# Is Multiple Inheritance Essential to OOP?
(PANEL)

**Yen-Ping Shan**, *IBM, PRGS Cary*, (moderator)
**Tom Cargill**, *Independent Consultant*
**Brad Cox**, *George Mason University*
**William Cook**, *Apple computer*
**Mary Loomis**, *Versant Object Technology*
**Alan Snyder**, *SunSoft, Inc.*

## 1 Background

Multiple inheritance has been adopted by many OO programming languages (such as C++, CLOS, and Eiffel). On the other hand, there are languages (such as Smalltalk, Objective C, and Object Pascal) that do not offer multiple inheritance. Despite missing the feature, these languages seem to be as effective as those that offer it. It is then natural to ask how essential multiple inheritance is to object-oriented programming.

This panel will discuss this question from a variety of perspectives, including but not limited to:

- Programming languages
  What's the definition of multiple inheritance in a specific language, why is it necessary, and how is it used?

- Modeling and OO Design
  How is multiple inheritance used in modeling and design? Does it help in managing the complexity and delivering better design?

- Object users and providers
  Strong separation between the interface and implementation has become the trend of many new languages (such as OO COBOL) and systems (OMG's CORBA). What effect does this have on the importance of multiple inheritance?

As part of their positions, the panelists are asked to address specific questions related to multiple inheritance. Examples of such questions are:

- Does an object that fulfills multiple roles make sense? If so, how does one describe such an object and how does one implement it? Is multiple inheritance essential to both description and implementation?

- What's the future of multiple inheritance?

- How should the audience confront multiple inheritance in their everyday practice?

- Does it make sense to rate and compare products based on whether they support multiple inheritance?

The panelists will discuss conceptual and practical aspects of the issues drawing from their personal experience.

## 2 Tom Cargill

The question as stated is very broad: the semantics of multiple inheritance (MI) and the interaction between MI and other language features vary considerably from one programming language to another. Rather than tackle the general question, I will make some observations about whether MI is essential to object-oriented programming in C++. I choose this restriction because most of the MI code that I have studied has been written in C++, and I believe I can make a more detailed statement in this limited context.

The reaction of most C++ programmers to MI is that it must be for expressing relationships among classes that belong to rich classification hierarchies. That is, programmers look for classes corresponding to abstractions that exhibit more than one specialization (is-a-kind-of) relationship. The two situations that superficially look most promising are "multiple classification" and "dynamic classification."

Multiple classification arises when an abstraction, throughout its lifetime, is a specialization of more than one other abstraction. Should a multiple specialization relationship be represented by multiple inheritance? Unfortunately, multiple inheritance turns out to be an unwieldy way to model multiple classification. On the other hand, multiple classification is simplified by viewing the various attributes of the class independently, and composing those attributes to form an object. Programming with a composition of attributes is generally simpler, more flexible and more expressive than attempting to model multiple classification with multiple inheritance.

Dynamic classification arises when an abstraction participates in different specialization relationships at different phases of its lifetime. For example, at some times a seaplane is a specialization of a boat; at other times a seaplane is a specialization of a plane. Dynamic classification is not supported by any of the inheritance mechanisms of C++, because every object is of precisely one type, determined at the time of the object creation. No metamorphosis is permitted; an object cannot modify its type dynamically. Dynamic classification can be expressed in C++ by the use of delegation, instead of inheritance. Delegation is relatively simple: one object receives messages and propagates them to another object, which then performs the delegated work. Using delegation, a SeaPlane object may behave like by a boat by delegating incoming calls to a Boat object.

Multiple classification and dynamic classification arise when modeling "natural" classes -- those that correspond to abstractions from the problem domain. Object-oriented systems also need "synthetic" classes -- those that do not correspond to abstractions found in the problem domain. Synthetic classes emerge during design and coding of a system, in response to internal, synthetic needs of the software.

The contexts is which I have seen and found MI to be useful in C++ programs have all involved synthetic class relationships. For example, coding callbacks from a server, such as graphics widget, to an arbitrary client is simplified by the use of an abstract class that defines the callback protocol. The static type system of C++ demands that a server performing the callback know the type of its client. If the client class inherits from the abstract protocol must receive multiple callback protocols, it must inherit from multiple abstract protocol classes. This is multiple inheritance that arises independent of any consideration of classification relationships. Indeed, this application of inheritance would be alien to a programmer familiar with only dynamically typed languages.

On the basis of the code that I have encountered to date, I conclude that where multiple inheritance is useful in C++ programming it is not for modeling natural class relationships, but as an implementation technique used in conjunction with synthetic classes. That is, I see multiple inheritance as an implementation convenience, not an essential modeling tool.

## 3 Brad Cox

Is multiple inheritance essential to object-oriented programming?

No, on the grounds that users of languages (Smalltalk, Objective-C) that don't support MI are indisputably engaged in object-oriented programming.

Is inheritance an essential feature of programming languages?

