Flavors

A non-hierarchical approach to object-oriented programming

by

Howard I. Cannon

Object-oriented programming systems have typically been organized around hierarchical inheritance. Unfortunately, this organization restricts the usefulness of the object-oriented programming paradigm. This paper presents a non-hierarchically organized object-oriented system, an implementation of which has been in practical use on the MIT Lisp Machine since late 1979.

Copyright © 1979, 1992, 2003 by Howard I. Cannon.
All rights reserved.

Object oriented programming systems and Flavors

Introduction and terminology

Object oriented programming systems are becoming more widely used. This powerful technique, pioneered by Simula [ref Simula] and Smalltalk [ref Smalltalk] [ref Hewitt?] [CLU?], provides a way of implementing highly modular systems and generic algorithms. For the purposes of this paper, an *object* consists of some *local state* and some *behavior*. An object is asked to perform an operation (computation) by specifying the generic name of the operation (*sending a message*) and by specifying arguments to that operation (a value may be returned). Associated with each object is a means by which a piece of code (*method*)can be found from the name of an operation. When a message is sent to an object, the object finds the appropriate method and runs it, giving it the supplied arguments. In the Lisp Machine system, a *message pass* is considered to be a function call. The arguments are evaluated in standard fashion, the message is sent, and values are returned.

A method is just a standard Lisp function. In order to perform the designated operation, the method accesses and possibly modifies the local state, calls other Lisp functions, or sends messages to objects. *Self* is a special name that designates the object that was sent the message that caused the currently executing method to be called. Using this name, an object may send a message to itself. Certain methods also return meaningful values.

This paradigm permits the implementation of *generic algorithms*. A set of messages (sometimes called a *protocol* [ref Xerox?]) is defined, that specifies what the external behavior must be if an object is to implement the protocol. The protocol does not define how the behavior is to be implemented by the object. Thus, it is possible to have complicated operations that work on any object that obeys a particular protocol, no matter how that object actually works. For example, a commonly defined protocol is the input-output or *stream* protocol. This protocol specifies that the **output-character** message must somehow output the character to wherever the stream is connected. However, the connection may be to a terminal, to a file, or even to a string in memory. To a generic algorithm that prints numbers, where the character goes, or how it is put there, is not important. The stream protocol would completely specify all of the messages necessary to perform input/output.

An object is created by *instantiating* a description of that object. An object is also called an *instance*, and the elements of local state are called *instance variables*. Thus, creating a file input/output stream consists of specifying a set of local state and methods, and creating an object that has that local state, and looks up messages in that set of methods. In the "0'th order" system, for each new kind of object, there would be a complete definition of every method, and complete enumeration of the local state. Though this is an adequate scheme for some purposes [did Simula have essentially such a scheme?], the introduction of additional defining mechanisms can greatly increase the functionality of an object oriented programming system.

Hierarchies

In Smalltalk, the description from which an object is instantiated is called a *class*. But instead of specifying all methods and all instance variables, a class also *inherits* these from its parent class (*superclass*). Thus, it is possible to take an already existing class and modify it by adding additional local state, by adding additional methods, or by *shadowing* (replacing) a previously existing method. When a class is instantiated, the instance variables specified by the class and all of its superclasses are collected and form the template for the instance's local state. Since each class has an association list of message name and method (formed as the methods are defined), when a message is sent to the object the association lists are inspected starting from the object's class and continuing through all of its superclasses (in order) until a method is found.

A domain in which object oriented programming has been applied is the domain of bit-mapped graphics.[1] Consider the case of a software facility to manage a bit-mapped display. The facility provides rectangular areas (called *windows*) that can be used by programs to perform various sorts of output operations on the display. The windows provide a means of allowing multiple programs to use the display at the same time, without interfering with each other. More importantly, they also provide a place to isolate unique display oriented features that a particular program or set of programs might need. Of course, no matter what unique features the window might have, it still needs to respond to a simple protocol that includes messages to clear it, change its size, move it around, and so on. Each window also has local state that includes its position and its size. Object oriented programming is an excellent implementation strategy for such a facility.

Creating the basic definition of a window might proceed as follows:

```
(defclass WINDOW
   OBJECT
   (X-POSITION Y-POSITION WIDTH HEIGHT))
```

This **defclass** defines the class **WINDOW**, which has a superclass of **OBJECT** (the class that is the superclass of all classes, and handles various housekeeping functions that all objects want to have) and four instance variables.

```
(defmethod (WINDOW :SET-POSITION) (new-x new-y)
   (move-screen-area X-POSITION Y-POSITION new-x new-y WIDTH HEIGHT)
   (setq X-POSITION new-x
         Y-POSITION new-y))

(defmethod (WINDOW :CLEAR) ()
   (erase-screen-area-primitive X-POSITION Y-POSITION WIDTH HEIGHT))

(defmethod (WINDOW :REFRESH) ()
   (send SELF ':CLEAR))
```

These **defmethods** define some methods to handle particular messages (message names are preceded by a colon, by convention). First, a method for the **:CLEAR** message, which takes no arguments, and simply calls a system primitive to erase a region of the screen, passing it the values of the four instance variables that define the position and size of the window. Then, a method for the message **:SET-POSITION**, which changes the position of a window. This

[1] In fact, graphics systems have been prime motivators of the development of object oriented programming on the Lisp Machine.

method first moves the window's bits from the old place to the new place, and then changes the window's own idea of where it is, by modifying the instance variables.[2] The third method for the **:REFRESH** message, which is defined to clean up the window's representation on the screen, simply sends the **:CLEAR** message back to **SELF**. This has the effect of simply clearing the window's region of the screen when the window receives the **:REFRESH** message. Making **:REFRESH** a separate message provides for more complicated behavior which will be motivated later.

