SLAC-PUB-5241
April 1990
(T/E)

OBJECT ORIENTED PROGRAMMING*

Paul F. Kunz
Stanford Linear Accelerator Center
Stanford University, Stanford, California 94309

ABSTRACT

This paper is an introduction to object oriented programming 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 paper will attempt to convey the concepts with examples-rather than explain the formal theory.

INTRODUCTION

In this paper, the object oriented programming techniques will be explored. We will try to understand what it really is and how it works. Analogies will be made with traditional programming 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 paper, these words will be defined, but thereader will probably not understand at that point the where and why of it all. Thus the paper is like a mystery story, where we will not know who's done it until the end. My word of advice to the reader is to have patience and keep reading.

KEY IDEAS AND CONCEPTS

The first key idea 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 is neither in COMMON blocks, nor passed as an argument. This data is private to the subroutine. In object oriented parlance it is ***encapsulated.*** Encapsulation of data is one of the key concepts of object oriented programming. 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 object oriented programming, the message tells an object what operation should be performed on its data. 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, thus it will be treated later on in this paper.

ENCAPSULATION AND MESSAGING

An object is executable code with local data. This data is called ***instance variables.*** An object will perform operations on its instance variables. These operations are called

* Work supported by the Department of Energy, contract DE-AC03-76SF00515.

Invited paper presented at Computing in High Energy Physics, Santa Fe, N.M., April 9-13, 1990.

```
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 --
```

Fig. 1. Sample **FORTRAN** code.

methods. To clarify these concepts, consider the FORTRAN code in Figure 1. 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 is not important for the discussion. For this sample code, the name of the object is `anObject`. The subroutine has 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. The are two methods defined: `setValue`, and `getValue`. What operations are performed on the data is defined in the FORTRAN statements. 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`, while 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 **FORTRAN** routine, 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. Why we are programming this way is probably not yet apparent; that will come later. But for now, the reader should note the very different style of manipulating data. In 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.

One thing the reader might note is that this messaging style of programming is a rather tedious way to get to the data that we want to operate on. Its 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 proper 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 Figure 2. This Objective-C code is equivalent to the **FORTRAN** code shown in Figure 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 de-

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

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

Fig. 2. Objective-C sample.

clares to what methods the object will respond. The interface is usually placed in a separate file, then included via the standard C include mechanism. Once again, the author can only say that the reasons for doing this are certainly not apparent at this time, but will be explained later. The second part of the code is the ***implementation;*** this is all the code between the `@implement at ion` and the next `@end`. 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 **("{}")** can be any amount of plain C code, including calls to C functions, and message expressions. Even calls to other compiled languages, such as **FORTRAN** can be placed here. The example in Figure 2 admittedly doesn't show very much of that possibility.

To send a message to the above object, from another object, one might find the following code fragments

```
id anObject;
[anObject setValue: 23;
i = [anObject getValue];
```

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

So far, we've 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 like I said in the beginning, this paper reads like a mystery story, we wouldn't know until the end. I don't want to lose you, so the next section will work on a much more concrete example using what we already are beginning to understand.

ANOTHER EXAMPLE: A HISTOGRAM OBJECT.

Its time to take another example, something more concrete. I've chosen to treat a histogram as an object. We'll examine the code to do one histogram. Figure 3 shows what the interface part might look like.

The object shown is of the class `Hist` which inherits from the root class `Object`. The meaning of the words ***class and-inheritance*** will be defined latter. 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

```
#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--
```

Fig. 3. Objective-C interface for `Hist` object.

