An Introduction to Object Oriented Programming, Inheritance and Method Combination

Bruce L. Horn

1 January 1988

CMU-CS-87-127(2)

Abstract

Object oriented programming, introduced in the 1960's by Simula, has become an important programming framework for large software systems. Among the attractions of object oriented programming languages is the potential for software reuse through inheritance.

Inheritance allows the programmer to define a new class of objects which inherit the procedures and state descriptions of an existing class. The programmer may then modify and extend the behavior and state of the base class. Because state information is static, subclassing state is straightforward: simply append new state information. However, behavior is dynamic; the problem of subclassing behavior is a bit more complicated, since it may not be correct to simply concatenate the behaviors of the superclass and the subclass and execute them sequentially. This issue is referred to as the method combination problem.

In this paper, an overview of object oriented programming will be presented, with a description of the philosophy of object oriented programming. The different approaches to inheritance will be compared, followed by a discussion of the method combination problem.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the

Avionics Laboratory
Air Force Wright Aeronautical Laboratories
Aeronautical Systems Division (AFSC)
Wright-Patterson AFB, Ohio 45433-6543

Introduction

Object oriented programming, introduced in the 1960's by Simula [Dahl66], has become an important programming framework for large software systems. Among the attractions of object oriented programming languages is the potential for software reuse. Programming environments such as the Smalltalk-80 environment [Goldberg83] provide programming tools and examples in a set of pre-defined system classes, similar to a software library. Programming in such a system consists of directly using existing classes by creating instances of classes (new objects), writing new classes from scratch, or creating derived classes by subclassing existing ones.

Subclassing, or inheritance, allows the programmer to define a new class of objects which inherit the procedures and state descriptions of an existing class. The programmer may then modify and extend the behavior of the base class, and add new state information, if desired. Because state information is static, subclassing state is straightforward: simply append new state information. However, behavior is dynamic; the problem of subclassing behavior is a bit more complicated, since it may not be appropriate to simply concatenate the behaviors of the superclass and the subclass and execute them sequentially. This issue is referred to as the method combination problem.

1.0 An Overview of Object Oriented Programming

The main difference between object oriented programming and traditional programming is that object oriented programs are composed of *objects*, packages of data and related procedures. Typically, a programmer using an object oriented language thinks primarily of the data structures, followed by the procedures that act upon those structures. Traditional programs are composed of a collection of procedures which are independent of the data and related only by convention; programmers using traditional languages often consider first the procedures, and then the data structures.

Typically procedures act only on certain types of data; for example, the procedure `DrawString(str)` expects a `String` as its argument, and `DrawRectangle(rect)` expects data of type `Rectangle`. Although compilers can check to see if the arguments are correctly typed, it is still up to the programmer to determine which procedure is the correct one to use in each case. By closely relating data and the procedures that act upon the data, object oriented languages alleviate this problem. For example, in an object oriented system, drawing a string could be done by executing `Draw(str, location)`, and drawing a rectangle by executing `Draw(rect, location)`; finding the correct procedure to execute is handled by the support system of the language[1] and is based on procedure's actual parameters. This seems like a minor advantage, but having language support for choosing the appropriate procedure contributes strongly to the usability of the language, and allows the programmer to focus on the semantics of the expression, rather than the appropriate syntax.

Object oriented programming has been used traditionally in simulation systems, due to the ease of describing simulation entities and their relationships as objects. More recently, user interface systems, including window managers, have been successfully designed using object oriented languages and programming styles.

1.1 Terminology

Object oriented programming is infamous for introducing new terminology, and there are many different terms that are used for similar concepts in different languages. Smalltalk-80 [Goldberg83] is one of the earliest and most successful object oriented systems; for consistency throughout this paper the Smalltalk-80 terms, usage, and syntax will be used. However, the syntax of program examples will diverge slightly from true Smalltalk syntax for readability.

An *object* is the fundamental data structure in object oriented languages. In a homogeneous system such as Smalltalk or Oaklisp, everything in the system is an object. However, most object oriented systems are not homogeneous. Flavors [Moon85] CommonLoops [Bobrow85], and CommonObjects [Snyder85] are implementations of object oriented programming added to CommonLisp, an existing language; in these sytems programming can be a mix of object oriented, procedural, and functional styles[2].

A description of the structure and behavior of a set of objects is called a *class*. The class describes those aspects of its instances which are shared. Objects are created from classes through *instantiation* in most languages, but they can also be created by *cloning* a prototypical object (in Actor [Agha86], for example). An object of a given class is called an *instance* of that class. While instances share the class structure and behavior, each instance may have different state.

The physical structure of an object is defined by its *instance variables*. These are the slots in the object which may be filled with values or references to other objects. Object behavior is defined through *methods*, procedures which are run in response to a *message* sent by another object. Methods may directly access the instance variables of the *receiver*,

[1]In a typed language, this can often be determined at compile time, while typeless languages such as Smalltalk must rely on a run-time lookup.

[2]Functional programming does not allow side effects, while procedural programming does. Object oriented programming manipulates local state as well as (possibly) global program state.

the object to which the message was sent. The receiver is the entity which, along with the message name, determines the correct method to run.

Messages may have *arguments* which are passed along to the method. In the previous example, executing `Draw(str, location)` would consist of sending the message `Draw` to the object `str`, the receiver, with the argument `location`.

In most cases only the receiver and the message are used to determine which method to run. Some systems, such as CommonLoops, may use the classes of all of the arguments to the message to find the correct method.

The syntax of message sending varies from language to language. Sending the message `fill` with arguments `color` and `mode` to the object `myRectangle` is done in several different languages as follows:

`myRectangle fill: color transfer: mode`	Smalltalk
`(fill myRectangle color mode)`	Oaklisp, CommonLoops
`(=> myRectangle :fill color mode)`	CommonObjects

Note that "Sending the message `fill` to the object `myRectangle`" means "run `myRectangle`'s method named `fill` on itself." This terminology is unfortunate in that it implies objects may concurrently execute methods along with other objects, which is quite uncommon in object oriented languages[3].

The names of messages sent to objects, called *message selectors*, are the object's interface to the outside world. The important aspect of a message selector is that it is only a name for the desired action, describing what the programmer wants to happen, not how it should happen. Dan Ingalls, one of the creators of Smalltalk, states that "The message sending metaphor provides modularity by decoupling the intent of a message embodied in its name from the method used by the recipient to carry out the intent." [Ingalls80]

The Smalltalk syntax is quite different than the different Lisp dialects; the receiver is first, followed by the message and arguments in infix form. The full message selector in the Smalltalk example is `fill:transfer:`; the arguments appear after keywords that terminate with colons.

