オブジェクト指向言語COB − 設計方針と言語仕様

3E−8

小野寺 民也 上村 務
日本アイビーエム東京基礎研究所

1 Introduction

The popularity of C-based object-oriented programming (OOP) has grown in recent years. In particular, C++ [5] is rapidly gaining acceptance by a large number of users. As these users accumulate experience, they have found certain common problems, which can be summarized as follows:

- *Intractable run-time errors.* Major sources are type misinterpretation, illegal pointer dereferencing, and failure of storage management. The errors usually result in a segmentation fault, and disable even a debugger.
- *Long recompilation time.* More often, a slight modification to a source code causes a massive recompilation.
- *Limited modularity and reusability.*

COB (C with OBjects) is a new object-oriented language, upward compatible with C, that is being developed at IBM's Tokyo Research Laboratory. The language attempts to address the above problems by putting as much emphasis on facilitating programming activities as on the quality of the final object code. In particular, the following design goals have been set up.

- Decreasing the number of error sources by providing safe language constructs and garbage collection.
- Increasing the modularity of program components.
- Maintaining compatibility with C.
- Recovering good run-time performance by extensive optimizations in a completed program.

2 Major Features

We describe in this section major features of COB such as classes, inheritance, objects, and type safety. We does not cover a number of features in COB, among which are exception handling and variable-sized objects. See [1] for these features.

```
class C < X {  // interface
    int pub_ivar;
    int pub_ifunc(void);
  common:
    int pub_cvar;
    int pub_cfunc(void);
};

class impl C < Y {  // implementation
    struct Tag { ... }; // local to class C
    typedef struct Tag *TagPtr; // local to class C

    TagPtr priv_ivar;
    TagPtr priv_ifunc(void);
  common:
    int priv_cvar;
    int priv_cfunc(void);
  definition:
    int priv_ifunc(void){...}
        ...
};
```

Figure 1: Class Definition in COB

2.1 Classes

COB promotes the organization of programs around *classes* in a similar way to C++. A class defines a type, instances of the type are called *objects.*

A class is defined by its interface and its implementation. The greatest differences from C++ are that a class interface specified by using the `class` construct contains only public declarations and that a class implementation given by the conjugate construct of `class impl` includes all the private declarations as well as the function definitions in the class. Figures 1 shows an example of a class definition in COB.

The implementation of complete separation is presented in [3] together with performance results.

2.2 Inheritance

COB supports multiple inheritance. A class may be inherited by another class as a *private* or *public* superclass. A public inheritance introduces *subtyping.* That is, a superclass becomes a *supertype* of a class, which is in turn a *subtype* of the superclass.

The COB Programming Language - Design Goals and Major Features
Tamiya Onodera, Tsutomu Kamimura
IBM Research, Tokyo Research Laboratory

The overall emphasis as regards inheritance in COB is on simplicity and resuability of classes. First, every public instance function member of a superclass can be redefined in a subclass, or is virtual in C++ terminology. Second, in order to resolve conflicts of member names inherited from superclasses, COB allows inherited members to be *renamed.* Renaming is also used to give more appropriate names in a subclass.

```
class Stack < LinkedList (insert as push, get
as pop) {...}
```

2.3 Object Creation and Deletion

The semantics of objects and variables of a class type is also different from that of C++, and is close to that of CLU [4]. Objects are created only in a heap by the common function `new`, variables of a class type are just used in a program to refer to objects, and the garbage collector automatically reclaims unused objects. The implementation and performance results are presented in [2].

The two instance functions `init` and `final` play special roles in object creation and deletion. The function `new` is automatically generated by the translator from the `init` function. When the garbage collector destroys an object, the `final` function is automatically called. The garbage collector also destroys any references to other objects from the garbage, if any.

2.4 Type Safety

While the type conversion from a subtype to a supertype is implicit and always valid, the conversion from a supertype to a subtype, called a *downward conversion*, must be made explicit with the cast operator. If the conversion is invalid, this causes a run-time type conversion error. Two safeguards are also embedded into translated code to maintain memory and type consistency: an array bounds check and a null dereferencing check.

COB provides a generic class `Any`. A variable of the type `Any` can take on a value of any class type, and can be made specific at run time in the same way as in downward conversion.

3 Rationale for the Design

The heap-only allocation of objects brings significant benefits to COB. First, it makes the semantics of objects and variables of a class type clean and simple. Second, it eliminates the absolute need for the translator to know the size of a class. This is a valuable step towards the complete separation of interface and implementation. Finally, it allows a faster and safer garbage collection: it is no longer necessary to decide whether objects come from the stack or the heap. C++ was not designed in this way, mainly because of the efficiency of stack allocation ([5], p.145). However, experience with the early version of COB, which allows stack allocation, shows that very few objects are automatic.

Interface simplification has a strong impact on program development. Experience also shows that many of the modifications to a class declaration only involve private members or private superclasses. All of them led to tedious recompilation in the early version of COB, as in C++. This problem of long recompilation can be avoided by a complete separation of interface and implementation.

Besides, every function member being redefinable promotes reusability, since a class is often reused by defining a slightly different version of an instance function of the class. The keyword `virtual` in C++ is a hint to the translator for generating efficient code, but it prevents reuse: programmers need to edit the source code when they want to redefine an instance function that is not declared as `virtual`.

4 Conclusion

Initial experience with COB indicates that support of high-level constructs, such as run-time type checking and garbage collection, provides significant advantages in many areas of application.

Acknowledgements

We would like to thank Nori Suzuki, Carl Hewitt, Brent Hailpern, and members of the programming language group for their helpful criticisms and discussions.

References

[1] *COB Language Manual.* IBM Research, Tokyo Research Laboratory, 1990.

[2] Kuse, K. et al. Automatic Storage Management System in COB. *to appear in 41th Annual Convention of IPSJ.*

[3] Yasuda, K. et al. The COB Programming Language - Implementation. *to appear in 41th Annual Convention of IPSJ.*

[4] Liskov, B. et al. *CLU Reference Manual.* Springer-Verlag, 1981.

[5] Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, 1986.