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'll 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.. .

```
- 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, I've written the C code like a FORTRAN program might do, so as to not confuse the issue with short cuts a C programmer might normally use. If you're looking for something profound in all this, there isn't, yet.

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 its 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. The only difference between one histogram object and another is 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. Its 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: xl;
[bHist acum: y I;
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 and functions 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've given these `ids` 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 don't even have to write the factory method, `"new"`, because it is inherited (we'll explain inheritance in a later section).

At this point the **FORTRAN** programer is probably confused, since we have shown code which seems to be written for only one histogram, and yet we have many. What's going on? One way to understand it is to look behind the scenes and see how memory is being allocated, as shown in Figure 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 stuff we not need 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 that belongs to the object we sent the message to. The net result for the program-

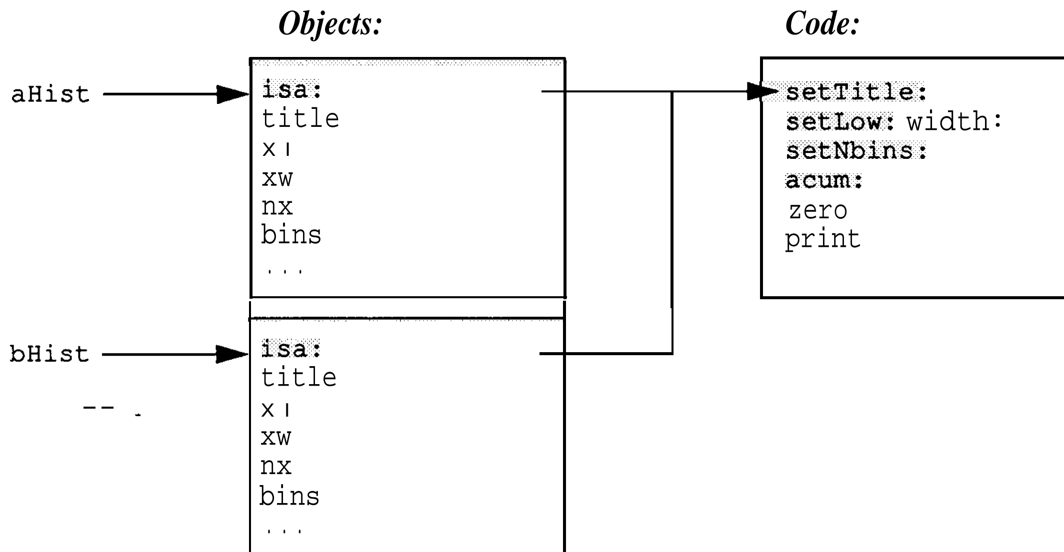

Fig. 4. Allocation of memory for objects.

mer 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

Now compare the object oriented style of writing a histogramming 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 object oriented approach. One of the arguments in these routines would 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 lookup 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 block 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, is exposed since it is in COMMON block. In short, everything is inside out when compared to the object oriented approach.

INHERITANCE

Another important aspect of object oriented programming is inheritance which has been alluded to already. Lets start with an example. Let us define an object called

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

Fig. 5. Interface code for `Hist 2` class

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

Fig. 6. Interface code for `lego` class

`Hist2`, which will be a two dimension histogram. The interface file might look like the code shown in Figure 5. It is just like the `Hist` object in the previous section. We'll 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'll call this class the `Lego` class. We can write its interface file as shown Figure 6.

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 above. It says that the class `Lego` is a subclass of `Hi st2`. ***The*** use of the word ***subclass*** is a misnomer, because in object oriented programing it doesn't mean something smaller, it means something bigger. 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: `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 over-ride it. Since the fashion that the `Lego` class displays its accumulation is very different from that of `Hist2`, the class `Lego` needs to over-ride 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: 45.1;
...
[aLego acum: x :y];
[aLego show];
```

