It is well-known that the computing industry is moving at a rapid pace. Every few months there is an introduction of a yet faster machine and new standards for the price-performance ratio. The trend is towards more power, more memory, larger disks, and higher speed networking. This trend shows no sign of abetting.

In the past, new capabilities of the computers we use have led to new ways of using them. The interactive use of computers today needs only to be compared to the batch usage of the past to understand this point. However, it is not clear if particle physics is taking full advantage of our new computer's capabilities.

Less well-known are equally important developments in the way computers are programmed. The object oriented programming (OOP) paradigm, artificial intelligence techniques, computer-aided software engineering, etc. are but a few of the new developments occurring in industry and elsewhere.

Many of these software technologies become practical due to the change of the computing capabilities. Particle physicists, in general, have a difficult time keeping up with the pace.

LEP is operational and HERA about to start. There is now some breathing room between the daily pressures to make things work and preparation for new initiatives such as the SSC, LHC, and various "factories". The next couple of years is a unique time to study new software technologies and to prototype some systems so that they are well understood before making final decisions on the environments to be used in the future.

To help disseminate information on the new software technology, the editors of Particle World have initiated a series of articles on software research and development. The first two of these appear in this issue. The editors hope that these series will be educational and informative.

OBJECT-ORIENTED PROGRAMMING(*)

P.F. Kunz

Abstract

This article is an introduction to object-oriented programming (OOP) techniques. It tries to explain the concepts by using analogies with traditional programming. The object-oriented approach is not inherently difficult, but most programmers find a relatively high threshold in learning it. Thus, this article will attempt to convey the concepts with examples rather than explain the formal theory.

1. Introduction

In this article, the OOP techniques will be explored. We will try to understand what it really is and how it works. Analogies will be made with traditional programming languages in an attempt to separate the basic concepts from the details of learning a new programming language. Most experienced programmers find there is a relatively high threshold in learning the object-oriented techniques. There are a lot of new words for which one needs to know the meaning in the context of a program. Words like *object*, *instance variable*, *method*, *inheritance*, etc. As one reads this article these words will be defined, but the reader will probably not understand at that point the where and why of it all. Thus, the article is like a mystery story, where we will not know who has done it until the end. Our word of advice to the reader is to have patience and keep reading.

2. Key ideas and concepts

The first key idea is that of an *object*. An object is really nothing but piece of executable code with local data. To the FORTRAN programmer, an object can be considered a subroutine with local variable declarations. By local it is meant data that are neither in COMMON blocks, nor passed as an argument. These data are private to the subroutine. In object-oriented parlance it is *encapsulated*. Encapsulation of data is one of the key concepts of OOP. The second key idea is that a program is a collection of interacting objects that communicate with each other via *messaging*. To the FORTRAN programmer, a message is like a CALL to a subroutine. In OOP, the message tells an object what operation should be performed on its data.

(*) This work was supported by the Department of Energy, contract DE-AC03-76SF00515.

The word *method* is used for the name of the operation to be carried out. The last key idea is that of *inheritance*. This idea can not be explained until the other ideas are better understood and will be treated later on in this article.

3. Encapsulation and messaging

An object is executable code with local data. These data are called *instance variables*. An object will perform operations on its *instance variables*. These operations are called methods. To clarify these concepts, consider the FORTRAN code in fig. 1.

```
      Subroutine anObject (msg, I)
      Character msg* (*)
      Integer I
      Integer*4 aValue
      If (msg .eq. "setValue") then
         aValue = I
         return
      ElseIf (msg .eq. "getValue") then
         I = aValue
         return
      Else
         print ("0Error")
         EndIf
      return
      end
```

FIGURE 1

FORTRAN code sample.

