Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 375–381.

Virtuoso Virtuality

Marianna Sipos

Department of Information Technology, Zrínyi Miklós University of Defence
e-mail: marianna.sipos@zmne.hu

Abstract

The development of object-oriented programming has not lost its momentum. Software engineering insists new solutions. C++, Java, C# the most popular programming languages and the .NET Framework class library give us in every version new viewpoints and new directions. The introduction of new versions concentrates on the new features. In this situation we often forget to analyse how to integrate a well known paradigm and a new technique. A lecture cannot cover all of the changes, so it concentrate on the development of virtuality and encapsulation in object-oriented programming.

Keywords: virtual method, early binding, late binding, polymorphism, override, sealed, encapsulation, property, event handling, delegate, event, signature, subscribe, dll hell, efficiency.

1. Polymorphism is a fundamental benefit of object-orientation

In 1980 Bjarne Stroustrup began developing 'C with Classes' cited as follows "invented C++, wrote its early definitions, and produced its first implementation... chose and formulated the design criteria for C++, designed all its major facilities, and was responsible for the processing of extension proposals in the C++ standards committee." [1] The language was first titled C++ and published in October 1985. From our viewpoint the main difference between the two languages is, that virtual functions were not implemented in C with Classes, but they were implemented in C++. [2] For more information about the development of C++ see [3]. If we see the first three commonly used object-oriented programming languages, we can see that all of them use virtuality.

Virtuality was first defined in C++ and all of the instance methods are virtual in Java. This means: all theories, which were later defined, can work with the virtual methods. This extension of virtuality gives us new challenges.

1.1. Difference between virtual and non virtual functions

1.1.1. Non virtual methods

The function is determined by the type of variable (reference). You can call only those instance methods which are defined in the class of the instance type. In the literature this solution is called early binding, because the function address is fixed in compile-time.

1.1.2. Virtual methods

The function is determined by the type of object. The type of object is one of the inherited classes or the variable class itself. When we override the method, we can achieve it through the objects of the inherited class. In the literature this solution is called late binding. The function address is fixed in runtime, because we know only at runtime which object will be referenced by the variable. The name late binding denotes that runtime is later than compile time.

1.2. How can we solve polymorphism?

Virtual functions give us flexibility, but flexibility is expensive. The more flexible you are, the less efficient you are. When every object calls it's own function, then the object has to know the address of the function. To solve the problem we use mainly the following techniques:

- Virtual Method Table.
- Every object stores the address of every virtual method.

1.2.1. Virtual Method Table (VMT)

The object stores the address of VMT. The VMT stores the address of virtual functions. Every class, which has virtual functions has VMT as well.

The steps of virtual method call are:

Step 1: go to the object address and read the VMT address!

Step 2: go to VMT and read the function address!

Step 3: go to the function!

This solution needs memory capacity to store VMT, and time for these 3 steps at every function call. (Figure 1.)

1.2.2. Objects store the virtual function addresses

All of the objects store the addresses of all of the virtual functions.

The steps of virtual method call are:

Step 1: go to the object address and read the function address!

Step 2: go to the function!

This solution needs only 2 steps at every function call, but each object stores all the virtual function addresses.

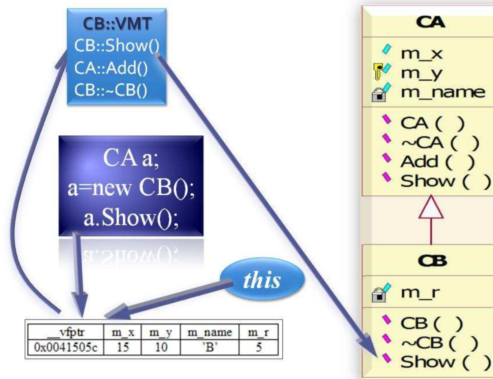

Figure 1: Usage of Virtual Function Table

2. How does polymorphism work in different languages?

2.1. C++

C++ has virtual and non virtual instance methods. If a function is virtual, it will be virtual in all inherited classes, even if we do not use the virtual keyword in the derived class.

2.2. Java

In Java all instance methods are virtual. If you see the efficiency solutions of virtuality, you can understand: this is one of the reasons, why Java is so slow. If we see the problem from the viewpoint of education, in Java books you cannot read about early binding. Students, who learn Java miss gaining the knowledge of the effectiveness of calling functions.

2.3. C#

C# has virtual and non virtual instance methods.

- First we use the keyword 'virtual' to make the method virtual.

- We use the keyword 'override' to override the base class virtual method.
- We can close the virtuality of the function with the keywords 'sealed override'.
- If you used the 'sealed override' keyword in the base class, you cannot override the function. You can either write a non virtual function with the same signature, or you can start a new virtual chain with the virtual keyword.
- In C# you cannot use the keyword 'virtual' to override the function as you can in C++. If you do, it will only start a new virtuality.

2.4. One reason for DLL hell

The usage of dynamic link library means that you put the parts of your application into DLLs. One of the benefit of using DLLs is, that you can change a DLL, without changing the other parts of the application. The new version of DLL substitutes the old version. Another benefit is that other applications can work with the same DLL.

The DLL hell occurs when a software changes a DLL, but another software does not work with the new DLL. Let us look at the following situation (Figure 2): we use version 1.0 of Component1. We have written a new virtual method named Fv in class B, which was inherited from class A. Class A is in Component1. The company issues a new version of Component1. Class A has in the new version a new virtual function with the name Fv. Of course the company doesn't know anything about our B class with the virtual function Fv. In Java and in C++ the function Fv in class B will override the function Fv in class A. The programmer of class A didn't want this override, nor did the programmer of class B.