Again, its 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

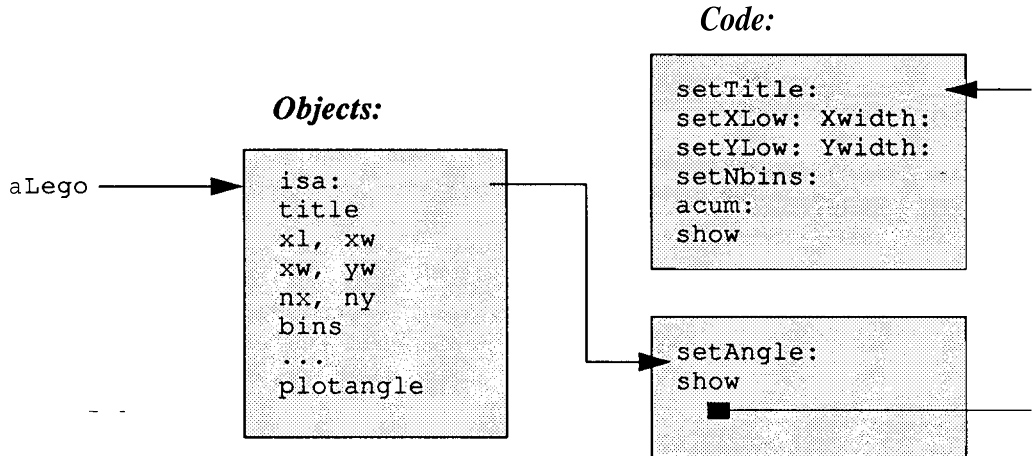

Fig. 7. Allocation of memory for one `Lego` object.

`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` over-rides the one in the `Hist2` class.

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 over-ride 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.

AN OBJECT ORIENTED MONTE CARLO

Another example of the use of inheritance is a particle generation Monte Carlo application written by Richard Blankenbecler.[2] The application was written as a prototype to test the ideas of object oriented programming techniques on a real physics problem. It consists of about a thousand lines of Objective-C code, and it takes unstable particles and decays them into final state stable particles.

The class structure, or inheritance tree, for the Monte Carlo is shown in Figure 8. The root class (which inherits from Object class) for this Monte Carlo is the class Particle.The instance variables of this class are the basic properties of all particles, such as mass, charge, spin, etc. The methods in this class are those that are in common with all particles, such as setting or returning any values of the instance variables. Under the Particle class, there are two classes: Boson and Fermion. This is so, because these types of particles behave differently under certain conditions, so the methods that handle them are necessarily different. Under these two subclasses, one finds the names of

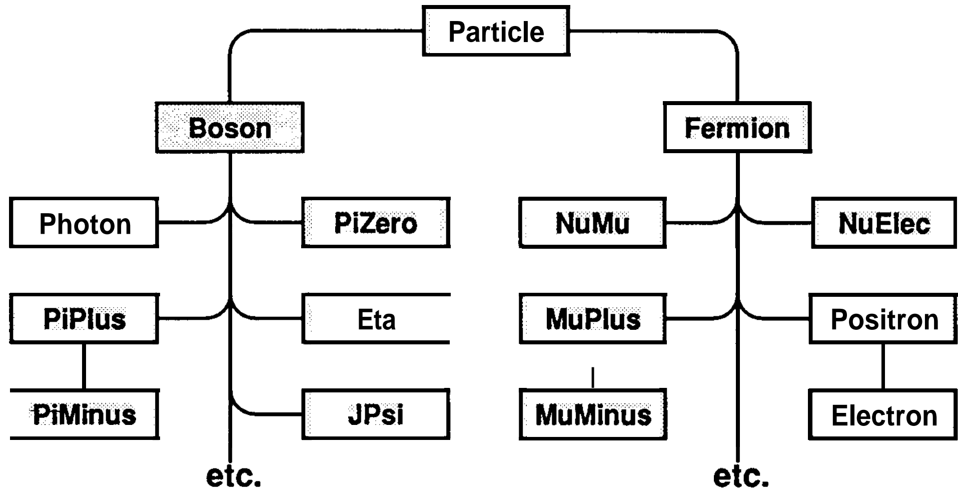

Fig. 8. Class structure of an object oriented Monte Carlo.

known particle states. For example, under the Boson class, we find the classes Photon, **PiPlus, PiZero,** Eta and **JPsi,** etc.

Each of these classes have a factory method with the name `create` in which the mass, charge, and other instance variables are set. Thus, to create a particle a message of the form

```
[PiPlus create];
```

is sent to a class, and an object is returned with the appropriate instance variables set correctly.

Under some of the known particle classes, there is a subclass of the anti-particle. For example, the **PiMinus** class is a subclass of **PiPlus.** In the `create` method of **PiMinus**, the first statement will `be` to send a `create` message to its superclass, the **PiPlus** class. After that, there is a message sent by the **PiMinus** class to itself to make a charge conjugate of itself. This method is named `makeAnt i` and it is an inherited method. The instance variables of mass and charge, for example, are set to their real values in the create method of the **PiPlus** class, thus the **PiMinus** will inherit the correct mass and the mass is only written in the code once. The **PiPlus** class will set the charge to plus one, while the **makeAnti** method will reset it to minus one when it is called by the **PiMinus** factory method.