Usually windows look better if they have a black border around them. It is not necessary to make all windows have this border, as a subclass of window can be defined that inherits from **WINDOW** and adds the border behavior.

```
(defclass WINDOW-WITH-BORDER
   WINDOW
   (BORDER-WIDTH))

(defmethod (WINDOW-WITH-BORDER :DRAW-BORDERS) ()
   ;; Draw four lines from <starting point> to <ending point> in
   ;; <drawing mode> with <width>
   (draw-line-primitive X-POSITION Y-POSITION
                        X-POSITION (+ Y-POSITION HEIGHT -1)
                        ior-drawing-mode BORDER-WIDTH)
   (draw-line-primitive X-POSITION Y-POSITION
                        (+ X-POSITION WIDTH -1) Y-POSITION
                        ior-drawing-mode BORDER-WIDTH)
   (draw-line-primitive X-POSITION (+ Y-POSITION HEIGHT -1)
                        (+ X-POSITION WIDTH -1)
                        (+ Y-POSITION HEIGHT -1)
                        ior-drawing-mode BORDER-WIDTH)
   (draw-line-primitive (+ X-POSITION WIDTH -1) Y-POSITION
                        (+ X-POSITION WIDTH -1)
                        (+ Y-POSITION HEIGHT -1)
                        ior-drawing-mode BORDER-WIDTH))

(defmethod (WINDOW-WITH-BORDER :REFRESH) ()
   (send SELF ':CLEAR)
   (send SELF ':DRAW-BORDERS))
```

A new class called **WINDOW-WITH-BORDER** is defined. This class has **WINDOW** as its superclass, and adds the instance variable **BORDER-WIDTH**. Two methods are added: the **:DRAW-BORDERS** method is called to draw the black borders around the rectangular area specified by the four instance variables defined by **WINDOW** (the details of this method are omitted), and the **:REFRESH** method is *redefined* to first clear the window and then draw the borders.[3]

Sometimes windows want to have labels that identify the window. A **WINDOW-WITH-LABEL** class might be defined as follows:

---

[2] In a real window system, this method would also have to account for windows that it might overlap.

[3] This is why the **:REFRESH** method was kept separate.

```
(defclass WINDOW-WITH-LABEL
   WINDOW
   (LABEL LABEL-X-OFFSET LABEL-Y-OFFSET))

(defmethod (WINDOW-WITH-LABEL :DRAW-LABEL) ()
   (draw-string-primitive LABEL
                          (+ X-POSITION LABEL-X-OFFSET)
                          (+ Y-POSITION LABEL-Y-OFFSET)))

(defmethod (WINDOW-WITH-LABEL :REFRESH) ()
   (send SELF ':CLEAR)
   (send SELF ':DRAW-LABEL))
```

Notice that this class is very similar to the **WINDOW-WITH-BORDERS**. It defines the label drawing method **:DRAW-LABEL**, and also redefines **:REFRESH** so that cleaning up the representation of the window causes the label to be redrawn.

Some windows want to have both a border and a label. Given the paradigm currently outlined, there are two ways of defining such a window: define the class **WINDOW-WITH-BORDER-AND-LABEL**, which has **WINDOW-WITH-BORDER** as its immediate superclass, and includes methods for labels; define **WINDOW-WITH-LABEL-AND-BORDER**, which has **WINDOW-WITH-LABEL** as its immediate superclass, and includes methods for borders. Both of these ways involve copying of code. In the former case, the methods for labels must be copied. In the latter case, the methods for borders must be copied. This copying could be extensive. Since copying in general delocalizes information, this copying works against the goal of modularity.

Multiple features

The problem raised here is a general one: the single-superclass scheme cannot handle *orthogonal attributes* in a modular fashion. In other words, where there are several features (e.g. borders and labels) that want to get combined in a pick-and-choose fashion, the single-superclass scheme as presented becomes hard to use. However, there is a scheme that still preserves the one superclass nature of the system.

Define **FRAME** which is a class that has as its local state a window and a list of other objects. A window would then usually be used only as part of a frame. A class would be defined for borders, and one for labels. Then to make a window with a label, one would make a frame with a window of the appropriate size, and would include an object made from class **LABEL**. With this scheme it is clear how to make all possible combinations without getting into both combinatorial explosion of names, and the need to duplicate methods. By defining **FRAME** appropriately, it is possible to get a **:REFRESH** sent to the frame *forwarded* to each object in turn (e.g. the window, the label). The **LABEL** and **FRAME** classes might be defined as follows:

```
(defclass LABEL
   OBJECT
   (LABEL))

(defmethod (LABEL :DRAW-LABEL) ()
   ;; The details of label drawing go here   )

(defmethod (LABEL :REFRESH) ()
   (send SELF ':DRAW-LABEL))
```

```
(defclass FRAME
   OBJECT
   (WINDOW FEATURES))

(defmethod (FRAME :REFRESH) ()
   (send WINDOW ':REFRESH)
   (dolist (feature FEATURES)
     (send feature ':REFRESH)))
```

There are some obvious problems with this scheme.

Firstly, it's not clear which value gets returned. Though none of the sample methods so far defined returns meaningful values, in an actual system there would be messages to get information such as the size of the window, the label, and so on. So, a frame might be asked for the text of its label. This might be an error if the frame had no label, the label object might want to handle the message, or perhaps there would be a more complicated object like **SPECIAL-BORDER-WITH-MAGIC-LABEL** that wanted to handle the message. Either the frame would have to have a method for every message that would explicitly forward the message (time consuming, and non-modular), or there would have to be some mechanism whereby a message that was not explicitly handled by the frame would get automatically forwarded to the appropriate object. If two different objects handled the same message, then the frame would have to decide whether it was correct to send the message to both objects, or only to one of them. Also, which value was to be returned would have to be specified. For every message, information of this sort is necessary. Of course, an object inside the frame may be some sort of frame-like thing itself, and this information would need to be specified at that level as well.

Secondly, the definition of self becomes more complicated. When a method wanted to send a message to the *current object*, it's no longer obvious just what the current object is! In the case of labels, the label could mean just the label object (in the case of **:DRAW-LABEL**), it could mean the frame (in the case of **:REFRESH**, if it was necessary for the label to ever send this message), or it could even mean another specific object that it knows the frame must have (the window, in the case of accessing the size and position). The method trying to send the message should not have to know this kind information, as it presents a potentially serious modularity problem.

Lastly, it is useful for things like labels to be able to access local state in other objects directly. A label knows it must be included with a window somehow, and it's convenient for the label to rely on the position and size of the window being directly accessible. The separate object scheme makes this fairly difficult. This scheme also makes it hard to replace methods of the window that might need to change when a label is included.

Multiple Superclasses

In order to overcome some of these difficulties, a scheme called *multiple superclasses* was devised. This scheme permits, at class definition time, the specification of many superclasses from which the new class is to inherit. So, instead of a strict hierarchy, more of a lattice structure is formed. In this scheme, some of the problems of the multiple objects scheme go away. The definition of self problem no longer exists, since there is only one object. All aspects of the object can share the local state. Again, separate classes can be defined (such as **WINDOW** and **LABEL**), and then combined to form composite classes (such as **WINDOW-WITH-BORDER**), so copying is not necessary.

