CHAPTER 6

TESTING OF INHERITANCE RELATED BUGS
IN OBJECT ORIENTED PROGRAMS

6.1 INTRODUCTION

The overall goal of testing is to gain confidence in the correctness of the program. Testing of object oriented program poses challenges due to presence of property bugs in programs (Binder 1999) and (David Hovemeyer and William Pugh 2004).

This work aims at detecting the bugs with reference to the inheritance property. Based on this concept, the class can inherit property from more than one level. The inheritance can be classified into Single Inheritance, Multiple Inheritance (Willis 1996), Multilevel Inheritance, Hierarchical Inheritance and Hybrid Inheritance. The implementation of inheritance in different object oriented languages such as C++, Smalltalk and Objective-C may seem to be similar but there are many differences to be considered. Java does not support multiple inheritances as is supported by C++. To implement multiple inheritances Java uses interfaces. The inheritance in Java is basically carried by using the keyword “extends” (Schildt 2001).
Derivation of a class from another derived class is called multilevel inheritance. In object oriented life cycle, inheritance related bugs may occur in two phases (i) **Design Phase** and (ii) **Implementation Phase**. By using data flow information one can find **Implementation Phase Inheritance Related** bugs such as Incorrect Initialization, Naked Access, Inadvertent Bindings, Missing Override, Naughty Children and Spaghetti Inheritance. Design phase inheritance related bugs are Square Peg in a Round Hole, Worm Holes, Gnarly Hierarchies, Weird Hierarchies and Fat Interface. This thesis mainly focuses on **Implementation Phase Inheritance Related Bugs** due to multilevel inheritance in Java programs. The outputs are inheritance related bugs detected with the help of this testing tool.

### 6.2 TYPES OF IMPLEMENTATION PHASE INHERITANCE RELATED BUGS

Implementation phase inheritance related bugs are explained below:

**Incorrect Initialization:** It occurs when a base class initialization is missed in its definition and child class constructor calls the initialization of base class, but there is no such kind of constructor in the base class. The deep level of inheritance hierarchy lead to incorrect initialization bugs.

**Example**

```java
Class shape
{
    Int height;
    Public
    Virtual void draw();
};
Class Rect: public shape
{
```

Public:
    Rect(int x, int y, int z) { 
        super(z);
        .......
    }
    Void draw();
    .......
};

Class Painting{
    Public static void main(){
        Rect r = new Rect(4,5,3); /* Incorrect initialization*/
        ......
    }
}

In the above example, the object for ‘Rect’ class is created in ‘main’ function of the ‘Painting’ class. The ‘shape’ class doesn’t have constructor, the ‘super’ cannot be initialized of its properties from the call of ‘Rect’ class. So this fragment has the bug of ‘Incorrect Initialization’. It can be identified at compile time of the program.

**Naked Access:** This is the case that a parent class attributes are visible in a child class and child class methods can modify those attributes directly. If there is any change in the attributes of subclass, it should not change the value of instance variable in super class. Changes in the parent class must be reflected upon in the child class also. If a base class is modified, both the base class and its subclasses should be retested. If the subclass is changed both the subclass and the super class features in the subclass should be retested.

**Example**
class Inher {
    int a;
    void getinput() {
        a = 10;
    }
    void output() {
        System.out.println("A= "+a);
    }
}

class First extends Inher {
    public void display() {
        a=a++;
        System.out.println("Thesis");
    }
}

class Testing {
    public static void main(String args[]) {
        First f= new First();
        f.display();
        f.output();
        f.getinput();
        f.output();
    }
}

From the above example, the base class ‘Inher’ has a data member such as ‘a’. When it is extended by ‘First’, the ‘First’ has updated the value of ‘a’ like ‘a++’.
Here is an example if unrestricted access of data member of the class ‘Inher’ with the variable ‘a’ declared in base class. The value of instance variable in derived class should not get changed when the instance variable has already been declared in base class.

