, and J. M. Bieman, “Syntactic Fault Patterns in OO Programs”, Proceedings of the 8th International Conference on Engineering of Complex Computer Software (ICECCS '02), Greenbelt, MD, pp. 2-3, November 2002

[24] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: Object-Oriented Modeling and Design. Prentice Hall, 1991.

[25] R. V. Binder, Testing Object-Oriented Systems Models, Patterns, and Tools, Addison-Wesley, NY USA, 1999.

[26] Saini D.K, “Testing Polymorphism in Object Oriented Systems for Improving software Quality”, - ACM SIGSOFT Software Engineering Notes, Vol. 34, No. 2, pp.1-5, March 2009.

[27] S Supavita , Object-Oriented Software and UML-Based Testing: A Survey Report, , 2009.

[28] Stroustrup, B., “Multiple Inheritance for C++”, The C/C++ Users Journal, pp. 2-15, May 1999.

[29] Xu Shenghong, Cao Wenjing, “A Software Function Testing Method Based on Data Flow Graph”, International Symposium on Information Science and Engineering, Vol. 2, pp28-31, 2008.

[30] Quotations on software testing http://hexawise.wordpress.com/2010/01/26/25-great-quotes-for-software-testers/

[31] Quotes for software testers http://blog.utest.com/16-great-quotes-for-software-testers/2011/01/

[32] Software testing quotations http://gsehl.editme.com/TestingQuotes

[33] UML Modeling http://agilemodeling.com

# List of Publications

[1] Kaur. Shubpreet, Goel. Shivani, "Testing Anomalies in Multiple and Multilevel Inheritance", published in International Journal of Computers and Communications (IJCC), Vol.1, No.1, pp. 1-14, June 2011.

[2] Kaur. Shubpreet, Goel. Shivani, "Detecting anomalies during Multiple Inheritance", published in Journal of Global Research in Computer Science (JGRCS), Vol. 2, No. 5, pp. 37-42, June 2011.