This is a strange way to write FORTRAN, but it will serve to illustrate the key concepts. It also uses FORTRAN extensions that are commonly used. The style of capitalization is that which is recommended for objective programming, but for the moment it is not important for the discussion. For this sample code, the name of the object is `anObject` which is a subroutine with two arguments. The first argument, `msg`, is used as the message, while the second, `I`, is used as a parameter. This object has one instance variable with the name `aValue` which is of type integer. There are two methods defined: `setValue`, and `getValue`. What operations are performed on the data is defined in the FORTRAN code. That is, if the value of the character string `msg` is "`setValue`" then the instance variable `aValue` is set to the value of the argument `I`. If, on the other hand, the string is "`getValue`" then the current value of the instance variable is returned via `I`.

To send a message to `anObject` from some other part of the program, one might find code fragments that look like

```
Call anObject("setValue", 2)
Call anObject("getValue", I)
```

In the first line, `anObject` will set its instance variable to value 2, while in the second line the current value of the instance variable will be returned into the argument `I`.

Now the reader should have some of the key concepts understood, at least in their simplest sense. The reader should note the very different style of manipulating data. In traditional languages like FORTRAN, we think of passing data to a routine, via arguments or COMMON blocks. Here the routine, i.e. the object, holds the data as instance variables and we change or retrieve the data via methods implemented for the object.

This style of programming by messages is a rather tedious way to get to the data that we want to operate on. It is time to invent a new syntax, one that could be read through some preprocessor that would generate the code that could be compiled. An example of such a preprocessor is Objective-C [1], a preprocessor to the C programming language. Objective-C is a super-set of C. It adds only one new data type, the object, and only one new operation, the message expression.

An example of Objective-C code is given in fig. 2. This Objective-C code is equivalent to the FORTRAN code shown in fig. 1. In the Objective-C syntax, the code is divided into two parts. The first part is called the *interface*: it is all the code between the `@interface` and the next `@end`. The interface part of the code serves two purposes. It declares the number and type of instance variables, in this case only one, and it declares to what methods the object will respond. The interface is usually placed in a separate file, then included via the standard C include mechanism. The author can only say that the reasons for doing this, if not apparent

```
#import <objc/Object.h>
@interface anObject:Object
{
   int aValue;
}
- setValue: (int) i;
- (int) getValue;
@end

@implementation anObject
- setValue: (int) i
{
   aValue = i;
   return self;
}
- (int) getValue
{
   return aValue;
}
@end
```

FIGURE 2

Objective-C sample.

at this time, will be so later. The second part of the code is the *implementation*: this is all the code between the `@implementation` and the `@end` that follows. Within the implementation, one writes the code for all the methods that make up the object. Each method begins with a "–" and the name of the method. Between the curly brackets "{}" there can be any amount of plain C code, including calls to C functions, and Objective-C message expressions. Even calls to other compiled languages, such as FORTRAN, can be placed here. The example in fig. 2 admittedly does not show very much of that possibility since it is so short.

To send a message to the object in fig. 2, from another object, one might find the following code fragments:

```
id anObject;
int i;
[anObject setValue: 2];
i = [anObject getValue];
```

In these fragments, `anObject` is declared to be data of type object, while `i` is declared to be type integer. The message expression is signalled by an expression starting with the left square bracket "[" and ending with the right square bracket "]". The syntax is a very strange to a FORTRAN programmer, or even a C programmer. There is more behind it than can be understood now, so the reader would do best by not questioning its rational at this point.

So far, we have introduced a lot of new terms and a very different syntax. But what is important is the very different way of handling data. Where we are headed is probably not yet clear, but as was said in the beginning, this article reads like a mystery story, we would not know until the end. The next section will work on a much more concrete example using what we already are beginning to understand.

4. Another example: a histogram object

In this section we will treat a histogram as an object. We will examine the code to do one histogram. In fig. 3 is shown what the interface part might look like. It is a simple example and not meant to be exhaustive or representative of what one would do for production quality code.

The object shown in fig. 3 is of the class `Hist` which inherits from the root class `Object`. The meaning of the words *class* and *inheritance* will be defined later. The instance variables of the histogram object (shown between the curly brackets) are the title, the low edge of the histogram, the bin width, the number of bins, etc. To make the example simple, we have a fixed maximum number of bins (100) and a fixed maximum title size. This is unnecessary in C because these arrays can be dynamically allocated when the histogram is defined, but for our present purposes, we will avoid introducing this feature of the C language.