**Inadvertent Bindings:** If a parent class data member and subclass data member have the same name, there will be a confusion on usage of the data members. Non private data or public data members of base class will be visible even in the child class also. But if the child class again declares a data member in the same name of the base class, then it leads to unexpected implementation related bug.

**Example**

```java
class Person{
    String address;
    void display(){
        address="salem";
        System.out.println("Address = "+address);
    }
}
class Student extends Person{
    String address;
    void output() {
        address="erode";
        System.out.println("Address = "+address);
    }
}
class Exceed{
    public static void main(String args[]){
        Student s = new Student();
    }
}
```
In the above example, the ‘Person’ base class which has a data member ‘address’ in ‘public’ access visibility is used for storing the native address of a person. The derived class ‘Student’ of ‘Person’ also has a data member in same name such as ‘address’, storing the address of school where the student studied. It makes confusion on usages of the same name used on different levels of Inheritance classes.

**Missing Override:** A child class specific implementation of a base class method is omitted, as a result, the base class method might be incorrectly bound to a child class object and a state could result that was valid for the base class but invalid for the subclass owing to a stronger subclass invariant. To solve this bug, the ‘virtual’ keyword can be used in base class function.

**Example**

```cpp
class Shape{
    public:
        void area(); /* missing virtual keyword*/
};

Class Circle:public Shape{
    ........
    public:
        void area();
    ........
};
```
If base and derived classes have member functions of same name will cause function overriding. In order to avoid function overriding, ‘virtual’ keyword has to be used before the method in base class.

**Naughty Children:** It is the case that a child class does not make use of any member function in the parent class but it has extended its scope to base class.

**Example**

class First{
    public int x;
    void getinput(int a){
        x = a;
    }
    void getcount(){
        x = x++;  
    }
    void display(){
        System.out.println("The value of x = "+x);
    }
}

class Second extends First{
    public int y=0;
    void access() {
        y=y+20;
        System.out.println("The value of y = "+y);
    }
}
class MainC{
    public static void main(String args[]){
        Second s= new Second();
        s.access();
    }
}

In the above example, the base class ‘First’ has two member functions for accessing its data member. The ‘First’ is extended by the class ‘Second’. But while execution of its member functions, the child class ‘Second’ is not used the member functions of it base class (First). So the derived class of ‘First’, the ‘Second’ is a naughty child.

**Spaghetti Inheritance:** A number of multilevel inheritances of more than 6 Levels (threshold value of level 6 is experienced by project experts in different projects) create complexity in subsequent classes (Binder 1999).

class Person{
    ............
}
class Student extends Person{
    ................
}
class Test extends Student{
    ............
}
class Grade extends Test{
    ................
class Employee extends Student{
    ..................
}

Class Payroll extends Employee{
    ..................
}

Class Loan extends Payroll{
    ..................
}

In the above example, if the levels of classes inherited goes beyond 6 levels, it will develop the bugs. This type bug is called Spaghetti bug. This can be avoided by splitting and using properties.

6.3 TYPES OF DESIGN PHASE INHERITANCE RELATED BUGS

Square Peg in a Round Hole: Newly derived class from the base class may not be fit to the current system. This type of bug may occur due to poor design of the software.

Example:
Figure 6.1  Square Peg in a Round Hole

The base class ‘Rectangle’ has a member function ‘resize(x,y)’ and it is inherited by the class ‘square’ for resizing the square window. A square needs only one value for its side. So the base class member function cannot be used by the child class. It is a problem may occur due to poor design.

Gnarly Hierarchies: The derived classes from the base classes may create a new boundary for other classes in the system. It may lead to shift from one domain to another domain of the system as shown in Figure 6.2.

Figure 6.2  Gnarly Hierarchies

From the above hierarchy, the system is to be designed for students. When it is to be extended for ‘Part-time’ student then the ‘employee’ details should be derived for the ‘part-time’ student subclass. So this may lead to derive so many details of an employee and it makes a shift from ‘student’ domain to ‘employee’ domain.
It is not necessary to create a separate class for deriving entire class of ‘employee’ for representing small information of employee. So this can be included in the class itself instead of deriving it from ‘employee’ class.