For example (omitting most methods):

```
(defclass WINDOW
   (OBJECT)
   (X-POSITION Y-POSITION WIDTH HEIGHT))

(defmethod (WINDOW :REFRESH)
   (send SELF ':CLEAR))

(defclass BORDER
   ()
   (BORDER-WIDTH))

(defmethod (BORDER :REFRESH) ()
   (send SELF ':CLEAR)
   (send SELF ':DRAW-BORDER))

(defclass LABEL
   ()
   (LABEL))

(defmethod (LABEL :REFRESH) ()
   (send SELF ':CLEAR)
   (send SELF ':DRAW-LABEL))

(defclass WINDOW-WITH-BORDER
   (BORDER WINDOW)
   ())

(defclass WINDOW-WITH-LABEL-AND-BORDER
   (LABEL BORDER WINDOW))

(defmethod (WINDOW-WITH-LABEL-AND-BORDER :REFRESH) ()
   (send SELF ':CLEAR)
   (send SELF ':DRAW-BORDER)
   (send SELF ':DRAW-LABEL))
```

defclass has been modified to allow a list of classes from which to inherit. **WINDOW-WITH-LABEL** could be easily defined.

The major problem with the multiple superclass scheme is that it doesn't resolve the issue of what to do if more than one superclass wishes to handle the same message. In some cases the answer is less obvious than in the multiple object scheme. In the case of **:REFRESH**, for example, each class wants to contribute some piece of the final behavior, but no such mechanism exists. The simple ways of resolving this problem (calling all of the methods, calling the last one defined (based on some metric)) don't work. In the preceding example, this problem was avoided by replacing the known bad **:REFRESH** method in **WINDOW-WITH-LABEL-AND-BORDER**.

To restate the fundamental problem: there are several separate (orthogonal) *attributes* that an object wants to have; various *facets* of behavior (features) that want to be independently specified for an object. For example, a window has a certain behavior as a rectangular area on a bit-mapped display. It also has its behavior as a labeled thing, and as a bordered thing. Each of these three behaviors is different, want to be specified independently for each object, and is *essentially orthogonal* to the others. It is this "*essentially*" that causes the trouble.

It is very easy to combine completely non-interacting behaviors. Each would have its own set of messages, its own instance variables, and would never need to know about other objects with which they would be combined. Either the multiple object or simple multiple superclass scheme could handle this perfectly. The problem arises when it is necessary to have *modular* interactions between the orthogonal issues. Though the label does not interact **strongly** with either the window or the border, it does have some minor interactions. For example, it wants to get redrawn when the window gets refreshed. Handling these sorts of interactions is the *Flavor system*'s main goal.

Flavors

A *flavor* is something that defines some *instance variables*, some *methods*, and specifies other flavors from which it inherits the same. A flavor is the Flavor system's analog of a class. In the Flavor system, the actual method that gets run when a message is sent to an object consists of some combination of the methods specified by the *component flavors* (the flavor of the object plus all the flavors that are inherited by that flavor, recursively) of that object. This is called a *combined method*. In the case of the window system, for example, the **:REFRESH** combined method would be built up piece by piece from the various component flavors. The way in which the methods are combined is controlled both by keywords on the individual methods, and by declarations in the component flavors.

When a flavor is instantiated, all of the component flavors are inspected in a well-defined order: first, the flavor itself, then all the flavors it inherits from, in left-to-right order, recursively (depth first). The *union* of the instance variables specified by each flavor is computed and serves as the template for the local state of the instance. An association list of message names and combined methods must also be generated.

In the class system(s), which method(s) to run is largely determined at run time. This is possible as only local knowledge is necessary to make the decision.[4] This is not true of Flavors: in general, determining the methods to be run requires inspecting all of the component flavors and generating a combined method. It is from this *use of global knowledge* that Flavors gain the ability to modularly integrate essentially orthogonal issues. At instantiation time, as the component flavors are inspected, combined methods are generated and an association list of message names and combined methods is constructed. In essence, the lattice structure is *flattened* into a linear one. This is important it makes the use of global knowledge practical, since in certain cases, the combined methods are not trivial, and could not easily be generated dynamically.[5]

How methods are combined is determined on a per message basis. A *method combination* can be viewed as a template for converting a list of methods into a piece of code that calls the appropriate methods in the appropriate order and returns the appropriate values. This code is the combined method.[6] The default, and most used, method combination is called **daemon combination**. @label(DaemonCombination) There are three types of methods for this

[4] This assertion is strongly related to the problem!

[5] It is not important, however, that this flattening is done at instantiation time, as opposed to incrementally.

[6] In the Lisp Machine implementation of Flavors, method combinations are like MacLisp macros [ref Moonual?] in that they are pieces of code that return the code for the combined method.

combination: *primary*, *before*, and *after*. Untyped methods default to primary type. The combined method first calls all of the before methods in order and throws away the value they return, then the *first* primary method is called and its value is saved, then the after methods are called in *reverse order*, and then the combined method returns the value returned by the primary method.

Using this method combination type, here is how a piece of the window system might be defined (some methods omitted for clarity). Note that method types are preceded by colons like message names, just by convention:

```
(defflavor WINDOW
   (OBJECT)
   (X-POSITION Y-POSITION WIDTH HEIGHT))

(defmethod (WINDOW :REFRESH)
   (send SELF ':CLEAR))

(defflavor BORDER
   ()
   (BORDER-WIDTH))

(defmethod (BORDER :AFTER :REFRESH) ()
   (send SELF ':DRAW-BORDER))

(defflavor LABEL
   ()
   (LABEL))

(defmethod (LABEL :AFTER :REFRESH) ()
   (send SELF ':DRAW-LABEL))

 (defflavor WINDOW-WITH-LABEL-AND-BORDER
   (LABEL BORDER WINDOW)
   ())
```

The overall structure of the flavors in this example is the same as the structure of the classes in the multiple superclass scheme. If the daemon combination type with no before or after methods is used, then flavors act exactly like a simple multiple superclass scheme. If there is only one flavor from which to inherit, and only primary methods are used, then the flavor scheme acts like the class scheme. However, the use of after methods allows the **:REFRESH** message to be handled modularly. Each feature of the window (**WINDOW** flavor, **BORDER** flavor, **LABEL** flavor) is able to contribute just the code needed to implement that feature's aspect of the total method. Then, all of the aspects are integrated when the combined method is generated.