As a partial summary of the features of this OOP approach to particle decay Monte Carlo, note the following characteristics. The program operations mirror the physics of the process; particle objects respond to direct physics commands. The program operations are in English (strings) to enhance readability, maintainability, and ease of modification. Parameters occur at only one spot in the code, with the particle that it describes, thereby avoiding possible conflicts. There are no arrays (and no exceeding array limits). The program uses well-tested FORTRAN routines to assign phase space momenta to the produced particles. The only "if" statements in the code are in the sections that assign a decay mode for the particle according to its branching fractions. Finally, there are no do loops in the main code; this avoids the bookkeeping in keeping track of the limits.

Output is available in several formats. Finally, the programer does not do explicit memory management nor bookkeeping; the modification of the code is straightforward.

GRAPHICS USER INTERFACES AND OOP

An example of the use of object oriented programming for the graphics user interface toolkit is shown in Figure 9. This example is the class structure of NextStep, the GUI 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.

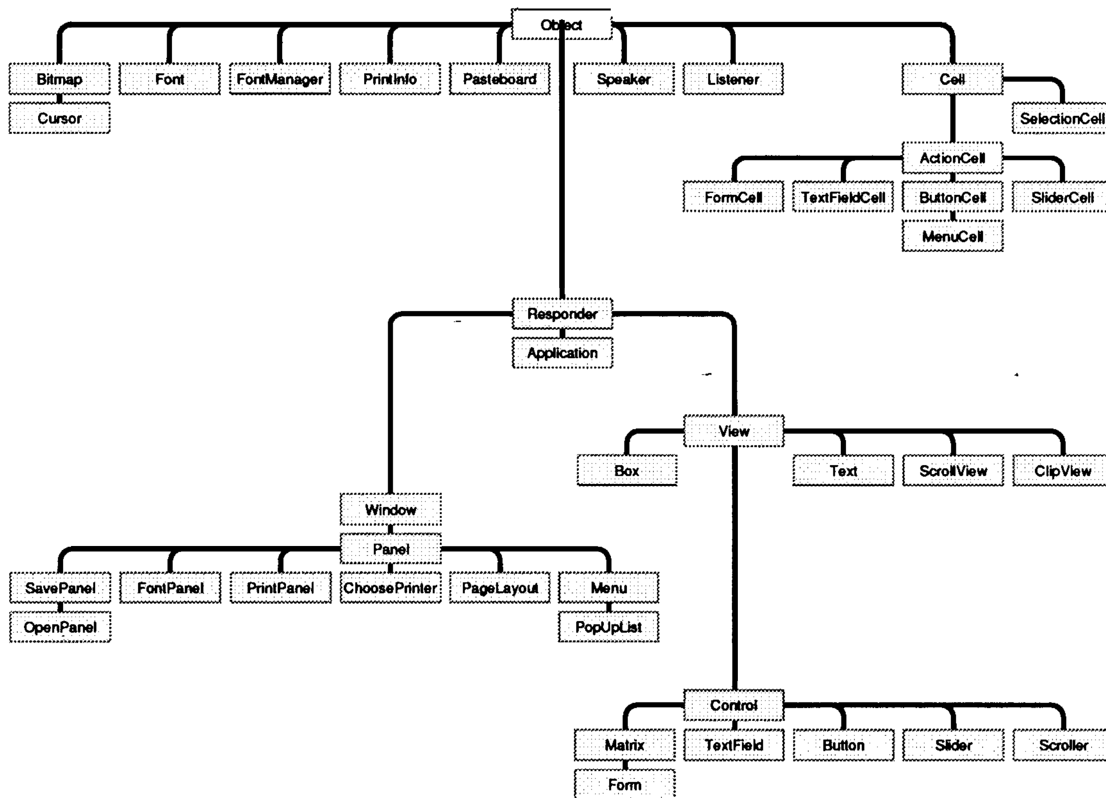

Fig. 9. The Graphics User Interface class structure on the NeXT computer.