Once the histogram object is created, the user would first send it messages to fix its title, set its low edge, bin width, etc. These messages might look like the following code fragments:

```
[hist setTitle:"my histogram"];
[hist setLow:0 width:1.];
```

To accumulate and print, the messages might look like

```
[hist acum: x];
[hist print];
```

The implementation of the histogram should be obvious. In the `acum:` method, for example, one would find exactly the same kind of coding one would find in FORTRAN. That is, something like

```
#import <objc/Object.h>
@interface Hist:Object
{
   char title[80];                    /* title of histogram */
   float xl, xw;                      /* low edge and bin width */
   int nx;                            /* number of bins */
   int bins[100], under, over;        /* bins and under/overflows */
}
- setTitle: (char *)atitle;           /* set the title */
- setLow: (float) x width: (float) y; /* set the edge and width */
- setNbins: (int) n;                 /* set the number of bins */
- acum: (float) x;                   /* accumulate */
- zero;                              /* zero the histogram */
- print;                             /* print it */
@end
```

FIGURE 3

Example interface file for `Hist` object.

```
- acum:(float) x
{
    i = (x -xl)/xw;
    if ( i < 0 ) under = under + 1;
    elseif ( i >= nx ) over = over + 1;
    else bins[i] = bins[i]+1;
    return self;
}
```

There is nothing but ordinary C code in this particular method implementation. By the way, we have written the C code like a FORTRAN programmer might, so as to not confuse the issue with short cuts a C programmer might normally use.

It is rare that one wants only one histogram, so we now examine what needs to be changed to have more than one. First of all, if we have multiple histograms it is clear that they all behave the same way. In object-oriented parlance, we say there is a *class* of objects called histogram. In our example, the name of the class is `Hist`, as seen on the `@interface` line (fig.3). The only differences between one histogram object and another are the values of its instance variables. Using the right object-oriented words we would say that one histogram object is an *instance* of the class `Hist`. We create an instance of the class `Hist` by sending a special type of message to the class `Hist`. It is called the *factory method*. The messages that are sent to the class are factory methods. The ordinary messages are sent to an object, which is an instance of a class. It is important to remember this distinction.

We send a message to the class to create an object, then we can start sending messages to the object. The code might look like

```
id aHist, bHist;
aHist = [Hist new];
[aHist setTitle: "histo one"];
bHist = [Hist new];
[bHist setTitle: "histo two"];
...
[aHist acum: x];
[bHist acum: y ];
etc.
```

The first message, "new", is sent the class `Hist`. This is known as a factory method. All the classes that are linked together to form the program module are known at run time, just like the subroutines are known in FORTRAN. Classes can only accept factory methods, so to distinguish them from objects, one capitalizes the first letter of the class name. Factory methods return the `id` of the object created. An `id` is a special variable type in Objective-C to identify objects. In the example, we have given these `id`s the names `aHist` and `bHist`. Once an object has been created, i.e. an instance of the class `Hist`, then we can send messages to the object to define the histogram, and accumulate into it. What other changes do we need to make to have multiple histograms? ***NONE***. In fact, we do not even have to write the factory method "new" because it is inherited (we will explain inheritance in sect. 5).

At this point, the FORTRAN programmer is probably confused, since we have shown code which seems to be written for only one histogram, and yet we have many. What is going on? One way to understand it is to look behind the scenes and see how memory is being allocated, as shown in fig. 4. We write code for the class `Hist` which contains the instance variables of the class, its normal methods, and maybe a factory method if it is not inherited. At run time, we message the class `Hist` with a factory method. This method allocates space in memory for the instance variables, and some other things we do not need to concern ourselves with for the moment. Thus, each object of class `Hist` has its private copy of the instance variables. The factory method returns the `id` of the object just created. We can then send messages to this object. Program execution jumps to one of the methods we see in class `Hist`, with the instance variables set to the private copy which belongs to the object we sent the message to. The net result for the programmer is profound. He writes the code for the `Hist` class as if there is only one histogram allowed. In the driver code, however, as many histograms as needed can be created via the factory method, and the system does all the bookkeeping.