It has been pointed out [cite Drescher somehow] that the daemon combination type can be implemented in a multiple superclass system with the addition of a primitive □ the ability for a method to continue sending a message as if the method was not there. With this operation, and a naming convention for methods, many of the features of the flavor system can be simulated [example in an appendix?]. However, since the view of inspecting all methods and generating a combined method is not taken by this approach, certain facilities are not available. However, this approach can be useful as an aid in understanding the Flavor system.

The Flavor system can be viewed as combining specific implementations of different protocols under control of *meta-protocols*. A meta-protocol specifies, on a per-message basis, details for

combining methods for that message. It also controls how instance variables are merged. For example, there is the window protocol and the label protocol: the **WINDOW** flavor is a particular implementation of the former, and the **LABEL** flavor is a particular implementation of the latter. The meta-protocol for combining windows with anything asserts that the **:REFRESH** message is combined with daemon combination, that the window always provides the primary method, and that all output is done either by the primary method or by after methods, or similar mechanisms.

Conventions

From one point of view, object oriented programming is a set of conventions that helps the programmer organize his program. When the conventions are supported by a set of tools that make them easier to follow, then an object oriented programming *system* is born. It is neither feasible nor desirable to have the system enforce all of the conventions, however. Since the Flavor system provides more flexibility than other object oriented programming systems, *programmer enforced conventions* become correspondingly more important. Therefore, *the Flavor system is as much conventions as it is code*.

Flavors help obtain modularity, but they can't force it. It is important for the programmer to carefully consider the design of his flavor structure beforehand. In general, message passing systems are more forgiving in the face of mistakes in modularity than other systems. The Flavor system can be particularly tolerant of such errors, *if the conventions are followed*.

The Shape of Structures

Shallow, wider structures are more desirable than deep, narrow structures. When structures are built by inheriting from flavors and then adding (replacing) methods, relationships between the new methods and the old are statically created – the methods cannot be used apart from the component flavors. The preferred alternative is to create new flavors to hold the new methods. Thus, it is possible to decide to include the new methods with different combinations of flavors. Flavors used in this way are like macros: they provide a means to name a group of methods and then use that name in place of the methods themselves. The Flavor system allows and encourages *delaying decisions about structure* until the latest possible moment.

For example, one possible window system organization would be to have these flavors: **WINDOW**, which would be the fundamental window, upon which most other windows are built; **WINDOW-WITH-LABEL**, which has **WINDOW** as a component flavor, and adds a label; and **BORDER**, which has no component flavors, but implements borders. This would permit construction of all of the types of windows previously discussed, as the border could be included or not in either **WINDOW** or **WINDOW-WITH-LABEL**. An alternate, and preferred, way is to have a separate flavor **LABEL**, with **WINDOW-WITH-LABEL** formed out of it and **WINDOW**. Then, if a new flavor **EXPERIMENTAL-WINDOW** was implemented, and **EXPERIMENTAL-WINDOW-WITH-LABEL** could be made without having to duplicate the label code. Admittedly, given the simplicity of the sample window system, this a contrived example. In a complicated system built out of flavors, things like **EXPERIMENTAL-WINDOW** are frequently defined.

A very malleable system can be built by never defining methods on any flavors that have component flavors. This total lack of structure guarantees that the groups of messages (i.e.

flavors), can always be used in a different manner. Thus, the world is divided up into two types of flavors: those which define methods, and those which build combine the others. An even more malleable system can be constructed by defining only one method per flavor. However, going to this extreme leads to a very large number of names that need to be remembered, making the system harder to comprehend and use effectively, thereby counteracting some of the features that object oriented programming offers in the first place. A compromise is often necessary: some flexibility in the final structure must be sacrificed to reduce the *cognitive space* of the program.

In this context, flavors can be classified in several ways. A *base flavor* serves as a foundation for building a family of flavors. It defines instance variables, sets up defaults, and is often not instantiable. A *mixin flavor* is one that implements a particular feature, which may or may not be included. Mixins are almost never instantiable, often have the word mixin in their name, and often define a handful of instance variables. Instantiable flavors are built out of a base flavor and several mixins. In the preceding examples, **WINDOW** is both a base flavor and instantiable, **BORDER** (which should have been called **BORDER-MIXIN**) is a mixin, and **WINDOW-WITH-LABEL** is an instantiable flavor.

When an essentially orthogonal feature is to be implemented, a new mixin is defined, with no component flavors. However, when an existing mixin needs to be slightly modified, and the modifications are such that they would make little or no sense apart from the original mixin, then a new mixin, with the old mixin as a component flavor, is defined. Direct use or modification of the instance variables defined by the original mixin would be one good reason for not making a standalone flavor. For example, if it was necessary for the label to be drawn in the upper right-hand corner of the window, instead of the lower left, and the code looked directly at the **label** instance variable, then including **LABEL-MIXIN** directly would be reasonable. If, however, the method sent the **:LABEL** message to access the text, then it would be reasonable to have the method defined in its own standalone flavor.

A flavor that defines an instance variable should be responsible for altering its value. This convention is necessary to prevent timing and interlocking problems. Variables that are used internally by a flavor can be directly modified by methods of that flavor, as these variables are not supposed to be used by other flavors. However, if a variable needs to be altered from the *outside*, by either or flavors or by the user, or if its change in state needs to be *observed* by other flavors, then a method should be defined to modify the variable. In order to allow other flavors to define methods that observe the alteration, the message to set variable *AnyVariable* is called **:SET-***AnyVariable*. The primary method for this message should be defined by the flavor that defines the instance variable.

Ordering Dependencies and Duplicate Elimination

An important premise of the Flavor system is that methods are combined in a well-defined order.[7] The major benefit gained by the definition of the order is the ability to predict in what order methods will be run. The ordering works out quite nicely for the daemon combination type – it turns out to be right almost all of the time! Adding a before type and after type pair of

[7] Though making the order well-defined seems obvious, it was not when the Flavor system was first conceived.

methods is like bracketing all of the ones further to the right.[8] Ordering can also be used explicitly to implement behavior that is otherwise very hard to achieve.

