International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 4– No.9, December 2012 – www.ijais.org

Importance of Inheritance and Interface in OOP Paradigm Measure through Coupling Metrics

Gopal Goyal
M.Tech Scholar
PCST, Indore, India
India

Sachin Patel
HOD, IT
PCST, Indore, India
India

ABSTRACT

A large numbers of metrics have been proposed for measuring properties of object-oriented software such as size, inheritance, cohesion and coupling. The coupling metrics presented in this paper exploring the difference between inheritance and interface programming. This paper presents a measurement to measure coupling between object (CBO), number of associations between classes (NASSocC), number of dependencies in metric (NDepIN) , number of dependencies out metric (NDepOut) , Number of children (NOC) and *Depth of* Inheritance Tree (DIT) in object oriented programming. A measurement is done for C# inheritance and interface programs. The metric values of class inheritance and interface prove which program is good to use and beneficial for C# developers.

Keywords –OOP, OOA, CBO, Inheritance, Interface, Coupling

1. INTRODUCTION

Object-oriented design and programming is the dominant development paradigm for software systems today. Recently so many languages are object-oriented (OO) programming languages [2]. In object oriented programming we provide abstraction by classes and interfaces. Classes are used to hold functional logic and an interface is used to organize source code. According to object oriented programming, the class provides encapsulation and abstraction and the interface provides abstraction and cannot inherit from one class but can implement multiple interfaces. The above said differences are minor and they are very similar in structure, complexity, readability and maintainability of source code [1]. Here, the difference in usage of class inheritance and interface concepts are measured for C# programs by coupling metrics. Density of source code directly relates to cost and quality. For measuring complexities, we have cohesion and coupling models. The coupling models presented in the literature show many possible interactions that can occur between objects in the software systems and offer metrics to measure complexity. Software engineering best practices promote low coupling between components in order to decrease interdependencies and facilitate evolution. This paper presents a comparison between object oriented interfaces and inheritance class.

2. PREVIOUS WORK ON OO SOFTWARE METRICS

Object-oriented measurement has become an increasingly popular research area. Metrics are powerful support tools in software development and maintenance. They are used to assess software quality, to estimate complexity, cost and effort, to control and improve processes. The metrics that are important to calculate reusability are related to inheritance and coupling.

A. *Traditional Metrics*

McCabe Cyclomatic Complexity (CC): Cyclomatic complexity is a measure of a module control flow complexity based on graph theory [3]. Cyclomatic complexity of a module uses control structures to create a control flow matrix, which in turn is used to generate a connected graph. The graph represents the control paths through the module. The complexity of the graph is the complexity of the module [4], [3]. Fundamentally, the CC of a module is roughly equivalent to the number of decision points and is a measure of the minimum number of test cases that would be required to cover all execution paths. A high Cyclomatic complexity indicates that the code may be of low quality and difficult to test and maintain.

Source Lines of Code (SLOC): The SLOC metric measures the number of physical lines of active code, that is, no blank or commented lines code [5]. Counting the SLOC is one of the earliest and easiest approaches to measuring complexity. It is also the most criticized approach [6]. In general the higher the SLOC in a module the less understandable and maintainable the module is.

Comment Percentage (CP): The CP metric is defined as the number of commented lines of code divided by the number of non-blank lines of code. Usually 20% indicates adequate commenting for C++ [7]. A high CP value facilitates in maintaining a system.

3. OBJECT –ORIENTED PROGRAMMING

The terms "objects" and "oriented" in something like the modern sense of object-oriented programming seem to make their first appearance at MIT in the late 1950s and early 1960s[8]. The object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the program. Instead, the data is accessed by calling specially written functions, commonly called methods, which are either bundled in with the data or inherited from "class objects." An object-oriented program will usually contain different types of objects, each type corresponding to a particular kind of complex data to be managed or perhaps to a real-world object or concept such as a bank account, a hockey player, or a bulldozer. Numerous software metrics related to software quality assurance have been proposed in the past and are still being proposed. Several books presenting such metrics exist, such as Fenton‘s [12], Sheppard‘s [13] and

others. Although most of these metrics are applicable to all programming languages, some metrics apply to a specific set of programming languages. Among metrics of this kind, are those that have been proposed for object–oriented programming languages.