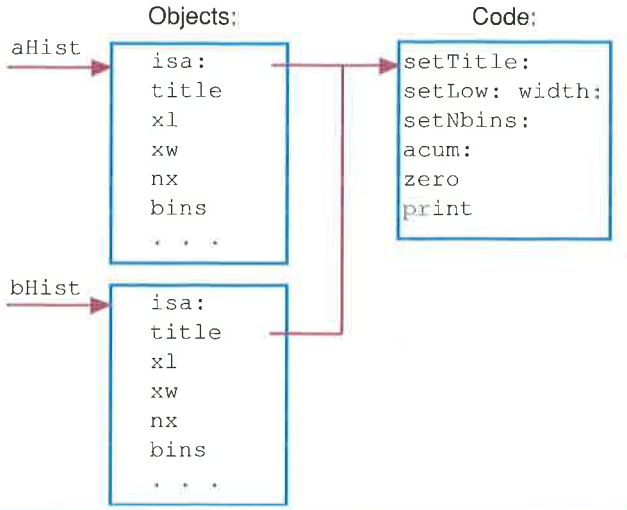

FIGURE 4

Allocation of memory for objects.

Now compare the object-oriented style of writing the histogram code with what one usually finds in a FORTRAN implementation. If we had written FORTRAN code to handle only one histogram, and decided that we needed multiple

histograms, the changes to the code would be extensive. First of all, the local variables that held the definition and bin contents would all need to become arrays, dimensioned by some maximum number of histograms allowed. We would probably put these arrays in a COMMON block and write one routine for each operation we wanted to perform on the histogram, corresponding to the messages in the object-oriented approach. One of the arguments in these routines would be some kind of identifier of which histogram the operation was to be performed. The identifier frequently is not just the index into the arrays but some character string, so we would need to write a look-up table to find the index from the identifier. To allow the flexibility of using the package for a large number of histograms with few bins, or a few histograms with many bins *without* re-compiling, one frequently sees the program allocating space in some large COMMON blocks for the bins and the definitions.

The net result in the FORTRAN implementation is that the person who writes the histogram package writes a lot of bookkeeping code, probably more bookkeeping code than definition or accumulation code. Instead of methods within a class being held together, we have independent routines, related only, perhaps, by some naming conventions. The data instead of being encapsulated are exposed, since they are in COMMON blocks. In short, everything is inside out when compared to the object-oriented approach.

5. Inheritance

Another important aspect of OOP is inheritance which has been alluded to already. Let us start with an example: define an object called `Hist2`, which will be a two-dimension histogram. The interface file might look like the code shown in fig. 5. It is just like the `Hist` object in sect. 4. We will assume that the `show` method prints a table showing the accumulation in each bin. Now, suppose we want to define another form of 2D histogram which shows its contents in 3D form with the Z axis being the contents of the bin, i.e. a `lego` plot. We will call this class the `Lego` class. We can write its interface file as shown in fig. 6. We will make the `Lego` class a subclass of the `Hist2` class.

```
#import <objc/Object.h>
@interface Hist2:Object
{
   char title[80];
   float xl, xw, yl, yw;
   int nx, ny;
   int bins[100][100],...;
}
- setTitle: (char *)atitle;
- setXlow: (float)x Xwid: (float)y;
- setYlow: (float)x Ywid: (float)y;
- acum: (float)x;
- show;
......
@end
```

FIGURE 5

Interface code for `Hist2` class.

```
@interface Lego:Hist2
{
   float plotangle;
}
- setAngle: (float) degrees;
- show;
@end
```

FIGURE 6

Interface code for `Lego` class.