Consider labels and borders: the label could be inside the border, or the label could be outside the border. One way of implementing this would be to have some sort of mixin that told a label to go on either the outside or the inside of the border. This mixin would have to look at the size of the border, or and possibly modify the border's position. There could also be a mixin for the border that allowed it to cooperate with the label. However, this scheme does not extend very well to more than two things (there might be other regions near the edges of the window, all competing for space). Another way of implementing this would be to define a flavor called **MARGINS-MIXIN**. This flavor would have to be mixed into all windows that had things in their margins.

The **MARGINS-MIXIN** flavor would define four instance variables: one for the amount of space in use near each edge of the window. These instance variables would be reset by a before type method on the **:REFRESH** message, and modified as each flavor that put something in the margins refreshed what it was responsible for. The **:REFRESH** methods might look like:

```
(defmethod (MARGINS-MIXIN :BEFORE :REFRESH) ()
   (setq LEFT-MARGIN-USED 0
         TOP-MARGIN-USED 0
         RIGHT-MARGIN-USED 0
         BOTTOM-MARGIN-USED 0))

(defmethod (LABEL-MIXIN :AFTER :REFRESH) ()
   (setq BOTTOM-MARGIN-USED (+ BOTTOM-MARGIN-USED LABEL-HEIGHT)
   ;; It is not necessary to adjust the left margin, since we aren't
   ;; really using any of it.
   (setq LABEL-X LEFT-MARGIN-USED
         LABEL-Y (- (send SELF ':HEIGHT) BOTTOM-MARGIN-USED))
   (send SELF ':DRAW-LABEL))

(defmethod (BORDERS-MIXIN :AFTER :REFRESH) ()
   (setq BORDER-LEFT LEFT-MARGIN-USED
         BORDER-TOP TOP-MARGIN-USED
         BORDER-RIGHT RIGHT-MARGIN-USED
         BORDER-BOTTOM BOTTOM-MARGIN-USED)
   (setq LEFT-MARGIN-USED (+ LEFT-MARGIN-USED BORDER-WIDTH)
         TOP-MARGIN-USED (+ TOP-MARGIN-USED BORDER-WIDTH)
         RIGHT-MARGIN-USED (+ RIGHT-MARGIN-USED BORDER-WIDTH)
         BOTTOM-MARGIN-USED (+ BOTTOM-MARGIN-USED BORDER-WIDTH))
   (send SELF ':DRAW-BORDERS))
```

Thus, by combining flavors such that **BORDER-MIXIN** is before (more to the left than) **LABEL-MIXIN**, the borders will show on the outside of the window, with the label directly inside the borders. If the flavors are combined in the opposite order, then the label will be outside of the borders. It is also trivial to add other flavors that use the margins, and they will interact nicely with borders and labels. This example illustrates not only the explicit use of ordering, but also the importance of careful design and forethought of the flavor structure.

---

[8] Historical note: the reason that this is called daemon combination is that originally the before and after type methods were going to be just like conventional daemons – the order in which they were run would be left explicitly undefined.

When building complicated flavor structures, it is often the case that several flavors will be combined that share component flavors at some level. The straightforward behavior would duplicate methods with types like before and after. This is, in most cases, undesirable. However, the flavor system, from its global vantage point, eliminates duplicate flavors: it includes methods from a flavor the first time it comes across the flavor in depth-first order, and ignores the flavor if it arises again. This falls under the category of a system enforced convention, as the duplicate elimination does not in any way affect the *correctness* of the system – not performing the elimination would leave a perfectly workable system.

Duplicate elimination is also useful as a means of allowing simple reordering of flavors. If a complicated flavor exists, made out of many component flavors, then it is possible to move a component flavor or two to the front simply by repeating it in the definition of a flavor that has the complicated flavor as a component. For example:

```
(defflavor MY-ULTRA-HAIRY-WINDOW
   ()
   (BORDER-MIXIN LABEL-MIXIN MY-ULTRA-HAIRY-WINDOW-INTERNALS))

(defflavor MY-ULTRA-HAIRY-WINDOW-WITH-LABEL-ON-THE-OUTSIDE
   ()
   (LABEL-MIXIN MY-ULTRA-HAIRY-WINDOW))
```

MY-ULTRA-HAIRY-WINDOW-WITH-LABEL-ON-THE-OUTSIDE is the same as **MY-ULTRA-HAIRY-WINDOW**, except that it has the label on the outside.

Initialization

@label(Initialization)

In the Flavor system, initializing a new instance calls for careful consideration. Initialization includes setting up initial values of the local state variables and modifying global data bases. Initialization is an action that requires contributions from almost every feature. The standard method combination mechanisms are useful in solving this problem. Daemon combination, along with a simple set of conventions and system support, can be used to build a modular and powerful initialization paradigm.

Initial values of the instance variables can be supplied when a flavor is defined. This feature falls under the heading of system supported conventions – there needs to be support in the flavor defining form. However, it is not logically necessary to supply this feature; it is only a programming convenience. An initial value would be used where a variable needs to start off at some constant value in each instance. For example, if a variable was being used to count the number of times some message was sent:

```
(defflavor SOME-FLAVOR
   ((COUNT-OF-A-MESSAGE 0))
   ())

(defmethod (SOME-FLAVOR :BEFORE :A) ()
   (setq COUNT-OF-A-MESSAGE (1+ COUNT-OF-A-MESSAGE)))
```

A **:INIT** message is defined. This message is sent by the system to the new instance when a flavor is instantiated.[9] Methods for this message are combined with the daemon combination type. The primary method is supplied by the base flavor. In the case of a window system, the lowest level flavor would be the simplest kind of window, from which all other windows are built (e.g. flavor **WINDOW**). All other flavors provide before and after methods which perform their particular initializations.

As with alteration, a flavor is responsible for initializing the instance variables that it defines. In what type of method the initialization is done depends upon several factors including whether or not it depends upon the initial values of other variables, and whether or not its value is depended upon by others. There is no clear guideline. If a particular set of flavors needs to be combined, where and when the variables are initialized issues that need to be considered when designing the flavors.

Initialization is parameterized by the use of an *initialization attribute list*. This is a list of attribute-value pairs that is passed as an argument to the **:INIT** methods. Its initial contents is supplied by the program, unsupplied attributes may be defaulted from declarations[10] and it may be modified by the **:INIT** methods. A declaration can also be used to make an instance variable *initable*, so that the system automatically inspects the attribute list and sets the variable if there is an attribute with that variable's name. The attribute list is a means via which, in a modular fashion, a user can communicate information to the initialization code, and via which flavors can pass information among themselves during initialization.