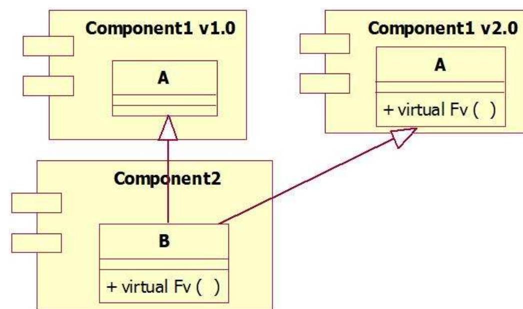

Figure 2: Override without intention

The problem has been solved in C#. In C# you can override a function only with the keyword 'override', and you can use the keyword 'override' only when the base class has a virtual function with the same signature.

3. Sealed methods, sealed types

You can use the sealed modifier on a method that overrides a virtual method in a base class. This enables you to allow classes to derive from your class and prevent them from overriding sealed methods. You can use the keyword 'sealed' only in front of virtual methods and it must always be used with 'override'. [4]

You can use the sealed modifier in front of classes as well. It prevents other classes from inheriting from the sealed class. We don't have any reason to use in sealed classes protected members or virtual functions.

We seal a class:

- when it has inherited protected members with security-sensitive information.
- when it has inherited many virtual members, because it is easier to seal the whole class as opposed to sealing each virtual member.

Sealed classes are more efficient, because they have no virtual functions.

- Their objects do not store VMT or virtual function addresses.
- All functions have early binding, so they have rapid execution.

3.1. Object class

Object is the base class of all types. The Object class has some virtual functions:

- public virtual bool Equals(Object obj)
- protected virtual void Finalize()
- public virtual int GetHashCode()
- public virtual string ToString() [4]

To make a type efficient we need to seal these functions, or seal the type itself, even if it loses extensibility.

Structs are efficient types because their instances are value types (not references) and are implicitly sealed.

3.2. Types and virtuality in C#

3.2.1. Interface

has virtual functions which need implementation.

3.2.2. Abstract class

can never be instantiated. An abstract class can have abstract methods which need implementation.

3.2.3. Class

By default a class inherits some virtual functions from its base class (from Object class), and we can declare some virtual functions in it. We can also seal the inherited virtual functions.

3.2.4. Sealed class

Has no virtual functions.

3.2.5. Static class

Static functions aren't virtual. Static class has only static functions, so it has no virtual functions.

3.2.6. Struct

Implicitly sealed, has no virtual functions.

3.2.7. Delegate

A function reference type. A delegate can refer to a static (never virtual) method, a non virtual (instance) method or a virtual (instance) method of a class. Only the return type and the parameters have to match.

4. Event handling

An event is an instance of a multicast delegate. [5] If a delegate can refer to all types of methods with the same parameters and type, then we can subscribe virtual, non virtual and also static methods to an event; they only need to have the signature of the events delegate. So we can use the benefits of event handling and polymorphism together.

You only have to pay attention to the difference between the two mechanism. Event handling works with multicast delegates, so more functions can subscribe to the same event. When the event occurs, each subscribed function will be called one after the other. When a function is overridden, it depends on the object, which function is called. Howewer you have to pay attention, not to subscribe the derived class function to the event, because if you do, the derived class function will be called twice, once instead of the base class virtual function, and again because it has been subscribed itself.

5. Development of encapsulation

At first encapsulation had the meaning: that there are in the same class coherent data and functions which operate on this data. The data members have private

visibility, the functions mainly public visibility.

Later the getter- setter technology was used. It means, that we can use get methods to get information from the data members and we can modify the value of the data members with set methods.

After that, we used properties which include the get-set methods. [6]

Today we can use properties to save data, and we don't need to write a data member, we only need to know, that behind a property there is a data member. Of course when we introduce more security or other functionality into the property's get set methods, we need to know the name of the data member behind it.

Get and set methods can be virtual as well, so we can use virtuality together with properties.

6. Summary

Object-orientation in C based languages has had an incredible development. In this paper you can see how it has gained more and more accent virtuality at least in Java, where all instance methods are virtual. Later in order to maintain efficiency it was necessary to stop the usage of virtuality using sealed override, later with sealed classes and then with implicit sealed types.

The same dynamic development can be seen in encapsulation. At first we only used visuality to hide data, then in Java and in C# the getter setter technology was introduced. Today in C# we can use properties to save our data without knowing the data members.

References

[1] Sroustrup, B.: The C++ Programming Language, Addison-Wesley Pub Co, 3rd edition, 2000. p.10.

[2] Sroustrup, B.: The Design and Evolution of C++, Addison-Wesley Pub Co, 1994, 461 pp.

[3] Porkolab, Z.: The new C++ standard - without concepts?, Proceedings of 8th International Conference on Applied Informatics, Eger, Hungary, January 27 - 30, 2010.

[4] Microsoft Visual Studio 2008, Documentation, Microsoft Press, Redmond, 2008

[5] Albert I., Balássy Gy., Charaf H., Erdélyi T., Horváth Á., Levendovszky T., Péteri Sz., Rajacsics T.: A .NET Framework és programozása, Szak kiadó, 2004.

[6] Sipos M: Programozás élesben, C#, InfoKit, Budapest, 2004.

Marianna Sipos Department of Information Technology
Zrínyi Miklós National Defence University of Hungary
9-11. Hungária krt, Budapest, H-1101
Hungary