The set of messages that can be handled by a particular object is defined by the object's class, and is called a *message protocol*. A given class may implement several protocols, for drawing in windows, writing to and reading from streams, accessing state, and so on.

[3]For an interesting exception see [Agha86].

Many different classes may share a message protocol, but define different methods to execute the messages. For example, operations such as < may be defined in several different classes: The class `Integer` would provide a method to implement a standard integer comparison, while the class `String` would provide a method to implement a lexicographic ordering. Such an operation, which can be defined over a set of different classes, is called a *polymorphic* operation.

Polymorphism also occurs with *parameterized classes*. Some languages, notably Trellis-Owl, support the concept of parameterizing a class over another class; for example, an `OrderedList` may have elements of any kind as long as they are all uniform: `OrderedList of Integer` or `OrderedList of Real`. In this instance, `OrderedList` is the parameterized class, and `Integer` and `Real` are the parameters which are used to determine the class of the elements of the `OrderedList`. These classes, once parameterized, are considered completely separate classes; however, they still implement the same message protocols, and are therefore polymorphic[4,5]. Ada generic packages provide similar functionality.

Subclassing is the procedure by which refinement of the descriptions of objects is accomplished. While a class defines the structure and behavior of a set of objects, a subclass of that class inherits the structure and behavior, and may extend or modify it as well.

1.2 Encapsulation

Encapsulation describes techniques which minimize dependencies of modules on each other by defining interfaces which are used for communication. Effective encapsulation allows programmers to make compatible changes within modules without fear of changing other behavior outside of the module.

In object oriented systems, the message selectors provide the object's interface, while the instance variables and methods define the object's implementation. The use of encapsulation provides the ability to hide an implementation of a class of objects while exporting an interface to them. Some languages, notably Smalltalk, allow only the methods defined by the receiver to access the receiver's instance variables directly. Methods in other classes that want access to the state of an object that is not the receiver must send a message to the other object. This restriction supports encapsulation for objects in the system.

[4] In systems like Smalltalk which look up methods at runtime, `OrderedList` can hold entries of differing types, as long as they implement the messages required to be in an `OrderedList` (< and =, typically).

[5] See [Schaffert86] for an example of parameterized types.

This protection mechanism promotes modularity in that clients need not know the object's implementation, just the message protocols it supports (e.g. its interface). Programmers unfamiliar with object oriented programming may say that they can do the same thing by defining a data structure with access procedures. This works as long as the convention of manipulating the structures only with the access procedures is followed; as soon as the data is directly manipulated in any place in the system, the modularity is broken. Similarly, modules with private variables provide the same kind of protection as object oriented programming, but the programmer is usually restricted to a single instance of the module within a system; since the set of private variables is owned by the module itself, it is impossible to have more than one set of the private variables[6].

Being required by the system to send a message to an object to find out something about the object's state is generally considered expensive. Simula and C++ [Stroustrup86] allow the programmer to partition a class's instance variables into public and private segments; public instance variables may be read and written by anybody, while the private variables are accessible only through the object's methods. Similarly, CommonObjects and Flavors allow the programmer to specify *initable*, *gettable* and *settable* instance variables which provide clients direct access to the object's state. Trellis/Owl [Schaffert86] allows the programmer to define *fields* of an object to be directly accessible.

1.3 Localization of Behavior

Factoring system functionality into classes and derived classes allows the programmer to define certain shared behaviors in only one place. For example, sorting can be done by comparison of objects (< and =). If a method is defined in an Array class that implements a sorting routine using < and = on the objects in the array, any group of objects may be sorted as an array as long as they understand (e.g. implement methods for) the messages < and =. The sorting routine need not exist anywhere else in the system[7,8]. This ability to localize functionality contributes to the compactness and ease of maintenance of object oriented systems[9].

[6]The Mesa language is an exception which provides for multiple instantiations of module state.

[7]Of course, if the collection is not homogeneous, there is the problem of determining whether, for example, a `number` is less than a `picture`. Surprisingly, this kind of comparison *does* occur in user interface systems.

[8]Where parameterized typing is available, such as in Trellis/Owl, one would generally define a `SortedArray` that takes a type as a parameter. That type would then be required to provide the appropriate comparison operators (<, =), and the availability of the operations could be determined at compile time.

[9]It may be useful to generalize this concept further, by parameterizing the sorting operation over the comparison operator. In this way *any* operator may be provided that compares two objects; for example, you may want to reverse the sorting order by providing the > operator instead.

1.4 Programming in an Object Oriented System

Programming in an object oriented system is a modeling process. The programmer describes, through classes, the objects that make up the program, and defines the messages and methods that will be used to communicate among the different kinds of objects. Objects are built up from other objects, and a natural construction hierarchy can be made, from low level objects to high level ones. Along another dimension, classes are refined through subclassing, thus localizing the structural and behavioral differences between each subclass and making it simpler to maintain the system.

1.5 An Example Class: `Window`

Object oriented systems have been shown to be useful in simulations and in building user interfaces. A typical class defined for a graphical user interface might be structured in the following way:

```
Class: Window
    SubclassOf:     Object
    InstanceVars:   visible, title, topLeft, botRight, scrollBar, pen
    Messages:       show, hide, draw, height: ht width: wd,
                    moveTo: point, bringToFront
    Methods:
    show [ ...code for show here... ]
    hide [ ...code for hide here... ]
    ...
```

This description of a `Window` says that each instance of `Window` has five parts (`visible..pen`), and can respond to seven messages (`show..bringToFront`). In fact, `Windows` will have more parts and be able to respond to more messages, depending on what is inherited from `Object`. Each part of a `Window` will typically be another object: `visible` might be a `Boolean` object, `title` a `String` object, and so on. Usually all that is required is that each part be able to respond to the messages used (`draw` for `String`, and so on).

Objects are instantiated in Smalltalk by sending the message new to the class[10]. The following is an example of creating a window and then setting the window's size:

```
myWindow <- Window new.
myWindow height: 200 width: 400.
```

2.0 Inheritance

Inheritance is the main concept that supports refinement and software reuse in object oriented languages. The ability to inherit state structures and behavior from an existing class allows the programmer to define new objects in the system not only in terms of existing objects, but also by modifying and mixing the descriptions of existing classes. The goal of object oriented programming is, in general, to factor into separate modules the different implementation details and behavior of abstract types; the goal of inheritance in particular is to support refinement of classes into derived classes, or *subclasses*.