Consider the following partial definitions:

```
(defflavor LABEL-MIXIN
   ((LABEL "I am a Label"))
   ())

(defmethod (LABEL-MIXIN :AFTER :INIT) (attribute-list)
   (if (attribute-present-p attribute-list "LABEL")
       (setq LABEL (attribute-extract attribute-list "LABEL"))))

(defflavor WINDOW-WITH-SPECIAL-LABEL
   ()
   (LABEL-MIXIN WINDOW))

(defmethod (WINDOW-WITH-SPECIAL-LABEL :BEFORE :INIT) (attribute-list)
@label(DefaultInit)
   (if (not (attribute-present-p attribute-list "LABEL"))
       (attribute-add attribute-list "LABEL" "Special Label")))
```

The **WINDOW-WITH-SPECIAL-LABEL** flavor causes the window to have an initialization attribute specifying the label if the user (or previous flavor, if there was one) didn't specify the attribute. In the absence of the attribute, **LABEL-MIXIN** defaults the label, otherwise it uses the supplied value. Since it checks in an after type method, any flavor may modify the label attribute, and it will be noticed by **LABEL-MIXIN**. This is a trivial example of the use of the attribute list.

---

[9] In the Lisp Machine implementation, the **:INIT** message is sent by default, but can be overridden.

[10] See section @ref(Declarations), page @pageref(Declarations)

Declarations

@label(Declarations)

Declarations are associated with flavor definitions. They can be used to extend the system in order to support recommended conventions, and they can be used to add implementation specific features. In the former category, there are several declarations that deal with instance variables. An instance variable can be declared *gettable*, *settable*, and/or *initable*. *Gettable* means that a method is automatically generated to handle a message with the name of the instance variable. The method returns the value of the instance variable. If the variable *AnyVariable* is declared gettable, then a method for the message **:***AnyVariable* is generated. *Settable* means that a method is automatically generated that takes one argument and sets the instance variable to the value of the argument. If the variable was called *AnyVariable*, then the method would be for the message **:SET-***AnyVariable*.[11] *Initable* means that the instance variable is set from an initialization attribute with the name of the instance variable.[12]

For example, the following flavor definition:

```
(defflavor A-PARTICULAR-FLAVOR
   (INSTANCE-VAR)
   ()
   (gettable INSTANCE-VAR)
   (settable INSTANCE-VAR)
   (initable INSTANCE-VAR))
```

would generate (essentially) these methods:

```
(defmethod (A-PARTICULAR-FLAVOR :INSTANCE-VAR) ()
   INSTANCE-VAR)

(defmethod (A-PARTICULAR-FLAVOR :SET-INSTANCE-VAR) (new-value)
   (setq INSTANCE-VAR new-value))

(defmethod (A-PARTICULAR-FLAVOR :BEFORE :INIT) (attribute-list)
   (if (attribute-present-p attribute-list "INSTANCE-VAR")
       (setq INSTANCE-VAR
             (attribute-extract attribute-list "INSTANCE-VAR"))))
```

The *default initialization attribute* declaration is used to specify default attribute value pairs for inclusion on the attribute list. As discussed in the section on initialization,[13] this provides a modular way for flavors to interact during initialization. The before type **:INIT** method of the **WINDOW-WITH-SPECIAL-LABEL** flavor on page @pageref(DefaultInit) demonstrates the effect of this attribute. In fact, it is not logically necessary for the system to write any code, as the function that instantiates the flavor can perform this operation on all flavors before the **:INIT** message is sent.

Implementation dependent declarations provide a means whereby the implementer can add extensions that do not logically affect the operation of the system. It is instructive to look at the Lisp Machine implementation, in which there are several of these declarations. They are used to

---

[11] This follows the convention mentioned earlier.

[12] See section @ref(Initialization), page @pageref(Initialization)

[13] See section @ref(Initialization), page @pageref(Initialization)

give the system hints about frequency of method usage, that the system uses to choose an ordering of method lookup tables. They are also used to control representation of instance variables in the instance, so that instance variables may sometimes be accessed from outside the environment of the instance without sending a message.

Required methods, instance variables, flavors; Included flavors

The Flavor system's generality can make it hard to debug certain sorts of errors, especially errors of omission. These occur when flavor **X** refers to methods or instance variables not supplied directly by it. For example, a flavor that positions labels needs to look at the label and at parameters of the window. However, it is not necessarily tied down to one particular implementation of labels, or one particular type of window. It is important that the Flavor system provide a mechanism where these dependencies can be made explicit.

In the case of methods, a declaration is provided that says that a method to handle a particular message is required. This means that if flavor **A-FLAVOR-THAT-USES-MESSAGE-WILLY** has **:WILLY** as a *required method*, any instantiable flavor *X*, that has flavor **A-FLAVOR-THAT-USES-MESSAGE-WILLY** as a component flavor, must define a method to handle the **:WILLY** message. If after instantiating flavor *X*, the Flavor system notices that the message is not handled, then the user is informed. Since this check is only made on flavors that are actually instantiated, no constraints are placed on the form of the intermediate flavor structure.

In the case of instance variables, the problem is more severe. Not only is it useful to assert that some flavor expects an instance variable to be present in an instance, but in order to compile code for that flavor the compiler must realize that an apparently free reference to that variable is in fact a reference to an instance variable. Therefore, specifying *required instance variables* is necessary if those instance variables are to be used. For example:

```
(defflavor LABEL-AUGMENTING-MIXIN
   ()
   ()
   (required-method :SET-LABEL)
   (required-instance-variable LABEL))

(defmethod (LABEL-AUGMENTING-MIXIN :AUGMENT-LABEL) (new-text)
   (send SELF ':SET-LABEL (string-append new-text LABEL)))
```

This example demonstrates the use of both required methods and required instance variables. **LABEL-AUGMENTING-MIXIN** appends a string to the current label, then replaces the label. It uses the **LABEL** instance variable to get the current label, and then sets the new label by sending the **:SET-LABEL** message. The **defflavor** specifies that both the instance variable and the method must be supplied.