There is only one instance variable and two methods in the class `Lego`. The instance variable `plotangle` is the angle at which the x–y axis should be shown when displaying. The two methods are to set that angle and to plot the histogram. So what happened to all the methods to define and accumulate the lego plot? They are inherited. Notice the `@interface` line in the code hereafter. It says that the class `Lego` is a subclass of `Hist2`. The use of the word *subclass* is a misnomer, because in OOP it does not mean something smaller, it means something bigger or something more specific. When one class is a subclass of another, it inherits all of its superclass's instance variables and all of its methods. Thus, the `Lego` class has all the instance variables of the `Hist2` class and one additional variable: `plotangle`. It also inherits all the methods of `Hist2` and adds one new one: `setAngle:`. What about the `show` method? A subclass can either take an inherited method exactly as it is in its superclass, or it may override it. Since the fashion in which the `Lego` class displays its accumulation is very different from that of `Hist2`, the class `Lego` needs to override the definition of the `show` method with one of its own. The use of the `Lego` object is just like any other object. That is, we might see something like

```
aLego = [Lego new];
[aLego setTitle: "this plot"];
[aLego setXlow: 0. Xwidth: 1.];
...
[aLego setAngle: 45.];
...
[aLego acum: x :y];
[aLego show];
```

Again, it is worthwhile to look behind the scenes and understand how memory is being laid out. Figure 7 shows how

memory is allocated after one lego-plot object is created. The object aLego consists of a concatenation of the instance variables of the Hist2 class and the Lego class. The isa pointer points to the code defined in the Lego class. That class also has a pointer to the code of the Hist2 class. Thus, when aLego is sent the message "setAngle:" the code defined in the Lego class is found. When aLego is sent the message "setTitle:", the method is not found in the code for the Lego class. Instead, the code found in Hist2 class is executed, because of inheritance. On the other hand, when aLego is sent the message "show", the show method in the Lego class is executed (not the show method in the Hist2 class), because the show method in Lego overrides the one in the Hist2 class.

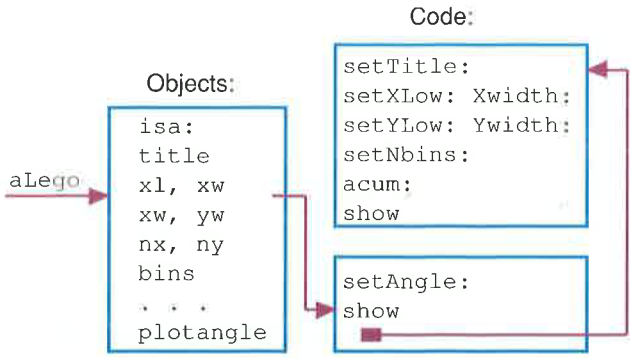

FIGURE 7

Allocation of memory for one Lego object.

One result of inheritance is much less code modification when we want to add functionality. Lego performs everything that Hist2 does and more. If Hist2 gets changed, so does Lego, so it is easier to maintain code. The author of the Lego class never needs to look at the code for Hist2; he only needs to know the methods he wants to override and can add his own new methods at will. It also works in the opposite direction. The lego plot needed an extra instance variable, plotangle. This variable was added to the class without needing to change anything in the Hist2 class to accomodate it.

6. Graphics user interfaces and OOP

Many modern workstations and PCs converse with the user with what is known as a graphics user interface. Essentially the user is presented with a graphic rendering of the data, and several methods or procedures are activated by the user pressing (in fact clicking on) a button.

Indeed, the graphics user interfaces are best implemented with OOP techniques. As an example, fig. 8 shows part of the class structure of NextStep, the graphical user interface developed by NeXT Inc., for use on their computers and licensed to IBM for use on their UNIX workstations. In this figure we see that a button is implemented by the Button class, which is a subclass of the Control class. Since controls are visible on the screen, they are a subclass of the View class, and since all views might respond to mouse input, they are a subclass of the Responder class. For those methods implemented in the Responder class, all subclasses of View, e.g. Control, Box, Text, and ScrollView classes, will behave the same way, since they inherit these methods.

The NextStep class structure also illustrates another aspect of OOP that one frequently makes use of. That is, an object can be composed of many different objects from different parts of the class structure. Imagine an application that has a panel, which is an object for user input, such as the one shown in fig. 9. Note that this application has panels which are a subclass of the Panel class, which in turn is a subclass of Window and Responder classes.