**Weird Hierarchies and Fat Interface:** Inheritance provides the facility of sharing the code on classes. If a class member function may be required by other class in the system then it may go for extending the properties of that class. But it is not recommended by testers for categorically fit.

**Example:**

```
Base:                  Subclass:
  Polygon: volume()       container
```

**Figure 6.3 Weird Hierarchies and Fat Interface**

Figure 6.3 shows the inherited member function from ‘Polygon’ class may be used by the ‘Container’ subclass. It is only for finding the volume of the container to fill it. In order to derive only one method ‘volume()’, the ‘container’ class inherit all member functions from ‘Polygon’ class. It unnecessarily shares the code and occupies the more amount of memory space.

The derived class inherits all its base classes’ member functions but some of them are not required by the derived class. The derived class is
capable of calling all its member functions which are hidden. This situation is called as ‘**Fat Interface**’.
6.4 DESIGN FOR TESTING OF INHERITANCE RELATED BUGS IN OBJECT ORIENTED PROGRAMS

The testing consisting of four major analysis such as Function_model, Class_model, Variable_model and Interface_model. The Java file name is taken as input and then the file is located. In case of missing file, an error message will be displayed. In any Java source code, the source code may work perfectly even if there are multiple spaces or new lines between any executable instructions. A statement may contain unnecessary spaces and lines.

E.g.: int a=10; // a valid statement to assign 10 to an integer named ‘a’

        int a=10; // works fine though having numerous spaces

        int
        a

        =

        10

        ; //this will perform the same function as the above statements like int a=10; or the second statement with numerous spaces.

These kinds of statements may hinder the proper execution of scanning and parsing. To avoid such statements a method named “format” is used. The method “format” prepares the source code file for testing by indenting the file and removing multiple spaces and multiple new lines. It is used to separate the identifiers, literals, keywords and the separators, thereby reducing the time taken for scanning and parser.
Separators play an extremely important role in analyzing the source code of any program. The parentheses ‘(’ and ‘)’ are used to contain the list of parameters in method definition and invocation. It is also used for precedence in expressions, containing expressions in control statements, and surrounding cast type. In this tool, parentheses are used to search for the method definitions in a class or an interface. The semicolon and the opening brace ‘{’ separators are used to mark the end of a statement.

The scanning of the formatted source code is performed character by character. This result is the creation of the various tokens used in the program. The characters are joined together to form strings which may either be a variable, a function name, a keyword or a separator. Then linked list is created for every executable statement; each node contains the various tokens forming the executable statement as shown in Figure 6.4.

![Linked list](image)

Figure 6.4  Linked list

This linked list is analyzed based on the different probabilities of the placement of tokens to form meaningful statements; this in turn leads to the creation of the various objects of the models defined above. The information is extracted from the statements is used to fill the instance variable details of the objects created.

Eg: The creation of any class A can be done as any one of the following ways:

i)   class A{}

ii)  class A extends B{}
iii) private class A extends C implements I {} 
iv) class A extends B implements I {} 

The creation of any variable A may be done as any one of the following ways:

i) int a; 
ii) int a=1; 
iii) private int a=2; 
iv) protected int a; 
v) static int a=0; 
vi) static int a=(2*3/4%5); 

These are just a few of the numerous ways by which a class or a variable may be created. Similarly it is also possible to show numerous ways for creating functions and interface.

After this, different classes namely Class_model, Variable_model, Function_model, Interface_model are created as shown in Figure 6.5. These objects of classes are used to store the various details of the classes, variables, functions and interfaces. The object of Class_model has instance variables which store details such as the name of class, the way of inheritance (whether implements or extends) used. The interface or the package in which the class is defined is also recorded for future references. The objects of the Class_model are used to store detail of the various classes which are encountered during the execution of the program.

The objects of Variable_model class are used to store the details of the various attributes related to a variable, such as the name of the variable, the type of the variable, the access granted for the usage of the variable which
might be public, private or protected. The objects of this class are created when we encounter any variable declaration in the program encountered.