The pioneering object-oriented language Simula [Dahl66] introduced the concepts of class and inheritance. A class may be prefixed by a superclass, thereby inheriting its description, and *virtual procedures* or methods defined in the superclass can be specialized by procedures in the subclass. In Simula (as well as C++), procedures must specifically be declared virtual so that they may be overridden by subclasses. Taking this concept to the limit, languages such as Smalltalk and Oaklisp make all methods virtual and overridable.

There are many advantages to inheritance in programming languages. Thomsen [Thomsen86] states them as follows:

- Better Conceptual Modelling. Since specialization hierarchies are very common in everyday life, direct modelling of such hierarchies makes the conceptual structure of programs easier to comprehend.
- Factorization. Inheritance supports that common properties of classes are factorized - that is, described only once and reused when needed. This results in greater modularity and makes complicated programs easier to comprehend and maintain since redundant description is avoided.
- Stepwise Refinement in Design and Verification. Inheritance hierarchies support a technique where the most general classes containing common properties of different classes are designed and verified first, and then specialized classes are developed top-down by adding more details to existing classes.
- Polymorphism. The hierarchical organization of classes provides a basis for introduction of parametric polymorphism in the sense that a procedure with formal parameter of class C will accept any C-object as actual parameter, including instances of subclasses of C.

[10] In Smalltalk, classes are also objects, and are themselves instances of a *metaclass*. Sending a message like new allocates memory for the instance and can initialize the instance variables as well.

2.1 What is Inherited?

A class inherits instance variable declarations as well as methods from its superclass. If an instance were to be created from the derived class immediately after subclassing, it would act exactly as an instance of the superclass, since its behavior is completely inherited. By adding new instance variables and new methods, and by overriding or augmenting superclass methods, the new class's attributes may be refined using the superclass as a base. In Smalltalk, *class variables* are also inherited; these variables are maintained by the class itself and shared by all of the instances of the class, as well as instances of subclasses.

Some systems allow the programmer to decide which methods are allowed to be inherited for use by subclasses, and which methods are visible outside the class by users. In Trellis/Owl, *private* operations are not inheritable at all, and are invisible to users of the class. *Public* operations are both inheritable and visible to users, while *subtype-visible* operations may be inherited but are invisible to users. These distinctions help to add clarity and type safety.

When a method is redefined by a subclass, usually it is a specialization of the superclass method, providing additional code to be run in some order. As noted in [Kristensen], most object oriented languages only provide for a redefinition of the superclass method, with method combination facilities to explicitly run the superclass method from the subclass. Although this is extremely flexible, forcing the programmer to explicitly execute the superclass method can lead to errors; it would be more desirable if the language supported the specialization of methods automatically. Simula provides this to a certain extent with its *inner* construct--code for methods in subclasses is textually surrounded by code in the superclass, with the insertion point given by the programmer as an `inner` statement. (A disadvantage of this scheme is that the superclass must know that it is going to be subclassed, and what kinds of refinement are expected.) These kinds of facilities will be discussed in detail later in the paper.

2.2 How is Inheritance Used?

Inheritance is used in several different ways. A subclass can be modified to provide different or additional behavior from its superclass. For example, creating subclasses `Rectangle`, `Oval`, and `Polygon` from the class `GraphicalObject` allows the programmer to specialize existing methods for `GraphicalObject` such as `display`, `resize`, `erase`, and `move`. A class such as `GraphicalObject`, which would never have instances itself, is called an *abstract class*, and is designed explicitly to define an interface to a particular kind of object; the actual implementation of that interface must be defined in subclasses. More typically, subclassing is used to extend a particular class with additional functionality; one might create a `FramedWindow` from an existing class `Window` as follows:

```
Class: FramedWindow
    SubclassOf:     Window
    InstanceVars:   frameWidth
    Messages:       frameWidth, frameWidth: width
    Methods:
    frameWidth [...code to return the frameWidth here... ]
    frameWidth: width [...code to set the frameWidth here and update the display... ]
    draw [...code to specialize the draw method in Window here... ]
```

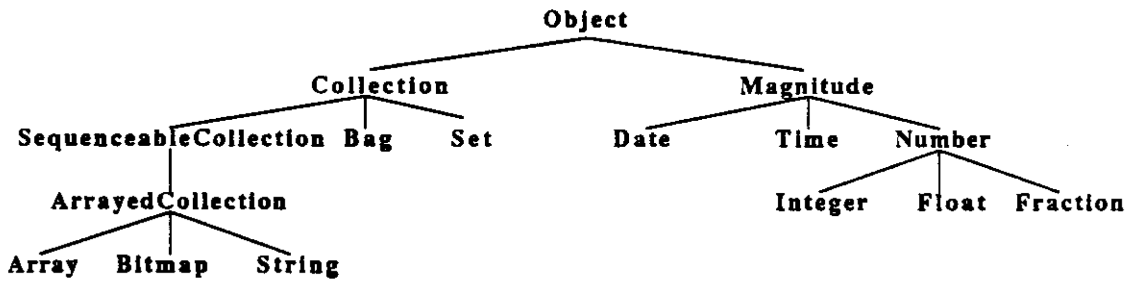
`FramedWindow` defines two new messages, `frameWidth` (to read the width) and `frameWidth: width` (to set it), and overrides the `draw` method of `Window` to provide the new behavior of drawing a frame. Since `FramedWindow` is still a `Window`, instances of `FramedWindow` have all of the instance variables defined in `Window` as well as the new instance variable `frameWidth`. Clients of `FramedWindows` can send all of the `Window` messages, as well as the two new messages for setting and reading the frame width.

2.3 Inheritance Structures

2.3.1 Hierarchical Inheritance

The simplest and most widely used inheritance structure is a strict hierarchy in which classes may inherit only from a single superclass. Most current object oriented languages, such as Smalltalk and C++, support only hierarchical inheritance. For many applications this is sufficient; for example, the entire Smalltalk-80 system was written with only hierarchical inheritance. The mechanism is simple, efficient, and straightforward, but limited in expressibility: some desired relationships are impossible to describe in a hierarchy. Extended mechanisms such as multiple inheritance provide more functionality, at the expected cost of increased complexity.

The following is a part of the Smalltalk-80 class hierarchy, including the top-level class Object and two of its immediate subclasses, Magnitude and Collection, and some of their subclasses. From examining the inheritance hierarchy, it is reasonably straightforward to discern what the attributes and messages defined in each class might be.



2.3.2 Inheritance by Delegation

In some languages, such as Actor [Agha86], objects are created by *cloning* an existing object. In this case, both the instance variables and the current values of those variables are inherited. If the new object changes the value of one of the inherited variables, it makes a copy of the instance variables for its own use, and no longer references its parent object's variable.