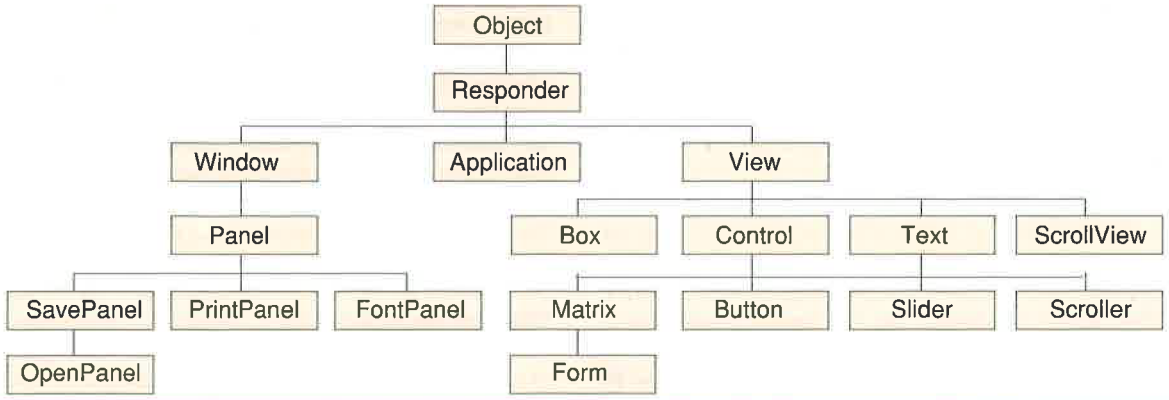

FIGURE 8

Part of the graphics user interface class structure on the NeXT computer.