No, on the grounds that programming languages are implementation tools, not specification tools. Unlike encapsulation, inheritance has demonstrated a mixed and often disappointing utility when used for

implementation instead of specification. Its most useful role is a specification tool; a way of classifying objects from their consumer's perspective, which is rarely consistent with that of the objects' developers.

Is multiple inheritance an essential feature of a true software engineer's toolkit?

Yes, on the grounds that specification tools and implementation tools both belong in a true software engineer's toolkit. By contrast, the software craftsmen of today get by with only implementation tools (object-oriented programming languages). And even so, those who are building reusable code have discovered that objects can be viewed from at least two and often more irreconcilable perspectives; the internal perspective of their developer and the external perspective of their users.

## 4 William Cook

In any discussion of inheritance it is essential to define one's terms. This is because inheritance has historically been used to refer to two distinct but related concepts:

1) Subtype/conformance/refinement relationships between types/interfaces/specifications. Illustrated by Cardelli's "semantics of multiple inheritance", knowledge representation hierarchies, and typing rules in C++/Eiffel/Emerald/etc.

2) Derivation on new program units by specifying changes to existing program units. Illustrated by subclassing in Smalltalk, derived classes in C++, inheritance in Eiffel, subclassing in CLOS, prefixing in Beta, etc.

I believe that one should distinguish these concepts. I call the first "subtyping" and the second "inheritance". Given this premise, it is possible to discuss the role of common notions of "multiple inheritance" in greater detail:

1m) Multiple subtyping. This means that one interface can be a subtype/refinement of several different simpler interfaces. This is quite desirable in practice and poses few conceptual difficulties. However, it can be somewhat expensive to implement. In the limit, the subtype relationship is a complex acyclic directed graph which can be extended at any node. The related classes may or may not inherit from each other.

2m) Multiple Inheritance. This means that a single program unit is derived by combining and modifying several existing units. At the limit, this problem resembles a complex merge of several program parts (or versions) into a combined whole -- except that the merge is done with declarations instead of a text editor. Unfortunately, this merge sometimes depends upon the internal details of the parts, causing a breakdown in encapsulation. When the program units are classes, inheriting class may or may not be a subtype of the classes it inherits.

From these observations I argue that subtyping (with multiple supertypes) is probably essential, but that multiple inheritance is not. However, multiple inheritance can be very convenient and lead to advantageous code sharing. For languages (like C++ and Eiffel) that do not separate subtyping and inheritance, one can use the combined mechanism for either purpose according to programming conventions, but the necessity for multiple subtyping drives one to require multiple inheritance too, with all it's complexity and problems.

## 5 Marry Loomis

Multiple inheritance is a mixed blessing. It is a useful construct in object modeling, but can introduce challenges for implementation.

Multiple inheritance enables the modeler to represent the real-world fact that objects can fulfill multiple roles. These roles depend on the vantage points of the viewers. People generally tend to classify real-world objects and concepts into neat buckets, but those classifications sometimes overlap. In many cases, multiple inheritance helps object modelers manage the complexity of representing the real world.

Perhaps ironically, however, multiple inheritance can introduce significant complexity into application design and programming. The developers of the underlying systems software -- including both object

programming environments and object database management systems -- can encounter even greater challenges dealing with multiple inheritance.

## 6 Alan Snyder

The value of multiple inheritance in object-oriented programming has been a subject of much debate. Part of the problem has been that inheritance serves multiple purposes in object-oriented programming languages: it can be used both to define specializations of concepts (interfaces) and to reuse and customize code (implementations).

In the world of distributed systems, there is great value in making a clear distinction between interfaces and implementations. In the OMG CORBA, for example, different languages are used to define interfaces and implementations: interfaces are defined using an interface definition language (OMG IDL), implementations are defined using programming languages. Once this distinction is made, we can analyze the value of multiple inheritance in these two contexts.

In OMG IDL, interface inheritance is the technique used to establish type conformance relationships. By supporting multiple inheritance, OMG IDL makes it easy to define an object whose interface conforms to multiple, unrelated interfaces. This situation easily arises because interfaces can be defined independently by multiple developers. Multiple interface inheritance has no difficult conceptual problems, such as ordering dependencies. Multiple interface inheritance is very convenient, but not strictly essential; without it, one would define multiple objects supporting the different interfaces and provide operations to get one from object to another.

Multiple implementation inheritance, on the other hand, is problematic. The goal is lofty: take several behaviors (implementations), mix them together, and produce a new implementation that provides the appropriate combined behavior. The problem is that unless the behaviors have been designed in advance to be mixed together in this fashion, it probably doesn't work. The implementations tend to interact with each other in many ways that are difficult to control and understand. For example, dependence on incidental ordering is a common problem. It is easy to design a multiple implementation inheritance model that supports any given example; it is not clear how to design one that supports new examples. Component reuse and customization are important goals. However, it is not clear that implementation inheritance is the best mechanism for achieving those goals, especially multiple implementation inheritance.