4. CLASS INHERITANCES AND INTERFACES

Inheritance is one of the fundamental concepts of Object Orientated programming, in which a class "gains" all of the attributes and operations of the class it inherits from, and can override/modify some of them, as well as add more attributes and operations of its own. In Object Oriented Programming, inheritance is a way to compartmentalize and reuse code by creating collections of attributes and behaviors called objects that can be based on previously created objects. In classical inheritance where objects are defined by classes, classes can inherit other classes. The new classes, known as subclasses (or derived classes), inherit attributes and behavior (i.e. previously coded algorithms) of the pre-existing classes, which are referred to as super classes, ancestor classes or base classes. The inheritance, relationships of classes gives rise to a hierarchy. The inheritance concept was invented in 1967 for Simula [8]. Interfaces allow only method definitions and constant attributes. Methods defined in the interfaces cannot have implementations in the interface. Classes can implement. The interface by providing bodies for the methods defined in the interface. An interface is a contract between a client class and a server class [8]. It helps to decouple the client from the server. Any intended change on the methods defined in the interface will impact both the client and server classes. Possible changes are as follows: 1) changing the name of a method, 2) changing the signature of a method, and 3) changing the return type of a method. There are two other possible changes that worth noting. If a new method is added to an interface, this will also impact the server and client classes that currently use or implement the interface. On the other hand if the implementation detail of a method inside a server class is changed, this change only affects the client class and not the interface. This specific case is more a code issue than a design issue and therefore it is not a concern in this evaluation. Interfaces have another very important role in the C# programming language. Interfaces are not part of the class hierarchy, although they work in combination with classes. The C# programming language does not permit multiple inheritance (inheritance is discussed later in this lesson), but interfaces provide an alternative. In C#, a class can inherit from only one class but it can implement more than one interface. Therefore, objects can have multiple types: the type of their own class and the types of all the interfaces that they implement. This means that if a variable is declared to be the type of an interface, its value can reference any object that is instantiated from any class that implements the interface.

5. COUPLING METRICS

Several authors have introduced different approaches and proposed measures to coupling in object-oriented systems [10]. In this paper two CK metric and three Genero M metrics are used for measure coupling performance.

B. *Number of children (NOC)*

Number of children metric was introduced by CK [9]. NOC defines number of immediate sub-classes subordinated to a class in the class hierarchy. This metric measures how many sub-classes are going to inherit the methods of the parent class. NOC relates to the notion of scope of properties. If NOC grows it means reuse increases. On the other hand, as NOC increases, the amount of testing will also increase because more children in a class indicate more responsibility. So, NOC represents the effort required to test the class and reuse.

C. *Coupling Between Objects-CBO*

According to CK [9] "CBO for a class is a count of the number of other classes to which it is coupled". A class is coupled with another if the methods of one class use the methods or attributes of the other class. An increase of CBO indicates the reusability of a class will decrease. Multiple accesses to the same class are counted as one access. Only method calls and variable references are counted. Thus, the CBO values for each class should be kept as low as possible [9].

D. *Number of Dependencies In(NDepIN)*

The Number of Dependencies In metric (NDepIN) is defined as the number of classes that depend on a given class [10]. This metric is proposed to measure the class complexity due to dependency relationships. The greater the number of classes that depend on a given class, the greater the inter-class dependency and therefore the greater the design complexity of such a class.

E. *Number of Dependencies Out(NDepOut)*

The Number of Dependencies Out metric (NDepOut) [10] is defined as the number of classes on which a given class depends. It is better to minimize NDepOut value, since; higher values represent a situation in which many dependencies are spreading across the class diagram

F. *Number of Association(NASSocC)*

The Number of Association per Class metric is defined as the total number of associations a class has with other classes or with itself. This metric is used to measure complexity and coupling [9, 10]. When the number of associations are less the coupling between objects are reduced. This metric was introduced by Brian.

G. *Depth of Inheritance Tree (DIT)*

Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritances, the DIT will be the maximum length from the node to the root of the tree. DIT is a measure of how many ancestor classes can potentially affect this class [9].

6. PROPOSED APPROACH

Goal: Exploring the difference between class inheritance and interface in C# programming through coupling metrics.

Hypothesis: SIX object oriented metrics are used for coupling measures in object oriented class inheritance and interface programs.

1. Two C# programs are used with inheritance concept in this paper.
2. These programs are introduced with maximum possible interface.
3. All six metrics are applied to both inheritance and interface programs.
4. The results are compared between inheritance and interface coupling measures.

7. RESULTS

The metrics discussed above are applied for both inheritance and interface programs. The results are show in Table 1 and Table 2.

H. Measures the difference between Inheritance and Interface

To validate the above metrics two object oriented inheritance programs have been taken and possible interfaces have been introduced in class inheritance programs. The first study considered is a vehicle classification class inheritance program which is represented in Figure 1.

```
class Vehicle
  {
    public string name;
    public int wheelscount;
    public void getData()
     {
     }
   public void displayData()
     {
     }
  }
class LightMotor :Vehicle
  {
   public int speedlimit;
   public int capacity;
  }
class HeavyMotor : Vehicle
  {
   public int speedlimit;
   public int capacity;
  public string permit;
  }
class GearMotor : LightMotor
 {
   public int gearcount;
  }
class NongearMtor : LightMotor
  {
   }
class Passenger : HeavyMotor
  {
    public int sitting;
  }
class Goods : HeavyMotor
  {
  }
```