The NextStep class structure also illustrates another aspect of object oriented programming 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 Figure 10. 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. The panel contains buttons, sliders, text field, 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.

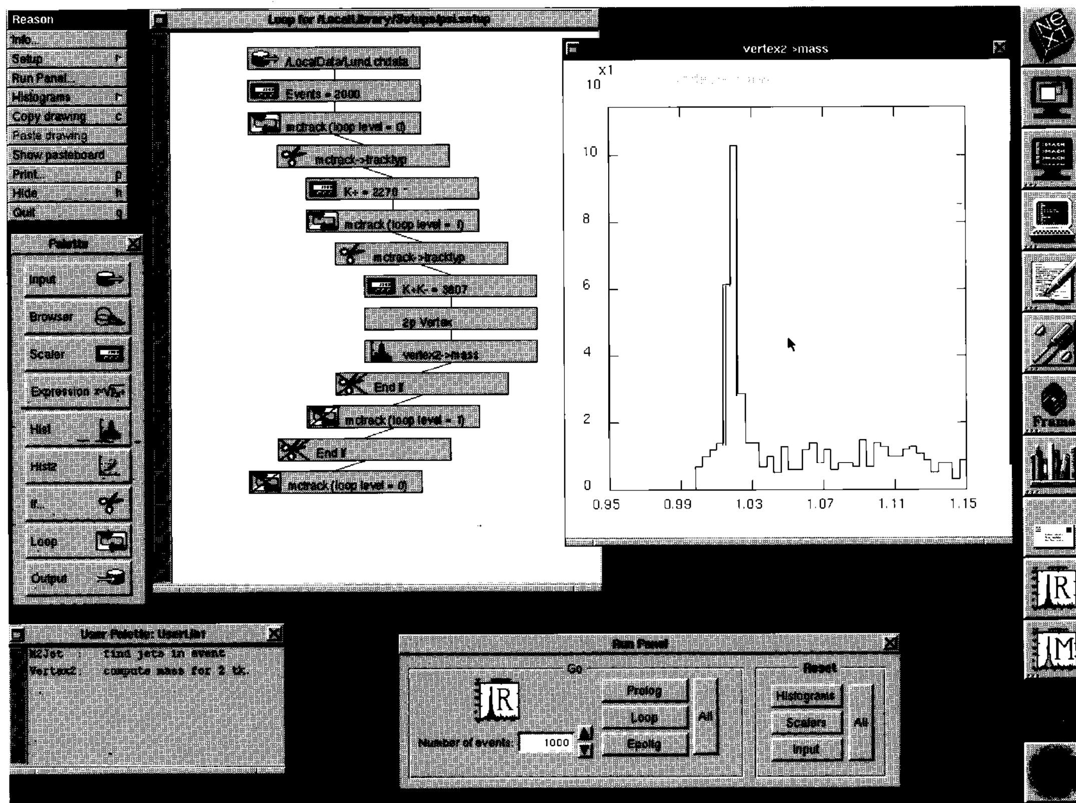

Fig. 10. Example of application with panels.

OBJECT ORIENTED FORTRAN

So far we have given all the examples in the Objective-C language. This language is C with one new data type, an object, and one new expression, the message, compared to the C language. The Objective-C language was originally implemented as a pre-processor 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 pre-processor 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 Figure 11. One can tell this is **FORTRAN** source code because of the use of symbols like `.false.` One can also recognize that it uses the same syntax as Objective-C with statements like `@implementation,` and with message expressions imbedded in the code. After passing this source code through a pre-processor, 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 will be removed by further development by the compiler vendor and planned for release in the summer of 1990.

```
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 11. Extract from object oriented FORTRAN code.

SUMMARY

This paper has presented an overview of object oriented programming. The basic concepts have been explored. The meaning behind word like instance variables, methods, etc. has been explained. We have see that although the style of programming is very different, it is not inherently difficult.

There are many benefits of object oriented programming. 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 declaration, 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 is 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 object oriented programming approach seems to offer some real solutions and a programmer that uses the object oriented techniques can be much more productive.

REFERENCES

1. Cox, Brad J., Object Oriented Programming, Addison-Wesley, 1986.
2. Blankenbecler, Richard, Private communication, to be published.