Because there is no concept of class, Actor provides inheritance through delegation. This means simply that each object is responsible for both choosing which messages it will handle, and for choosing an object to handle those messages that it is not prepared to handle. This is a very flexible mechanism which can simulate other kinds of inheritance, but the programmer must define the inheritance behavior himself. Other languages without the class concept (such as Object Logo) instantiate objects in the same way as Actor, by cloning a prototype, but handle the inheritance behavior automatically.

Note that objects that inherit from a prototype in the Actor style end up sharing the values of the instance variables of that prototype. Again, this is extremely flexible and useful in some circumstances, but not typically what is desirable. In the building of programming systems it is necessary to provide both rigor and flexibility. Although inheritance by delegation and prototyping are powerful new ways to handle method lookup and inheritance, they may not provide enough inherent structure to support the design of large, maintainable systems. Therefore, this paper will focus on more traditional, structured, class-based languages.

2.3.3 Inheriting from Multiple Superclasses

The ability to inherit from multiple superclasses can be quite useful in certain circumstances; these circumstances, and the consequences and problems that arise from this functionality, will be discussed in detail later in the paper. A more ambitious and flexible scheme is *Boolean classes* [McAllister86], where any boolean combination of classes can define a class. There have not been any systems yet built that implement this concept, so it will not be considered in this paper.

2.4 Encapsulation Issues with Inheritance

Since a subclass inherits the instance and class variables of its superclass, it will often be convenient for a method to operate on them. However, it is important that the subclass only access the superclass through the messages defined by the superclass, just like any other client; directly accessing the variables can cause problems. For example, Smalltalk allows subclasses to directly access all instance variables. In Smalltalk, if a programmer removes an instance variable from a superclass, all subclasses will need to recompile their methods since they access their own instance variables by offset from the beginning of the object. An even more serious problem occurs when a subclass method actually *uses* the deleted instance variable: since the subclass was dependent on the implementation of the superclass, the method will have to be recoded.

By forcing objects to access their instance variables defined by their superclass through the defined interface, these problems do not occur. However, it may be necessary to define special methods for accessing superclass state from the subclass which may not be appropriate for other clients of instances of that class. One solution is to be able to declare these subclassed methods as special as in the Trellis/Owl *private* methods, so that they do not appear in the external message protocols of the superclass[11]; Simula provides this capability as well with the `hidden` attribute.

2.5 Typing Issues with Inheritance

Although inheritance has primarily been considered a programming convenience, providing a nice way to reuse and refine existing code, it may also be considered to be part of a typing system [Schaffert86]. For example, in Simula, if a class `FramedWindow` inherits from `Window`, instances of `FramedWindow` may be used anywhere a `Window`

[8]C++ provides a mechanism, called *protected*, to accomplish this. Hiding the superclass implementation from subclasses is an important aspect of the design of CommonObjects. An excellent discussion of these issues can be found in [Snyder86].

may be used, since `FramedWindows` can be expected to have at least the behavior of a `Window`. A slot or parameter whose type is set to be `Window` may actually be any subclass of `Window`. Clients then can use their knowledge of the class hierarchy (e.g. that `FramedWindow` is a kind of `Window`) in the use of the code.

Unfortunately, viewing inheritance on the one hand as a type system, and on the other hand as a programming convenience, is somewhat problematic. Client code that uses knowledge of the class hierarchy can cause problems. Snyder [Snyder86] says

> If the use of inheritance is part of the external interface, then changes to a class definition's use of inheritance may affect client code. For example, consider a class that is defined using inheritance. Suppose the designer of that class decides that the same behavior can be implemented more efficiently by writing a completely new implementation, without using the previously-inherited class. If the previous use of inheritance was visible to clients, then this reimplementation may require changes to the clients. The ability to safely make changes to the inheritance hierarchy is essential to support the evolution of large systems and long-lived data.

For example, if a programmer changes `FramedWindow` to inherit from a different class, say `FastWindow` (with a different message protocol than `Window`), code which assumed that `FramedWindow` was a specialization of `Window` might not pass a type-checking procedure[12]. The use of the knowledge that `FramedWindow` was a subclass of `Window` in the type hierarchy breaks the encapsulation.

Clearly, if the behavior of `FramedWindow` matches the behavior required by the client of `FramedWindows`, the use should be allowed by the type system regardless of class inheritance. Separation of the concepts of type and class, by using message protocols to denote the type of a class independent of the inheritance structure, is a possible solution. The `windowing` protocol, implemented by `FramedWindow` as well as `Window`, might be the message set `[show, hide, draw, width:height:, moveTo, bringToFront];` the declaration of this protocol as implemented by `FramedWindow` would allow `FramedWindows` to be used anywhere a `windowing` protocol is required. This approach has been implemented in the Emerald system [Black86]. Subclassing and refining protocols is also possible but is beyond the scope of this paper.

Encapsulation and typing issues with inheritance are discussed in detail in [Snyder86].

[12]This would be a problem if the system checked types based on the names of the classes, rather than the actual message protocols supported by those classes.

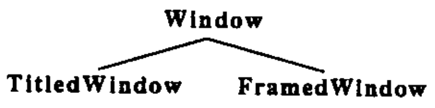
3.0 Multiple Inheritance

The ability to inherit from more than a single superclass is called *multiple inheritance*. In contrast with the simple design and implementation of hierarchical inheritance, multiple inheritance raises quite a few design issues concerned with message dispatch, combination, and typing.

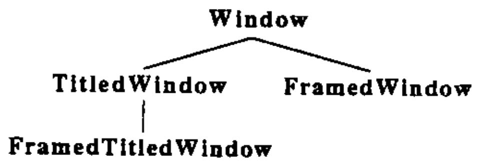
3.1 The Need for Multiple Inheritance

Hierarchical inheritance has been shown to be very powerful and useful in the design of programming systems [Goldberg83]. However, in many cases this restricted inheritance cannot model common system structures as nicely as a multiple inheritance scheme could.

Our classes `Window` and `FramedWindow` can be used to illustrate this problem. Say a new subclass of `Window`, `TitledWindow`, is created. `TitledWindow` defines additional state and behavior for displaying a title on the window, and inherits everything else from `Window`. The class hierarchy looks as follows:

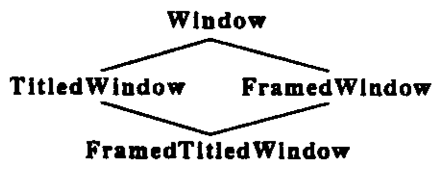


Now, we would like to create a window that is both framed and titled. In a hierarchical inheritance system, since we are restricted to inheriting from a single superclass, we are forced to choose one of the two branches to subclass and copy the code from the other branch to implement its behavior:



Since we copied the code from `FramedWindow`, we now have two places which define the framing specialization of `Window`. We have lost the localization of the behavior of framing. An additional disadvantage is that a `FramedTitledWindow` is not a specialization of a `FramedWindow`, and therefore cannot be accepted as implementing at least `FramedWindow` behavior in a type-checking system.