Figure 1: Vehicle classification using Class Inheritance – adopted from [11]

The above said class inheritance Figure 1 is introduced with possible number of interface and represented in Figure 2.

```
interface Lightmotor
{
   public  void Getspeedcap();
}
 interface Vehicle
 {
    public void GetData();
    public void DisplayData();
    public void Getnamewc();
 }
 interface Heavymotor
 {
    public void Getspeeder();
 }
class Gearmotor : Vehicle, Lightmotor
 {
    public int gearcount;
    public void GetData();
    public void DisplayData();
    public void Getnamewc();
    public void Getspeedcap();
 }
class Nongearmotor : Vehicle
 {
    public void GetData();
```

```
    public void DisplayData();
    public void Getnamewc();
 }
class Passenger : Vehicle, Heavymotor
{
   public void GetData();
   public void DisplayData();
   public void Getnamewc();
   public void Getspeeder();
}
class Goods : Vehicle, Heavymotor
 {
    public void GetData();
    public void DisplayData();
    public void Getnamewc();
    public void Getspeeder();
 }
```

Figure 2: Vehicle classification using Interface

For the above said two programs the coupling metrics are measured and tabulated in Table 1. By comparing the table values for both the programs the interface values are reduced for almost all metrics.

Table 1: Comparing Measures for Figure 1 & 2.

	Metric Classes	CBO	NASSoc C	NDepI N	NDep Out	NOC	DIT
Fig 1	Vehicles	2	2	2	0	2	2
	Lightmo	3	3	2	1	2	2
	Heavym	3	3	2	1	2	2
	Gearmot	1	1	0	1	0	0
	Nongear	1	1	0	1	0	0
	Passenge	1	1	0	1	0	0
	Goods	1	1	0	1	0	0
Fig 2	Vehicles	0	0	0	0	0	0
	Lightmo	0	0	0	0	0	0
	Heavym	0	0	0	0	0	0
	Gearmot	0	0	0	0	0	0
	Nongear	0	0	0	0	0	0
	Passenge	0	0	0	0	0	0
	Goods	0	0	0	0	0	0

The Second program chosen is shapes hierarchy and is given in Figure 3.

```
class Shape
{
   public void Draw();
   public void Element();
}
class RegularPolygon:Shape
{
   public int Linesegment;
   public void Perimeter();
}
class Ellipse: Shape
{
  public  int  curved;
  public int  Surface;
}
class Triangle : RegularPolygon
{
      public int sumofangles = 180;
      public void setsides();
      public void Area();
}
class Rectangle : RegularPolygon
{
      public  int sumofangles = 360;
      public void setsides();
      public void Area();
}
class Circle : Ellipse
{
   public int symmetrical;
   public void Circumference();
```

```
}
class Salene : Triangle
{
   public int Nosidesequal;
}
class Isosceles : Triangle
{
   public int sideequal2;
   public int Anglesequal2;
}
class Equilateral : Triangle
{
   public int sidesequal3;
   public int Anglesequal3;
}
class Square : Rectangle
{
   public int oppositesidequal;
   public int angles4;
}
```

Figure 3: Class Inheritance program for Shapes

The above class inheritance program is converted into interface concept program and is represented as Figure 4.

```
interface Shape
{
   public void Draw_Element();
}
interface RegularPolygon
{
   public void Linesegment();
   public void Perimeter();
}
interface Ellipse
{
   public void Circumference();
}
class Triangle : Shape
{
   int Sumofangles = 180;
   public void Draw_Element();
   public void setsides();
   public void Area();
}
 class Rectangle : RegularPolygon
{
   int sumofangles = 360;
   public void Perimeter();
   public void Linessegment();
   public void setsides();
   public void Area();
}
class Circle : Ellipse
{
   int symmetricalpictur;
   public void Circumference();
}
class Scalene : Triangle
{
   int notequalsides;
}
class Isosceles : Triangle
{
   int sidesequal;
   int angleequal;
}
class Equilateral : Triangle
{
   int sidesequal;
   int angleequal;
}
class square : Rectangle
{
   int opposite;
   int sidesequal;
   int anglesequal;
}
```

Figure 4: Program for shapes using Interface

International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 4– No.9, December 2012 – www.ijais.org

For Figure 3 and Figure 4 the above said metric values are measured and tabulated in Table 2.

Table 2: Coupling Measures for Figure 3 & 4.