The Function_model forms an essential part of the program by keeping track of the details of the various functions which are created either in a class or an interface. The function name is recorded. This model stores the access granted for each function and it also stores if the statement scanned is just a function prototype or the complete function definition. The whole representation of a method is called signature. This model stores the signature of any function using a linked list. This linked list containing the signature is used in determining inheritance.

The Interface_model objects are used to store the details of the interface name and the various function prototypes. The objects of this model are developed to scan the interface definition.

![Diagram showing classes used in Tool Implementation]

**Figure 6.5 Classes used in Tool Implementation**
Next, four object arrays are created, the first array for storing the Variable_model, the second array for storing the Class_model, the third array for storing the Function_model and the fourth array for storing the Interface_model as detailed in figure 6.5. These objects in arrays serve as data banks used to test the different kinds of Bugs.

The following are the explanation of finding the bugs using arrays:

The ‘Class_model’ array is used to get the details of classes from the program and their properties such as access specifier, class name and inheritance information. By counting number of arrays, the number of classes used can be found out. The parent-child relationship can be represented in the form of matrix by using the above extracted details. The matrix can be created by specifying ‘1’ when the corresponding matrix elements are in parent–child relationship otherwise ‘0’.

Through the above information and the number of classes are in multilevel, the number of levels will be calculated by using the ‘extended by’ information and Class_model array values. If the number of level is greater than the threshold value of six, then the program will have the ‘spaghetti bug’.

In Class_model array, ‘acc_model.acc_arr[]’ property is used to describe about the attribute information of a class. The ‘Variable_model’ array has the class name and its attribute details. The information in both arrays can be matched to compare whether any children class in acc_model class updates the attribute of its parent classes. This is to find whether the system has ‘Naked Access Bug’.

The ‘Function_model’ gives the member function information such as class name, function name, access specifier like public, private and
protected and checks whether any interface is declared in the class or not. The class name is compared with Class_model array to find which function is held by the parent class. Then the functions called by a derived object are verified with the Function_model base class function. If the derived object has not used by any of the base class functions then the program has ‘Naughty Child’.

The ‘Variable_model’ array is used to find out the variables declared in the program. Comparison will be made on one class’s variable with other class’s variable name, access specifier and its data type. From comparison, if a subclass has the ‘Variable_model’ as its base class then the program suffers from the ‘Inadvertent Binding Bug’.

The steps followed in testing the Object Oriented Program for identifying inheritance related bugs are pictorially represented in figure. 6.6

**Step 1:** Input an Object Oriented Program to the ‘IRBT’ tool.

**Step 2:** Analyzing the given program with respect to classes, methods, attributes and inheritance property.

**Step 3:** The number of classes in multilevel inheritance will be calculated to compare with the threshold value. If Parent and Children relationship exceeds level 6, then the program has Spaghetti bug, else the program is in normal inheritance structure.

**Step 4:** The attributes information of parent and children classes are analyzed to find any copy of the parent attribute is created in children class also. If a child has a copy of the parent attribute then it results in the bug called ‘Inadvertent Binding’ bug in that program, else there is no ambiguity between parent and child attributes.
Step 5: It is checked if any attribute of parent is modified by the method of the child class. If it is so, then the bug called ‘Naked Access’ results.

Step 6: It is verified if the parent methods are used by the child class or not. If the methods of the parent class are not at all used by child class, then child class is said to be ‘Naughty Child’.

Step 7: Stop the process.

A set of Java programs is taken to test inheritance related bugs to find out if there is any bug in those Java programs.