With multiple inheritance, a different class hierarchy is possible:



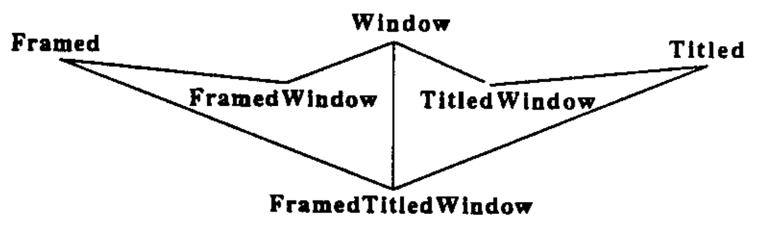
In this case, the behaviors of `FramedWindows` are fully factored; no code is duplicated, and the specialization of `FramedTitledWindow` from its two superclasses is clear.

3.2 Factorization of Orthogonal Attributes

Multiple inheritance allows factorization of the behavior and state in a system, and provides for factoring non-shared attributes, as shown above, in a simple way. Classes may inherit from more than one superclass, resulting in a class description that includes the descriptions of all superclasses (instance variables as well as method definitions).

Subclassing is typically used to refine the implementation of a superclass, by adding new behavior, restricting or modifying old behavior, and adding state. When the ability to inherit from more than one superclass is provided, programmers often inherit orthogonal behavior and state to provide special functionality. For example, given a class `Linker` which links instances together and has the standard linked-list traversal functions, any class can inherit state and behavior from `Linker` to result in a class which instantiates objects which can be put in a linked list. This type of class, used to provide completely orthogonal behavior, is often called a *mix-in*. As with abstract classes, mix-ins are not instantiated, only inherited.

In the `Window` inheritance example, where there are special attributes having to do with framing and titling windows to add to the basic `Window` class, it would make sense to make the attributes available through `Framed` and `Titled` mixins:



Message protocols are often defined by mix-ins, since a mix-in implements a certain kind of behavior that is orthogonal to the other behaviors of the class. In the previous examples, `Linker-Mixin` would implement the `linkable` protocol, while `Titled-Mixin` would implement `window-titleable`. Currently, mix-ins seem to be the only constructs that actually provide support for the concept of message protocols in object oriented languages. They may be sufficient to implement message protocol structures, but making a new language construct that supports the idea of protocols apart from inheritance may be a superior alternative.

4.0 Method Dispatch

How is the correct method chosen when an expression is executed in an object oriented program? A complicated process may be required to find and combine behaviors from various classes, due to the fact that the behavior of an object depends on every class from which it inherits. What determines the method that is expected to run? These questions often may be answered at compile time, but cases exist where it is impossible to determine the exact class of an object, and therefore the correct method to execute, until run time[13]. For example, one might have a collection of `GraphicalObjects` which are created at runtime and then displayed by running through the collection, sending the `draw` message to each element. The system cannot know when the code is compiled what the collection consists of; it may be a `Rectangle`, followed by a `Polygon`, followed by a `Circle`, each with their own different `draw` method.

4.1 Discrimination on the Class of the Receiver

One of the simplest schemes for method dispatch is to determine the method to run based on the class of the receiver. As is true for many design issues in object oriented programming, this is sufficient for most cases, where the message arguments do not change the meaning of the operation. Methods can then be stored in the receiver's class,

[13]However, type safety may still be checked.

and accessed using a simple *message->method* lookup mapping.

Unfortunately, many binary operations are symmetric and polymorphic relative to both the receiver and its argument. For example, `3.0+4.0` is interpreted in Smalltalk as sending the message + to `3.0`, which is of class `Float`, with the argument `4.0`. As long as the argument is also a `Float`, floating point addition is correct. The difficulty begins when the argument is not a float, say an integer. `3.0+4` can no longer be handled by the standard `Float` addition method, and the code in the `Float +` method must handle this case specifically, first noting the type of the argument as `Integer`, and then coercing it to an instance of `Float`.

C++ supports user-defined type conversion for just this problem. In the `Float +` example, the programmer would supply a `Float` ***constructor*** that would create an instance of a floating point number given an integer. Because a `Float` was expected on the right hand side of the +, and because a constructor exists to convert the `Integer` to a `Float`, the C++ compiler automatically generates code to do the conversion.

4.2 Discrimination on Multiple Arguments

CommonLoops can take all of the arguments into consideration when looking up a method for a given message. This mechanism encourages the implementor of the program to consider the message to be an operation based on a set of argument classes, rather than an index to a method defined by the receiver. To handle the addition problem with Integers and Floats, it is a simple, if tedious, task to define four different methods: `+[Integer, Integer]`, `+[Integer, Float]`, and so on.

Discrimination on multiple arguments can be simulated easily by checking the classes of the arguments explicitly in the method, but this solution is less modular[14].

4.3 Method Dispatch with Hierarchical Inheritance

Method dispatching is quite simple with hierarchical inheritance and discrimination on the receiver. If the message has a defined method in the receiver's class, that method is run; otherwise, the system searches for the method in the receiver's superclass. If no method is found anywhere up the superclass chain, a runtime error occurs ("Message not understood").

[14]For a clean solution to this problem using only discrimination on the receiver, see [Ingalls86].

If discrimination on multiple arguments is used, the process is more complicated if the method is not found immediately; there is no unique superclass to check next, and so all possible superclasses of the arguments must be tried in some order in the expectation that some set of classes will be a key to a defined method. This is quite similar to the multiple inheritance method dispatch problem.

4.4 Method Dispatch with Multiple Inheritance

Choosing a method to run in a multiple-inheritance language is not simple. Say the receiver's class doesn't implement the message; where does the system look next? Since multiple superclasses may implement it, there must be a scheme by which a particular method is chosen from a set of possible inherited methods. A typical solution is to enforce a linear class ordering. Oaklisp simply performs a depth-first search of the superclass tree, while Flavors and CommonLoops attempt to flatten the inheritance graph. A depth-first search is simple and efficient, but it doesn't always work well.

The way that Flavors flattens the inheritance graph can be described with three rules:

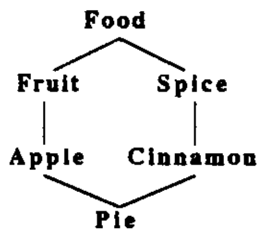
1) A class always precedes its inherited classes.
2) The local ordering of inherited classes is preserved.
3) Duplicate classes are eliminated from the ordering.