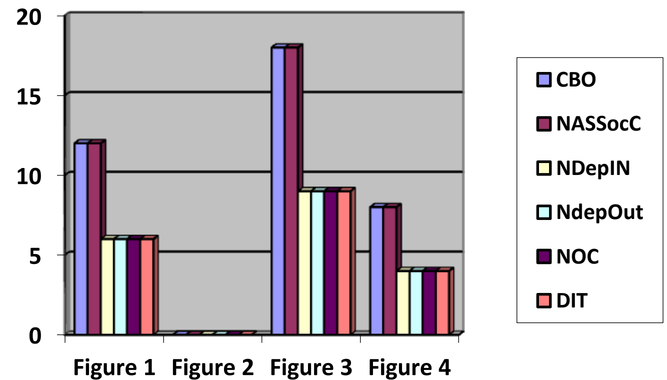
	Metric / Classes	CBO	NAS SocC	NDep IN	NDep Out	NOC	DIT
Figure 3	Shape	2	2	2	0	2	2
	Regular polygon	3	3	2	1	2	2
	Ellipse	2	2	1	1	1	1
	Triangle	4	4	3	1	3	3
	Rectangle	2	2	1	1	1	1
	Circle	1	1	0	1	0	0
	Scalene	1	1	0	1	0	0
	Isosceles	1	1	0	1	0	0
	Equilateral	1	1	0	1	0	0
	Square	1	1	0	1	0	0
Fig 4	Shape	0	0	0	0	0	0
	Regular polygon	0	0	0	0	0	0
	Ellipse	0	0	0	0	0	0
	Triangle	3	3	3	0	3	3
	Rectangle	1	1	1	0	1	1
	Circle	0	0	0	0	0	0
	Scalene	1	1	0	1	0	0
	Isosceles	1	1	0	1	0	0
	Equilateral	1	1	0	1	0	0
	Square	1	1	0	1	0	0

By Comparing the table values from Table 1 and Table 2 for the above programs the total coupling measures for each metric for all programs are tabulated in Table 3.

Table 3: Coupling Measures for total Inheritance and Interface programs

Metric / Figure	CBO	NAS SocC	NDepIN	NDep Out	NOC	DIT
Figure 1	12	12	6	6	6	6
Figure 2	0	0	0	0	0	0
Figure 3	18	18	9	9	9	9
Figure 4	8	8	4	4	4	4

Graph 1: Coupling Measures Comparison



8. CONCLUSION

This paper presents an idea on how to reduce coupling in object oriented programming. It is helpful for the developers to check which concept is best between inheritance and interface. When CBO is reduced reusability will be increased. We have proposed an approach to measure the reusability of object oriented program based upon CK metrics. Since reusability is an attribute of software quality, we can quantify software quality by measuring software reusability. Hence, this approach is important to measure difference between class inheritance and interface.

9. REFERENCE

[1] I. Jacobson, G. Booch, and J. E. Rumbaugh, The unified software development process. Addison-Wesley, 1999.

[2] V. Krishnapriya, Dr. K. Ramar, " Exploring the Difference between Object Oriented Class Inheritance and Interfaces

Using Coupling Measures", 2010 International Conference on Advances in Computer Engineering, 978-0-7695-4058-0/10 $26.00 © 2010 IEEE.

[3] McCabe and Associates, Using McCabe QA 7.0, 1999, 9861Broken Land Parkway 4th Floor Columbia, MD 21046.

[4] McCabe, T. J., "A Complexity Measure", IEEE Transactions on Software Engineering, SE-2(4), pages 308-320, December 1976.

[5] Lorenz, Mark & Kidd Jeff, Object-Oriented Software Metrics, Prentic, Hall, 1994.

[6] Tegarden, D., Sheetz, S., Monarchi, D., "Effectiveness of Traditional Software Metrics for Object-Oriented Systems", Proceedings: 25th Hawaii International Conference on System Sciences, January, 1992, pp. 359-368.

[7] Rosenberg, L., and Hyatt, L., "Software Quality Metrics for Object Oriented System Environments", Software assurance Technology Center, Technical Report SATC-TR-95-1001,NASA Goddard Space Flight Center, Greenbelt, Maryland 20771.

[8] http://en.wikipedia.org/wiki/Object-oriented_programming.

[9] Chidamber S. and Kemerer C.: "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, 1994.

[10] Marcela Genero, Mario Piattini and Coral Calero," A Survey of Metrics for UML Class Diagrams", in Journal of Object Technology, Vol. 4, No. 9, Nov-Dec 2005.

[11] Pradeep Kumar Bhatia, Rajbeer Mann, " An Approach to Measure Software Reusability of OO Design", Proceedings of 2nd International Conference on Challenges & Opportunities in InformationTechnology(COIT-2008),RIMT-IET,Mandi gobindgarh, March 29, 2008.

[12] N. Fenton & S.L. Pfleeger, ―Software Metrics: A Rigorous & Practical Approach‖, Second edition, 1997, International Thomson Computer Press. M.J. Sheppard & D. Ince, ―Derivation and Validation of Software Metrics, Clarendon Press, Oxford, UK, 1993.