A Brief Study on Inheritance

Sujeet Kumar & Ashish Kumar Gupta

Department of Information and technology Dronacharya College of Engineering,Gurgaon-122001, India

Email:sujeet.16939@ggnindia.dronacharya.info ; Email:ashish.16907@ggnindia.dronacharya.info

Abstract-

In object-oriented programming (OOP), C++ strongly support the concept of Reusablity. In this paper we have studied the inheritance and its types of inheritance. We have studied different applications, some of them are Overriding and Code Reuse. We have also studied issue related to Complex inheritance. In this paper we have also studied various alternatives. In this paper we have sudied the inheritace concept, types, alternatives and applications. We have also studied briefly about subclasses and superclasses of inheritance .

Keywords:

Overriding; Code Reuse; Subclasses; Super classes

1. INTRODUCTION

Inheritance is when an object or class is based on another object or class, using the same implementation or specifying implementation to maintain the same behavior (realizing an interface; inheriting behavior). It is a mechanism for code reuse and to allow independent extensions of the original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a hierarchy. Inheritance was invented in 1967 for Simula.

1.1 Types of inheritance

There are various types of inheritance, depending on paradigm and specific language. A fundamental difference is whether one can inherit from only a single other object or class, which is known as *single inheritance,* or whether one can inherit from multiple other objects or classes, which is known as *multiple inheritance*.

- **Classical inheritance** is used in class-based programming, where objects are defined by classes, and classes can inherit attributes and implementation (i.e., previously coded algorithms associated with a class) from pre-existing classes called *base classes*, *superclasses*, or *parent classes*.
- **Differential inheritance** is used in prototype-based programming, where objects inherit directly from other objects.

1.1.1 Subclasses and superclasses

A *Subclass*, "derived class", *heir class*, or *child class* is a modular, derivative class that inherits one or more language entities from one or more other classes (called *superclasses*, *base classes*, or *parent classes*). The semantics of class inheritance vary from language to language, but commonly the subclass automatically inherits the instance variables and member functions of its superclasses.

1.2 Applications

Inheritance is used to co-relate two or more classes to each other.

1.2.1 Overriding

Many OOP languags permit a class or object to replace the implementation of an aspect—typically a behavior—that it has inherited. This process is usually called *overriding*. Overriding introduces a complication: which version of the behavior does an instance of the inherited class use—the one that is part of its own class, or the one from the parent (base) class? The answer varies between programming languages, and some languages provide the ability to indicate that a particular behavior is not to be overridden and should behave as defined by the base class. For instance, in C#, the base method or property can only be overridden in a subclass if it is marked with the virtual, abstract, or override modifier. An alternative to overriding is hiding the inherited code.

1.2.2 Code reuse

Implementation inheritance is the mechanism whereby a subclass re-uses code in a base class. By default the subclass retains all of the operations of the base class, but the subclass may override some or all operations, replacing the base-class implementation with its own.

In the following Python example, the sub class CubeSumComputer overridesthe tran sform() method of the base class SquareSumComputer. The base class comprises operations to compute the sum of the squaresbetween two integers. The subclass re-uses all of the functionality of the base class with the exception of the operation that transforms a number into its square, replacing it with an operation that transforms a number into its cube. The subclass therefore computes the sum of the cubes between two integers.

```
class SquareSumComputer:
    def __init__(self, a,
b):
        self.a = a
        self.b = b

    def transform(self, x):
        return x * x

    def inputs(self):
        return range(self.a,
self.b)

    def compute(self):
        return
sum(self.transform(value)
for value in self.inputs())

class
CubeSumComputer(SquareSumCom
puter):
    def transform(self, x):
        return x * x * x
```

In most quarters, class inheritance for the sole purpose of code reuse has fallen out of favor. The primary concern is that implementation inheritance does not provide any assurance of polymorphicsubstitutability—an instance of the reusing class cannot necessarily be substituted for an instance of the inherited class. An alternative technique, delegation, requires more programming effort, but avoids the substitutability issue. In C++ private inheritance can be used as form of *implementation inheritance* without substitutability. Whereas public inheritance represents an "is-a" relationship and delegation represents a "has-a" relationship, private (and protected) inheritance can be thought of as an "is implemented in terms of" relationship.