To make the ordering, we walk through the graph from the most specific class in depth first order, adding classes to the list. If we can place each new class in the list without violating one of the rules, we add it, otherwise we move to the next node. If we walk through the whole graph without adding all of the classes to the list, we walk through again and add the remaining classes by applying the rules. It is possible that no ordering can be made that obeys all three rules; an error is signaled in this case, and the programmer must re-structure his classes or write his own method dispatch mechanism.

An example modified from [Moon85] shows how each of these rules is used. We are given the following class definitions:

```
Pie         SubclassOf:   Apple Cinnamon
Apple       SubclassOf:   Fruit
Cinnamon    SubclassOf:   Spice
Fruit       SubclassOf:   Food
Spice       SubclassOf:   Food
```

The inheritance graph is as follows:



We start with `Pie`, the most specific class in the graph, and traverse the graph to `Apple` and `Fruit`:

```
(Pie Apple Fruit)
```

The next class is `Food`, but adding `Food` would make it precede `Cinnamon` and `Spice`, which are both `Foods`. Skipping this class, we re-traverse the graph, adding `Cinnamon` and `Spice`, and finally `Food`:

```
(Pie Apple Fruit Cinnamon Spice Food)
```

and now we have a complete ordering.

This ordering, where the most specific class is listed first is called `base-flavor-last` order, where the base-flavor is the most general class (class `Food` in our example). This allows a class to override methods described in superclasses. The opposite ordering, `base-flavor-first`, is used in some circumstances to order the execution of methods in least-to-most specific order when combining them; see [Moon85] for details.

Flavor's method for flattening the inheritance graph can be quite complicated and subtle in places, and any method for flattening inheritance graphs, including depth-first search, will introduce "ghost" superclasses; for example, although `Fruit` does not inherit from `Cinnamon`, the ordering is such that `Cinnamon` could end up providing a method that `Fruit` did not implement. These problems often are symptoms of poor structuring; however, it is more desirable to have the system help prevent these occurrences.

Trellis/Owl doesn't use any implicit method inheritance mechanism. Instead, when a class inherits a method definition from more than one superclass, the programmer must explicitly specify which definition is wanted, or write a new definition in the subclass.

5.0 Method Combination in Multiple Inheritance

Method combination is a critical part of the design of a multiple inheritance language. Inheriting state structure in such a language is straightforward: structure is static, and can be accessed in any order by the object's methods. Name conflicts in structure are easily handled; the programmer only names his own fields in his subclass definitions, and access to superclass fields must be done by message passing[15]. Subclasses just merge their additional state with the state of their superclasses. On the other hand, inheritance of methods is a little more difficult. Since methods are executed sequentially, inheriting a method and refining it in a subclass may require a special ordering or control structure to provide the right behavior.

Method combination provides the ability for objects to use specific parts of inherited behavior in combination with new behavior. For example, say `FramedWindow` inherits from the class `Window`. Objects of these classes can display themselves in response to a message `draw`. If the object being drawn is in fact a `FramedWindow`, the `draw` method might invoke the `draw` method for `Window` first, followed by code in `FramedWindow`'s method for drawing the frame.

5.1 Method Combination Issues

There are several issues involved in designing a method combination mechanism. Sometimes the default behavior of a class (the implementation of the method in the superclass) needs to occur as well as some additional behavior, as in the `FramedWindow` example. Instead of rewriting or copying down the code in the superclass' method, a form of method combination can be invoked.

What determines the unit of behavior in method combination? In most object oriented systems, entire methods are taken from the object's class and appropriate superclasses and put together into a *combined method* to be run. It is important that these methods not be accessible individually if they each only define part of an object's behavior for a given message. For example, if you send a message `draw` to a `FramedWindow`, and the superclass `Window` sets up a clipping region before drawing and restores it afterwards, it should not be possible for the programmer to execute the window's `draw` method in isolation, without executing the clipping code.

[15]Note that Smalltalk allows direct access to superclass instance variables, so name conflicts are still a problem.

Another issue is whether behavior defined in a superclass method should be guaranteed to be run when a subclass defines a method of the same name. Typically, this is what is desired; rarely does a subclass method completely replace the code in the superclass, and in most cases it would indicate poor factorization of code. However, one very useful case exists where this is required: that of default behavior. For example, in Smalltalk, all objects inherit a default `print` method which is invoked only if the subclasses do not define their own `print`. In this way, complex objects like `Windows` which are virtually unprintable are printed as "`A Window`". Simpler objects for which printing makes sense define their own print methods (a `Complex` number might print as "`3+4i`"). Because this case is so useful, it is important in a method combination scheme to be able to support this behavior.

5.2 Inner

The original method combination technique was introduced by Simula. A special keyword, `inner`, was used in a superclass method definition to indicate that subclass methods defined for this message should be run at that location in the superclass method. This facility allowed the superclass to run code both before and after code from the subclass was executed, but required the programmer to structure his code top-down so that subclasses would have access to the superclass's method. In most other languages this need not be pre-determined; any superclass methods that are needed can be used at will, and in any combination.

Inner requires that the programmer understand the superclass that he is deriving code from. This can either be considered a feature or a problem, depending on the point of view of the programmer.

Consider the method for `draw` in class `Window`. In window management systems it is often necessary to set up a graphics clipping region before drawing and restore it afterwards. The method for `draw` using `inner` would be written as follows:

```
Window methods:
    draw    [...set up clipping region...
            inner
            ...restore clipping region...]
```

This structuring would allow subclasses to automatically inherit the setting and restoring of the clipping regions; their code would be executed at the location of `inner` in the `draw` method.

Often the code can be merged at compile time, as with Simula, if the method structures are completely determined. Only when the procedures are virtual will there need to be a run-time lookup.

Inner cannot be overridden; when a message is received, code in superclasses is guaranteed to be run when they define the corresponding message. This behavior has advantages as well as disadvantages, as expected: subclass behavior and side-effects will more closely resemble that of their superclasses, since the code will always be run. On the other hand, sometimes subclasses are designed to completely redefine messages inherited from their superclasses, as with default behavior. This is impossible using the standard definition of `inner`.

To solve this problem, `inner` could be defined to return a value. In the previous example, the default `print` method in `Object` could be written as follows:

```
Object methods:
    print
        ["First invoke subclass definition, if any"
        printResult - inner.
        if printResult != nil then return printResult
        else return "A " + self className.]
```

The subclasses would then implement their own `print` method, returning its result. If there was no method defined, `inner` would return `nil`, and the superclass would provide the default printing behavior.