As an extension of required instance variables and methods, there are times when a flavor **EXTENSION-OF-NILLY** wants to tell the Flavor system that it relies on the details of some other flavor **NILLY**, and that if some third flavor includes **EXTENSION-OF-NILLY**, it must also include **NILLY**. **NILLY** is a *required flavor*. This not merely a shorthand way of specifying that all the instance variables and all the methods of **NILLY** are required, but rather that **EXTENSION-OF-NILLY** needs **NILLY** itself in order to function correctly. Normally, **EXTENSION-OF-NILLY** would merely include **NILLY** as a component flavor, but when these two flavors need to be separated in the ordering, direct inclusion is not feasible.

For example: flavor **BORDER-AND-LABEL-FEATURES-MIXIN** relies on details of both **BORDER-MIXIN** and **LABEL-MIXIN**, and it shadows primary methods in both of them. However, it is still legitimate for **BORDER-MIXIN** or **LABEL-MIXIN** to occur in either order. In this case, **BORDER-AND-LABEL-FEATURES-MIXIN** would have the other mixins as required flavors. The programmer would then be free to choose the ordering. Though in most cases this would best be handled by splitting **BORDER-AND-LABEL-FEATURES-MIXIN** into two flavors, there are times where this might not be desirable.

An extension of required flavors is the concept of *included flavors*. An included flavor *X* is treated as a component flavor after all other flavors have been examined and *X* is not found. When using included flavors, the Flavor system makes two passes over the flavor structure. On the first pass, it handles all component flavors in the standard fashion. On the second pass, all flavors are again scanned and included flavors are considered components if they have not been previously mentioned. Making a flavor included is a way of asserting that the flavor must be in the final flavor structure, yet it is to be a default and considered last in the ordering. For example:

```
(defflavor SPECIAL-MARGIN-A ()
   ()
   ()
   (included-flavor SPECIAL-MARGIN-MIXIN))

(defflavor SPECIAL-MARGIN-B ()
   ()
   ()
   (included-flavor SPECIAL-MARGIN-MIXIN))
```

Assuming that flavor **SPECIAL-MARGIN-MIXIN** uses room in the margins of a window, it would be unfortunate if its position in the ordering was tied to the position of either **SPECIAL-MARGIN-A** or **SPECIAL-MARGIN-B**. However, if either of those flavors is used, **SPECIAL-MARGIN-MIXIN** must be a component flavor. Making **SPECIAL-MARGIN-MIXIN** be an included flavor resolves the conflict.

Combination types

The ability to define new combination types makes the Flavor system arbitrarily extensible. Daemon type[14] combination is appropriate for almost all of the messages defined by a set of flavors. However, there are usually several messages in the set that need to have their methods combined in a different fashion. A declaration in the flavor declaration tells the Flavor system what combination type to use for a particular message. If no combination type is specified, then daemon combination is used. If a type is specified, it can be specified by any flavor in the lattice. If more than one flavor specifies the combination type, then the specifications must be consistent.

There is one parameter that can be given when specifying a combination type: whether the methods are to be combined in forward or reverse order. In the case of daemon combination, this is not very useful. However, many combination types have only one type of method, and in that case the parameter is meaningful. Some examples of useful non-daemon combination types are:

[14] See section @ref(DaemonCombination), page @pageref(DaemonCombination)

- *Lisp **OR***
  Returns the first non-null value of a method. The method ordering parameter is meaningful.
- *Lisp **AND***
  Like **OR** but returns null if any of the methods returns null, else returns the value of the last method.
- *List*
  Returns a list of the values returned by each method.
- *Inverse list*
  Given as an argument the list produced by list type combination, calls each method with successive elements of the list. If several flavors contribute methods to a list type result, and also contribute methods to this type combination, then these methods get called with the value of the flavor's list type method. Thus, the list type message collects information from a set of several flavors, and inverse list type message "puts the information back". This is a nice example of a non-trivial use of method combination.

It is useful, especially in the case of non-daemon method combination types, to be able to specify a default method to be used if no untyped (i.e. primary) methods are supplied. The *default* method type is used for this purpose. If no untyped methods are specified for a particular message, then the default type methods are turned into untyped methods.

Naming Conflicts

A problem with the Flavor system is the potential for naming conflicts between instance variables of different flavors, or between messages of different protocols. The Flavor system defines no active mechanism to resolve such conflicts as they have never been a problem in practice. The message name conflicts can be resolved by making a message name consist of both a name and a protocol with respect to which the name should be interpreted. Making the protocol an explicit entity has other desirable side-effects, such as providing a repository for documentation [ref Xerox here somewhere again, CLU?].

The instance variable name conflict problem can be solved in several ways. A declaration could be added that states that certain instance variables of a flavor *X* are accessible only to flavors that have *X* as a component, and explicitly *import* the instance variables. For example, flavor **FLAVOR-WITH-HILLY** defines instance variable **HILLY**, and declares it *local*. Flavor **ANOTHER-FLAVOR-WITH-HILLY** also defines an instance variable **HILLY**, but doesn't declare it local. Combining these two flavors into one would then have no naming conflicts, as either **FLAVOR-WITH-HILLY**'s **HILLY** is imported, in which case it shadows the normal **HILLY**, or it is not, in which case the normal **HILLY** is visible. Instance variables could also be defined with respect to a protocol. In this case, the solution is similar to the message name conflict solution.

Wrappers

None of the standard method combination types provide any sort of *encapsulation* ability: it is not possible for a method to create a dynamic environment for some other method(s). For example, it might be necessary to lock a lock during the execution of all relevant methods.

Locking the lock in each method is not acceptable, as there would be time slots where another requestor could be granted the lock, thus destroying the atomicity of the original operation. Another example is that of *dynamic binding*,[15] where a variable is bound during the execution of the relevant methods. Yet another example is the performing of an action that needs to be undone when all of the methods complete execution.[16] A mechanism is needed to define a method that *wraps* dynamically around the execution of the other methods: in other words, the other methods to be executed must be called from within the execution of this *wrapper*.

Since a wrapper needs to enclose other methods, a mechanism must exist for the remaining methods to be passed to the wrapper, which can then call them at the appropriate time. The Flavor system is already generating a "combined method". Thus, wrappers can be implemented as source-to-source transformations that take the source code for a combined method and return code that includes the combined method within it.[17] This new code now becomes the combined method. The system then iterates over all wrappers, and the final combined method now reflects the non-wrapper methods with the wrappers around them.