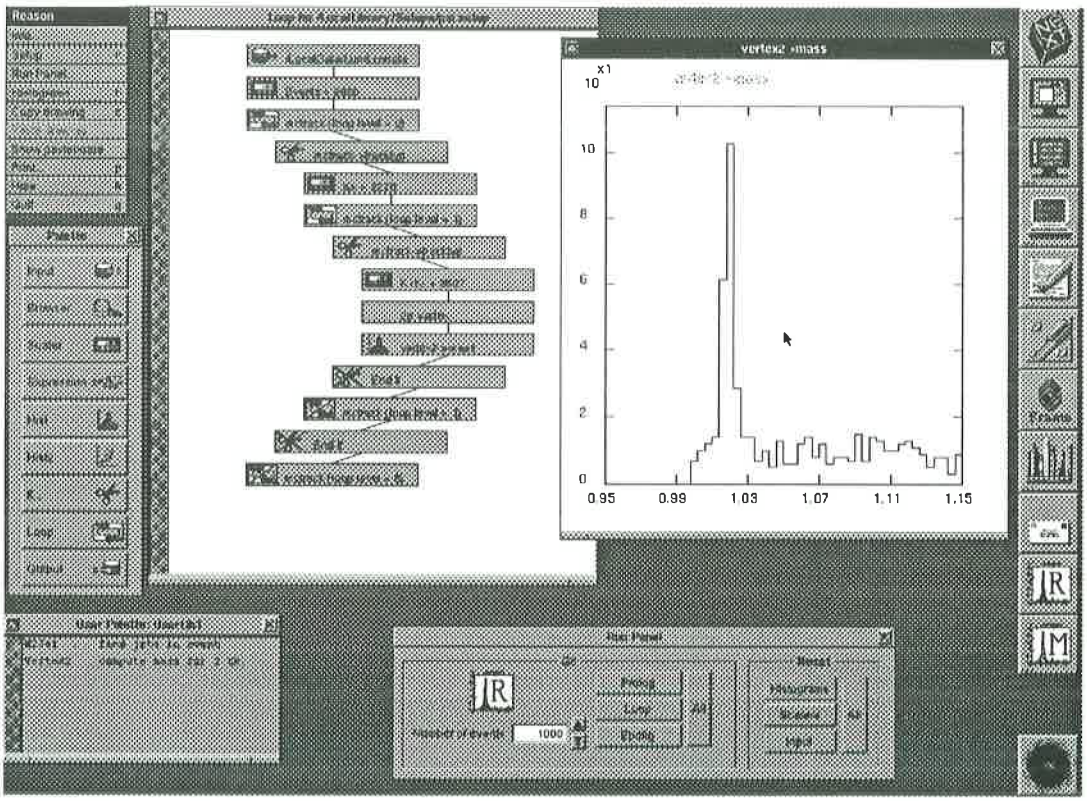

FIGURE 9

Example of application with panels.

The panel contains buttons, sliders, text fields, etc., each of which are also objects which are subclasses of Control and View subclasses. Thus, the panel object for the application is made up of objects from various classes and the ensemble is treated as one object by the application.

7. Object-oriented FORTRAN

So far we have given all the examples in the Objective-C language. This language is just C with one new data type, an object, and one new expression, the message, when compared to the C language. The Objective-C language was originally implemented as a preprocessor to the C compiler. It generates C code which is then compiled and linked to the Objective-C run-time library. It was designed so that the syntax could be added to other languages as well.

Such a preprocessor can also be written for FORTRAN. For the NeXT computer, the Absoft company has done exactly that, in order that programs written in FORTRAN can make use of the NextStep class library. An example of object-oriented FORTRAN code is shown in fig. 10. One can tell this is FORTRAN source code because of the use of symbols like `.false.` One can also recognize that the same syntax is very similar to Objective-C with statements like `@implementation`, and with message expressions imbedded in the code. After passing this source code through a preprocessor, it is compiled by the FORTRAN compiler and linked with the standard NeXT libraries. Thus, one has an existence proof of an object-oriented FORTRAN. However, the current implementation permits only one instance of an object. This limitation may be removed by further development.

8. Summary

This article has presented an overview of OOP. The basic concepts have been explored. The meaning behind words like

```
INCLUDE "appkit.inc"                          ! Include Application Kit
INCLUDE "Timer.inc"
INCLUDE "Cube.inc"                            ! Include the interface

@implementation Cube : View

@+ newView:REAL*4 rect (4)
self = [self newFrame:&rect]
[self setClipping:NO]                         ! This speeds drawing
width = 2.0                                   ! Start with line width of 2.0
suspend = .false.                             ! Start with cube rotating
                  ! Start Timer with a small delay
[Timer newTimer: @0.02D0
+       target:   self
+       action:   Selector("display\0")]
   newView_ = self                            ! Return, by convention, self
   @end

@- step                                       ! Suspend rotation. do a single step
   suspend = .false.                          ! Temporarily turn off
   [self display]                             ! Display new rotation of cube
   suspend = .true.                           ! Suspend cube
   step = self                                ! Return, by convention, self
@end
```

FIGURE 10

Extract from object-oriented FORTRAN code.

instance variables, *methods*, etc. has been explained. We have seen that although the style of programming is very different, it is not inherently difficult.

There are many benefits of OOP. Generally, the program is much more readable and maintainable. Also the code is more easily re-usable and is generally very modular. In short, the goals of software engineering are easily achieved with the object-oriented approach. Compared to traditional programming, object-oriented code has much fewer array declarations, thus minimizing the possibility of inadvertently exceeding array boundaries. Through creation of objects, the system does the kind of bookkeeping that one would need to do in the traditional programming approach. Inheritance makes it easy to modify and extend existing objects, while preserving the encapsulation of data. Overall, it is much easier to implement large sophisticated programs.

In an age where one frequently talks of a "software crisis", the OOP approach seems to offer some real solutions and a programmer who uses the object-oriented techniques can be much more productive.

Reference

[1] B.J. Cox, Object-oriented programming, An evolutionary approach, ed. Addison–Wesley (1986).

Address:
Paul F. Kunz
Stanford Linear Accelerator Center
Stanford University
Stanford, CA 94309 (USA)

Received on June 1990.
Revised on May 1991.