5.3 Super and Self

In single inheritance systems, such as Smalltalk, the special message receiver `super` is provided to allow methods defined in superclasses to be explicitly executed. In effect, a message is sent to the receiver *as if it were an instance of its superclass*, restarting the method lookup one class level higher. Another special receiver `self` is provided as well, to send messages to the current object.

In our example `Window` classes, the `draw` method in `FramedWindow` might invoke `Window`'s `draw` method before drawing the frame:

```
FramedWindow methods:
    draw [super draw.  ...code to draw the frame here...].
```

Being able to send a message to yourself is very convenient, and is used quite frequently in the Smalltalk system. However, what determines who `self` really is when the current method running happens to be in a superclass of the initial receiver? Should the method lookup begin as usual at the class of the receiver, or should it begin in the current method's class? Arguments could be made for both sides: clearly the implementor of the superclass method knows nothing about the subclass's existence, and expects `self` to run methods defined in its own class. On the other hand, if the subclass had refined the message being sent to `self`, it might be an error not to run the refined method. Smalltalk, in fact, begins the lookup in the object's class and not the superclass; some languages provide an additional construct `receiver` to explicitly invoke this behavior.

When a class inherits from more than one superclass, choosing which method to run is a difficult process if there are multiple definitions. This new definition may use the superclass methods by directly naming the superclasses as qualifiers to the method invocation (`Window'draw` instead of `super draw`), as with Trellis/Owl. Often this is sufficient, but there are some cases in which more specific control is needed. In our example class `FramedTitledWindow`, the obvious method for draw would be

```
FramedTitledWindow methods:
    draw [FramedWindow draw.  TitledWindow draw].
```

This is clearly a problem, since both `FramedWindow` and `TitledWindow` inherit from `Window`, whose `draw` routine will be invoked twice. There are several solutions to this problem; most involve taking the `draw` method apart in the superclass and making it into several different methods, which are called separately by the subclass.

5.5 Before and After Demons

Special methods called *demons* are provided in the Flavors system for method combination. `before` and `after` demons, which are run before and after the primary method in the superclass, can be defined in a subclass. Using demons, our `Window` example would look like the following:

```
FramedWindow methods:
    after draw [self drawFrame]

TitledWindow methods:
    after draw [self drawTitle]

FramedTitledWindow methods:
```

no method for draw required; inherited from Framed and Titled windows.

Note that because Flavors supports multiple inheritance, there may be more than one `before` and more than one `after` demon which can be run for a given method. When a method is chosen to be run, all of the applicable `before` demons are run in `base-flavor-first` order (as computed by the flavor ordering procedure), then the primary method is run, then all the `after` demons in `base-flavor-last` order are run. This arrangement ensures that the most general demons wrap around more specific ones, with the primary method (by definition the most specific) in the middle.

This implicit ordering can be problematic. If the order of `drawFrame` and `drawTitle` matter in the combined `FramedTitledWindow` method, it is possible that the order determined by Flavors is incorrect. Ordering typically matters when both methods side-effect shared state in the object itself, or count on the object being in a particular state. The only way to handle this problem is to explicitly specify the order of inheritance of the superclass methods *in this particular method*; there may be no correct class-wide ordering; however, Flavors does not provide this capability.

5.6 Wrappers

Sometimes it is convenient to have code in a subclass method run both before and after code in a superclass method. Flavors provides *wrappers* to allow the programmer to "wrap" code around an existing method. Before and after demons are not sufficient, because the sequential ordering of demons and main method is enforced. Sometimes it is necessary that the main method *not* be run, or to have a more complicated control structure than just a sequential ordering of the demons and primary method.

For example, consider a subclass of `Window` called `BoldWindow`, where every line in the window is drawn twice as wide. `BoldWindow` could be implemented by a wrapper that sets the pen width to two, calls the `Window draw` method, then restores the pen width[16]. Wrappers provide more execution structure than a `super` call in that they are

[16]This could also be done with demons, but there would have to be two separate demons, one before and one

combined in the same way as demons, and multiple wrappers on a method are nested appropriately.

5.7 Programmer-Defined Combination

Even before and after demons and wrappers don't provide enough flexibility to implement the kind of method combination desired in Lisp systems. Flavors allows the programmer to define his own method combination facilities as well, with the full generality of Lisp. Consider the following example, again modified from [Moon85], where the class Army inherits from the class Military:

```
Class: Military
    instanceVars:   reserve-diesel-supply reserve-gas-supply reserve-coal-supply
    messages:       totalFuelSupply
    methods:
    --define totalFuelSupply as the sum of all totalFuelSupply results in all classes
    generic totalFuelSupply: fuelType [methodCombination: sum]
    totalFuelSupply: fuelType
        [fuelType = diesel    ifTrue: [reserve-diesel-supply]
        fuelType = gas        ifTrue: [reserve-gas-supply]
        fuelType = coal       ifTrue: [reserve-coal-supply]
        0]

Class: Army
    subclassOf:  Military
    instanceVars:   aux-diesel-supply, field-diesel-supply, aux-gas-supply,
                    field-gas-supply
    messages:       totalFuelSupply
    methods:
    --Define this subclass's totalFuelSupply, to be added to superclass's result
    totalFuelSupply: fuelType
            [fuelType = gas    ifTrue: [aux-gas-supply + field-gas-supply]
            fuelType = diesel ifTrue: [aux-diesel-supply + field-diesel-supply]
            0]
```

Sending the message totalFuelSupply to an instance of Military will return the appropriate value in the Military structure, while sending it to an instance of Army will add the Army's totalFuelSupply to that of the Military's, returning the sum. The same effect could be achieved with a super call in the Army's totalFuelSupply

after, to accomplish the same thing as a single wrapper.

method, but if `Army` inherited from more than one superclass, a more complicated solution would be necessary.

5.8 A Generalization of Inner: Alternation

The execution of methods with `inner` may be considered to be a sort of alternation between methods: first the superclass method is run up to the `inner` statement, then subclass methods are run to completion, finally returning to the superclass to finish its method execution. In a hierarchical inheritance structure, `inner` acts as a subroutine call to the next method down in the hierarchy.

With multiple inheritance, `inner` may be generalized to be a *changeover point,* (denoted by a * in our examples), where another method is invoked to run until it either terminates or hits a changeover point itself. This allows methods to interleave execution in a powerful and general way. The mechanism is called *alternation* [Thomsen86].

In Thomsen's paper, alternation is part of a multiple inheritance strategy that includes inheritance of data, procedure, and process in a unified framework. The main idea as presented here involves providing a method inheritance hierarchy which is separated from the data inheritance hierarchy, which gives the flexibility of ordering which is often needed in subclass methods.

