2Y-1

Views in Object-Oriented Programming

T. Ohira and T. Kamimura
IBM Research, Tokyo Research Laboratory,
5-19 Sanbancho, Chiyoda-ku, Tokyo 102, Japan

Abstract

COB(C with OBjects) is an object-oriented extension of C developed at IBM Research, Tokyo Research Laboratory. Main emphasis of COB is on program organization, compile-time type checking, reusability, and good run-time performance. COB uses classes as the primary means of organization and as the basic unit of object-oriented programming. One of the unique features COB introduces is the notion of class views. The programmer can use views to control the visibility of class information and an appropriate balance between run-time performance and the need for recompilation. This paper explains the notion of class views of COB in detail.

1.Classes

COB uses classes as the primary means of organization and encapsulation. Classes are the basic unit of object-oriented programming. They can also be used as modules in a standard sense to encapsulate data and related operations.

The following example displays the form of a class definition. A class defines a data type; instances of the data type are *objects*. A *class definition* is divided into three sections: a *common section*, an *instance section* and an *implementation section*. The common and instance sections have *private* and *public* parts.

```
class stack {
 common::
   private:
      int stack_num=0;
      void error(char *message);
   public:
      int number_of_stacks(void);
 instance::
   private:
      char element[100];
      int top;
    public:
      void init(void);
      void push(char x);
      char pop(void);
implementation::
      void error(char *message)
   {printf("%s\n",message);exit(1);}
          .....
};
```

The common and instance sections contain data and function declarations and definitions as their *members*. The implementation section contains definitions of functions declared but not defined in the previous sections. The common members are shared by all objects of the class; the instance members are created separately for each object. Private members can be accessed only from the inside of the class, i.e. from the bodies of member functions; public members of the class are accessible from anywhere in a program.

Variables of class `stack` can be declared as follows: "`class stack x, *p;`", where `x` is a variable of class `stack` and `p` is a pointer variable to an object of class `stack`. One can reference instance members by qualifying them with the object, for example "`x.push('A')`", and "`p->pop()`". One can reference common members using their names followed by "`@`" and the name of the class. A public common function may be declared as *global*; the name of a global function must be unique in the global scope, unless the function name is *overloaded*, so the the function may be referenced without "`@`" and the name of the class.

COB supports *multiple inheritance*; an object of a subclass inherits the instance members of all its superclasses, both direct and indirect. Inheritance only affects the instance section of a class.

The details of classes and multiple inheritance are explained elsewhere[1,2].

2. Views

When a large program consists of many classes, each class should have a well-defined interface, and should be used only in accordance with this interface. It is not always straightforward to decide what information about a class should be specified in its interface. On the one hand, an interface that contains a lot of detail per-

mits efficient and convenient access to the class. On the other hand, an interface that contains too much detail may have to be changed when the implementation of the class is changed, in which case the programs that rely on the interface may have to be changed as well. The proper level of detail in the interface depends on the priorities assigned to efficiency, convenience, and modularity.

In principle, the *class declaration* of a class can be used as the class interface. There are two problems with this. First, the compiler uses the class declaration not only to determine what operations on the class (such as variable accesses and function calls) are permitted, but also how the class should be represented. As a result, a class declaration generally contains too much detail to serve as an interface. Second, no single class declaration can contain the right details to serve as an interface for all possible uses of the class. For this reason, COB supports *class views*.

A view of a class resembles the declaration of that class, except that it contains only a *subset* of the public functions and variables available in the original class. A view can also specify superclasses that may be a subset of superclasses defined in the original. A program that uses a class view as the interface to that class can use the class only through the functions and variables visible through the view. As a result, the program does not need to be changed when the declaration of the class is changed in a way that does not affect these functions and variables. A class can have any number of views and each such view is treated as a type different from the original class. An object variable or a pointer variable that accesses an object through the interface given by a view must be declared to be of that view type. Different views of the same class as well as the original class are *assignable* to each other; hence objects can be assigned to variables of these types without cast operators.

3. Committed and Delayed Bindings

Views are widely used in databases; their application to object-oriented languages was suggested by [3]. COB extends the notion of views by placing the time of binding between a view and the original class under programmer control.

A program that uses a class view may request either *committed* or *delayed* binding. In committed binding, the compiler uses the actual class declaration to determine the representation of the class *at compile-time*; in delayed binding, the compiler generates code that will determine the representation of the class *at run-time*. This allows the programmer to choose an appropriate balance between run-time performance and the need for recompilation. The time of binding does not affect the semantics of the program using the view.

Committed binding requires information on the original class be available in the compilation unit in which the view is declared and used. This is typically enabled by creating a header file containing the original declaration and by including the header file into the program using the view. With committed binding, the compiler is able to determine the object representation of the class given by a view.

When the original information is not available in the compilation unit, the compiler assumes delayed binding. In this case, the user program must be able to determine the representation of the class at run-time. This is done using the *view table* for the view in question. The view table contains an address or an offset for each function and each variable visible through the view. Given a class declaration and a class view of that class, it is possible to construct the view table for that view. This process is called *view compilation*. View compilation is typically performed by compiling a header file containing both an original class declaration and a view declaration.

These two bindings have essentially the same semantics; a program using delayed binding produces the same result if the binding is changed to committed binding. With delayed binding, however, it is not possible to create an object in the stack, nor to introduce its subclasses. These restrictions are necessary as the representation is not determined at compile time in delayed binding. Objects of the class given by a view can still be created in the heap in this case.

Committed binding has an advantage in that access to members of objects of the class is as efficient as using the original class, while it is disadvantageous as a change on the original class declaration will make recompilation of the user program of its views necessary. On the other hand, delayed binding is slightly slower and use of views is restricted, but a change on class declaration will not affect the program using its views if views remain the same. It simply needs recompilation of view tables.

References

[1] Kamimura, T., et al., *An Overview of COB*, JSSST WOOC '88, March, 1988.

[2] Kamimura, T., *Object-Oriented Extensions of ProcedurL Languages*. Joho-Shori, vol.29, no.4, pp310-317, 1988.

[3] Hailpern, B. and Nguyen V., *A Model for Object-based Inheritance*, IBM Reserach Report RC12481, January, 1987.