Another way of implementing wrappers would be to have the wrapper be just like a non-wrapper method, but have it take an extra argument which would be a kind of *continuation* – it would get called with the remaining arguments to the wrapper (which would be the arguments in the original message). Thus, the wrapper would execute some code, invoke the continuation, and when the continuation returned, the wrapper would continue execution. For example:

```
(defwrapper (SOME-FLAVOR :SOME-MESSAGE) (continuation arg1 arg2)
   (with-THE-LOCK-locked
     (invoke-continuation continuation arg1 arg2)))

(defmethod (SOME-FLAVOR :SOME-MESSAGE) (one-arg another-arg)
   (setq PROTECTED-INSTANCE-VARIABLE (+ one-arg another-arg)))
```

In this example, a wrapper is defined that locks **THE-LOCK**, and then invokes the continuation. A sample method then modifies **PROTECT-INSTANCE-VARIABLE**, which is an instance variable that is presumably protected by **THE-LOCK**. The code is written in this manner, so that before and after type methods can look at the instance variable without fear that it will change, since the lock will be locked atomically during the execution of all of the methods.

The latter scheme is more elegant than the former for several reasons. First, it allows the wrapper to modify the arguments to the continuation, which is occasionally a useful technique. Second, it does not require duplicating the code of the wrapper for every combined method that is generated. This would happen if the wrapper produced a large combined method by source-to-source transformations, since a new combined method is necessarily generated for every instantiated flavor that includes the wrapper. If the wrapper is large, this can lead to a significant savings. The former scheme is more powerful, however, in that the latter scheme can be implemented in terms of it.[18]

---

[15] As in Lisp, see [MAMA].

[16] This is an extension of binding. See **UNWIND-PROTECT** in [MAMA].

[17] Thus, in the Lisp Machine implementation, wrappers are just like macros.

[18] This is in fact what is done in the Lisp Machine implementation. The macro style wrappers were implemented first. The need for the continuation style was not realized until much later, at which time they were implemented in terms of the macro style

One final question remains, and that is where in the ordering to include the wrappers. Two choices are available: the wrapper could be included where it was found, and wrap around only the methods to the right of it; or, the wrappers could be moved out and processed before all other methods. The latter choice is the most useful. For example, in the case of a lock, the rightmost flavor wants to supply the locking wrapper, but that the lock wants to remain locked over all methods, even ones to the left. Experience shows that this is the correct behavior.[19] However, it is desirable to supply both behaviors.

Finally

Future work

Making protocols explicit is one of the most important ideas to be explored.[20] Once this is accomplished, some difficult problems can be more easily solved. Several of these problems, whose potential solution is via the use of protocols, have been mentioned in the text. The naming conflict problem is one of them. Another application of explicit protocols relates to security: each protocol could specify from which other protocols it may receive messages (to be from a protocol *X* means to be running a method invoked by a message of protocol *X* being sent). If this specification is permitted on a per flavor/protocol-pair basis, then it provides a way of isolating certain internal protocols from the "outside world". For example, one might want to create a window whose label cannot be modified by the user. Instead of redefining all of the messages associated with a label, the label protocol is made inaccessible to message sent from outside the environment of the object.

More investigation into the areas of "self-documentation" and programming environments for use with Flavors is necessary. With Flavors, it is possible to build complicated systems whose behaviors come from many scattered places. Expediently locating these places and finding the details of the behavior require assistance from the programming environment. Protocols may also play some role here as they provide a place to isolate certain kinds of knowledge. For example, what a message's generalized description is and what flavors support the protocol.

Meta-protocols demand further investigation. They may serve some of the same purposes as protocols, but at the level of protocol combination: as a repository for documentation, as a place to store conventions in a program enforceable manner, etc. The concept of the meta-protocol is a direct outgrowth of this paper, and has not been adequately explored.

Related Work

ThingLab is a program, embedded in Smalltalk, that permits the user to graphically construct constraint networks [ref Borning]. It has inside of it a multiple superclass scheme. It is notable in this context as one of its early implementations of multiple superclasses utilized the idea of method combination, without formalizing it.

---

wrappers. Upon reflection, the continuation style wrappers appear to be the preferred form. For a better description of the macro style wrappers as implemented on the Lisp Machine, see [MAMA].

[19] In the Lisp Machine implementation, the former choice is not available. It has been requested only once or twice.

[20] Such work is currently going on with Smalltalk at Xerox PARC [ref Ingalls?].

The PIE system, by Bobrow and Goldstein [ref PIE papers], is a recently implemented multiple superclass scheme that handles interactions between superclasses via constraints between the instance variables, and via *views*. It is also embedded in Smalltalk. A view is much like a protocol: a message is sent with respect to a view, and each superclass is responsible for handling one particular view of the object [check this for accuracy!!], and the constraints are used to keep the views consistent. In the Flavor system, it is possible to implement this scheme with a method combination type and several other simple mechanisms.

Conclusions

Object-oriented programming systems are not panaceas. However, the problems with these systems that the Flavor system solves have prevented them from being effectively utilized in many applications for which they are ideally suited. On the Lisp Machine, a very large window system,[21] designed to be both simple to use in a simple manner, yet easy to extend for more sophisticated applications, **could not have succeeded without the Flavor system**.

The Flavor system is a practical general-purpose non-hierarchically based object-oriented programming system. Many of the ideas, especially those in the Conventions section, have developed through extensive experience with the system on the MIT Lisp Machine: they are very difficult to justify in a compelling fashion. However, the utility of the Flavor system has exceeded expectations – many of the design goals were met and surpassed.

"*Try it, you'll like it*" – Alka-Seltzer.

Acknowledgments

The author gratefully acknowledges the contributions of the following people:

Mike McMahon, Dave Moon and Daniel Weinreb, for invaluable help with both the theory and practice of Flavors on the Lisp Machine.

Bill Gosper, for encouragement while developing on the initial theory, and while writing this paper.

Alan Borning and Dan Ingalls, experts on message passing, protocols, and Smalltalk.

Smalltalk, on whose shoulders much of this work is based.

Bernie Greenberg, for his Multics MacLisp Flavor implementation, and help with this paper.

Gary Drescher, for many insightful thoughts about the nature of Flavors and the universe.

Author's Notes – December 20, 2003

This document was created from a Scribe-formatted source with a file date of 9/5/1992. The previous revision to the text predates that time stamp. This document has been reformatted for Microsoft Word, spelling errors have been corrected and minor grammatical errors have been fixed. As this document represents an effort to preserve an older paper in a modern form, no substantial changes have been made to the text herein (though it certainly begs for them!)

[21] About one million characters of Lisp source code.