1.3 Inheritance vs subtyping

Inheritance is similar to but distinct from subtyping.[1] Subtyping enables a given type to be substituted for another type or abstraction, and is said to establish an **is-a** relationship between the subtype and some existing abstraction, either implicitly or explicitly, depending on language support. The relationship can be expressed explicitly via inheritance in languages that support inheritance as a subtyping mechanism. For example, the following C++ code establishes an explicit inheritance relationship between classes **B** and **A**, where **B** is both a subclass and a subtype of **A**, and can be used as an **A** wherever a **B** is specified (via a reference, a pointer or the object itself).

```
class A
{ public:
   void DoSomethingALike()
const {}
};

class B : public A
{ public:
   void DoSomethingBLike()
const {}
};

void UseAnA(A const& some_A)
{

some_A.DoSomethingALike();
}
```

```
void SomeFunc()
{
   B b;
   UseAnA(b); // b can be
substituted for an A.
}
```

In programming languages that do not support inheritance as a subtyping mechanism, the relationship between a base class and a derived class is only a relationship between implementations (a mechanism for code reuse), as compared to a relationship between types. Inheritance, even in programming languages that support inheritance as a subtyping mechanism, does not necessarily entail behavioral subtyping. It is entirely possible to derive a class whose object will behave incorrectly

1.4 Issues

Complex inheritance, or inheritance used within an insufficiently mature design, may lead to the yo-yo problem

1.5 Alternatives

While inheritance is widely used, there are various alternatives. Some people advocate object composition instead of inheritance; see composition over inheritance. Similar mechanisms to inheritance (reusing implementation) can be achieved by mixins and traits.

1.6 Conclusion

The research paper concludes on studing on inheritance, its types. In object-oriented programming (OOP), C++ strongly support the concept of Reusablity. In this paper we have studied the inheritance and its types of inheritance. We have studied different applications, some of them are

Overriding and Code Reuse. We have also studied issue related to Complex inheritance. In this paper we have also studied various alternatives. In this paper we have sudied the inheritace concept, types, alternatives and applications. We have also studied briefly about subclasses and superclasses of inheritance .

1.7 References

- Cook, William R.; Hill, Walter; Canning, Peter S. (1990). "Inheritance is not subtyping". *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. pp. 125–135.doi:10.1145/96709.96721. ISBN 0-89791-343-4.
- **Jump up^** Mitchell, John (2002). "10 "Concepts in object-oriented languages"". *Concepts in programming language*. Cambridge, UK: Cambridge University Press. p. 287. ISBN 0-521-78098-5.
- Hughes, J. R. (1986). Genetics of smoking: a brief review. *Behavior Therapy*, *17*(4), 335-345.
- Orr, H. A. (2005). The genetic theory of adaptation: a brief history. *Nature Reviews Genetics*, *6*(2), 119-127.
- True, W. R., Rice, J., Eisen, S. A., Heath, A. C., Goldberg, J., Lyons, M. J., & Nowak, J. (1993). A twin study of genetic and environmental contributions to liability for posttraumatic stress symptoms. *Archives of general psychiatry*, *50*(4), 257-264.
- Heath, A. C., & Martin, N. G. (1993). Genetic models for the natural history of smoking: evidence for a genetic influence on smoking persistence. *Addictive behaviors*, *18*(1), 19-34.
- Heath, A. C., & Martin, N. G. (1993). Genetic models for the natural history of smoking: evidence for a genetic influence on smoking persistence. *Addictive behaviors*, *18*(1), 19-34.
- Barnes, D. J., Kölling, M., & Gosling, J. (2006). *Objects first with Java: A practical introduction using Bluej*. London: Pearson Prentice Hall.