The Object-Oriented Programs inheritance bugs like ‘Spaghetti’, ‘Naked Access’, ‘Naughty Child’ and ‘Inadvertent Binding’ has been found and listed in Table 6.1.
Figure 6.6 Steps for identifying Inheritance related Bugs
### 6.5 RESULTS AND DISCUSSIONS

Table 6.1 Inheritance Related Bugs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Classes</th>
<th>Parent</th>
<th>Child</th>
<th>Spaghetti Bug</th>
<th>Naked Access</th>
<th>Naughty Child</th>
<th>Inadvertent Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Occur or Not</td>
<td>Occur or Not</td>
<td>No. of naked access on child classes</td>
<td>Occur or Not</td>
</tr>
<tr>
<td>LibnMain</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>BookIssue</td>
<td>12</td>
<td>5</td>
<td>4</td>
<td>No</td>
<td>No</td>
<td>NIL</td>
<td>Yes</td>
</tr>
<tr>
<td>StudentPer</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>No</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>BinaryUtils</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>No</td>
<td>Yes</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>Diary</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>No</td>
<td>Yes</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>ClientJav</td>
<td>9</td>
<td>5</td>
<td>4</td>
<td>No</td>
<td>No</td>
<td>NIL</td>
<td>No</td>
</tr>
<tr>
<td>BusinessWriting</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>Bank</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>Higherstud</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>Performance</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>2</td>
</tr>
</tbody>
</table>
In the Table 6.1, different example source codes are taken to perform ‘Inheritance’ analysis. The table shows that the total number of classes, parent classes and children classes in the source code. The bugs such as ‘Spaghetti Bug’, ‘Naked Access’, ‘Naughty Child’ and ‘Inadvertent Binding’ which are derived from the concepts are used in source code. The ‘Yes’ in cell shows that the bug(s) is existing or not.

From the Table 6.1 ‘BusinessWriting.java’, the level of multilevel inheritance is greater than 6. So this code is raising the ‘Spaghetti Bug’. The following structure is a Java code of ‘BusinessWriting.java’ classes with 7 levels of multilevel inheritance is an example for ‘Spaghetti Bug’.

```java
class Person{
    .......
}
class Enstu extends Person{
    .......
}
class Bwrite extends Enstu{
    ............
}
class ASN extends Bwrite{
    ............
}
class CWrite extends ASN{
    ............
}
class Stud extends CWrite{
    ............
}
```
Class ASNIP extends Stud {

…………..
}
Class Icook extends ASNIP {
…………..
}

The below Java program ‘Bank.java’ is an example for ‘Naked Access’.

Class Account extends BankCustomer{
private double balance;
…………………………
}
Class IntScheme extends Account {
…………
public void getIntdetails(double i)

{
    intrst=i;
    balance=balance*intrst;
    amt +=balance;
}
………
}

From the above fragment, the ‘balance’ property is processed in the IntScheme class (derived class) of its base class (Account). So this fragment rises the ‘Naked Access’ Bug.

class Student extends Alumni
{
    public static void main(String args[])
}
The above fragment shows that the class ‘Student’ uses only its own functions for assigning the values. But it doesn’t use any other functions which are in ‘Alumni’ child does not use the methods in parent class properly. So it has the ‘Naughty child bug’.

‘Higherstud.java’ source file consists of two classes such as ‘Person’ and ‘Department’.

class Person
{
    private String name;
    int age;

    ...........
    ...........
}

Class Employee extends Person
{
    ...........
    ...........
}
Class Department extends Employee
{
    Private String name;
    .......
}

These two classes have a data member called ‘name’. This might cause confusion to the developer team about the usage of that data member in the derived class ‘Department’. So this situation invokes the ‘Inadvertent Binding’ bug.

But, the other bugs like ‘Square Peg in a Round Hole’, ‘Fat Interface’, ‘Weird Hierarchies’ and ‘Worm Holes’ may be developed when the proper design is not made. The implemented tool is useful to finding Implementation phase inheritance bugs during object-oriented development.

6.6 CONCLUSION

The compiler checks the syntax errors and converts the source code into byte code. But the compiler is not a complete solution for the bug detection. This system is developed to test the Java based products as a static testing tool. The static test is applied to detect the inheritance related bugs in the Java programs. Programs are completely tokenized using linked list and stack used to store all information for finding implementation inheritance related bugs such as ‘Inadvertent binding’, ‘Naughty Child’, ‘Spaghetti’ and ‘Naked Access’. This will be useful in software changes and maintenance.