Alternation requires that methods state explicitly the superclass methods that they refine. The order of methods listed determines the alternation order. For example:

```
Window methods:
   draw inherits: ().
          [A...set up clip, draw self...
          * "Changeover point, just like inner"
          B...restore clip...]

FramedWindow methods:
   draw inherits:  (Window.draw)
          [C...code for drawing the frame...]

TitledWindow methods:
   draw inherits:  (Window.draw)
          [D...code for drawing the title...]
```

Since FramedWindow draw inherits Window's draw method, they alternate in top-down order (Window.draw first, then FramedWindow.draw, etc.). When there is no explicit changeover point, alternation occurs when the currently executing method terminates.

With multiply-inherited operations:

```
FramedTitledWindow methods:
    --Define an ordering for method combination for this method
    draw inherits: (FramedWindow.draw, TitledWindow.draw)
        [...no code required...]
```

In this case, the alternation order would be (Window, FramedWindow, TitledWindow, FramedTitledWindow), the breadth-first search of the method inheritance tree (which can be different than the class inheritance tree). The code will be executed in the topdown order A, C, D, B, with the alternation point providing the point of specialization. No demons or other combination techniques are required, and the desired behavior is accomplished.

Directly naming the methods inherited can bring up some of the same problems as are encountered when directly naming superclasses: the knowledge of the names of the superclasses inherited breaks the encapsulation, and changing a name could require method recoding[17]. Relative naming, such as super, is more flexible, but can allow errors that would be less likely to occur with direct naming.

Inner and alternation both have the problem that it is hard to specify the state a subclass method will encounter when it executes at an alternation or inner point. The superclass maintainer is free to change his code, perhaps invalidating assumptions that the subclass implementer made. However, this is no different from any inheritance situation where the subclass changes or counts on state stored in its superclass variables; the state of the superclass method itself, and the changes in state it has made up to the point of the inner statement, must be considered a part of the object state as well. Supporting behavior such as in the window clipping example using super is no better, because saving and restoring the clipping state would have to be broken out into separate methods, to be called individually by the subclass.

[17]However, one could argue that method recoding should occur, since the superclasses have changed dramatically enough to warrant a re-examination of the subclass code.

6.0 Conclusion

Object oriented languages derive much of their expressibility and power from the ability to inherit state and behavior from superclasses. While inheriting state is straightforward, inheriting behavior requires an external structuring mechanism to sequence the behavior of the superclasses and subclasses in a useful way. With hierarchical inheritance the constructs `inner` and `super` are sufficient for most method combination requirements, although the philosophies of use are different. `Inner` requires that the methods for classes be structured top down, since the programmer must provide the ability explicitly for superclass methods to be refined. `Super` does not require this, since any superclass method can be invoked from a subclass; however, it is not possible for subclass methods to be executed automatically within a single superclass method.

When a language supports multiply-inherited classes, the issues of method dispatch and combination become much more complicated. Quite a few mechanisms have been developed, including extensions of `inner` (alternation) and `super`. Flavors goes even farther, providing demons for incrementally adding behavior before and after a primary superclass method, as well as support for user-defined method combinations.

The alternation technique reflects a different programming philosophy than that of demons and `super`. With alternation, the ordering of code inherited from superclasses can differ from method to method. Also, the unit of combination can be less than an entire method, and a single superclass method can wrap code around subclass methods. This is quite convenient; programmers using languages without alternation would be forced to write two different superclass methods, one to be executed before the subclass method and one after. Unfortunately, these methods are available to other clients as well, and can be executed in isolation, which could have undesirable results.

Any method combination can be used as long as it can express the different kinds of refinement of behavior in a clean, highly encapsulated manner. Ultimately, choosing among competing method combination techniques is a question of programming style. In the Lisp and Smalltalk worlds, the ability to modify existing code as freely as possible is important; method combination techniques which allow as much flexibility as possible are desirable. In languages like Simula, the overall structure of the system is considered to be important, and top-down mechanisms, such as alternation, are preferable.

References

[Agha86]
Agha, Gul: "An Overview of Actor Languages", SIGPLAN Notices V21 #10, October 1986

[Black86]
Black, Andrew, et al.: "Object Structure in the Emerald System", OOPSLA 1986

[Bobrow85]
Bobrow, Daniel, Kahn, Kenneth, et al: "CommonLoops: Merging Common Lisp and Object oriented Programming", OOPSLA 1986

[Dahl66]
Dahl, Ole-Johan, and Nygaard, Kristen: "Simula--an Algol-based Simulation Language", ACM Communications 9:9 {Sept. 1966], 671-678

[Goldberg83]
Goldberg, Adele, and Robson, David: "Smalltalk-80: The Language and its Implementation", Addison-Wesley, 1983

[Ingalls86]
Ingalls, Daniel H. H.: "A Simple Technique for Handling Multiple Polymorphism", OOPSLA 1986

[Kristensen] Kristensen, Bent Bruun; Madsen, Ole Lehrmann; Møller-Pedersen, Birger; and Nygaard, Kristen: "Classification of Actions, or Inheritance also for Methods", (pre-release), Ålborg University Center, Ålborg, Denmark

[Kristensen87]
Kristensen, Bent Bruun; Madsen, Ole Lehrmann; Møller-Pedersen, Birger; and Nygaard, Kristen: "The Beta Programming Language", to appear in *Research Directions in Object Oriented Programming*, edited by B.S. Shriver and P. Wegner, Spring 1987

[Lang86]
Lang, Kevin, and Pearlmutter, Barak: "Oaklisp: An Object Oriented Scheme with First Class Types", OOPSLA, October 1986

[McAllister86]
McAllister, David, and Zabih, Ramin: "Boolean Classes", OOPSLA, October 1986

[Moon85]
Moon, David, and Keene, Sonya: "Flavors: Object Oriented Programming on Symbolics Computers", Common Lisp Conference, December 1985

[Schaffert86]
Schafftert, Craig, et al.: "An Introduction to Trellis/Owl", OOPSLA 86

[Snyder85]
Snyder, Alan: "Object Oriented Programming for Common Lisp", Hewlett-Packard Company, ATC-85-1, February 1985

[Snyder86]
Snyder, Alan: "Encapsulation and Inheritance in Object Oriented Programming Languages", OOPSLA 1986

[Stroustrup86]
Stroustrup, Bjarne: "The C++ Programming Language", Addison-Wesley, 1986

[Thomsen86]
Thomsen, Kristine: "Multiple Inheritance, a Structuring Mechanism for Data, Processes, and Procedures", Department of Computer Science, Århus University, April 1986