SLAC-PUB-5629
August 1991
(T/E/I)

Object Oriented Programming*

Paul F. Kunz

Stanford Linear Accelerator Center
Stanford University
Stanford, California 94309, USA

ABSTRACT

This paper is an introduction to object oriented programming. The object oriented approach is very powerful and 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.

1.0 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 the reader 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.

The first key idea is that of an *object*. An object is really nothing more than a piece of executable code with local data. To the FORTRAN programmer, an object could 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.

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 methods. To clarify these concepts, consider the FORTRAN code in Fig. 1. This is a strange way to write FORTRAN, but it will serve to illustrate the key concepts. 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 an input or output 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 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`**; when

* Work supported by Department of Energy, contract DE-AC03-76SF00515.
Invited lecture presented at the 1991 CERN School of Computing, Ystad, Sweden, 23 Aug. - 2 Sept. 1991.

```
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

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. The variables that are local to a FORTRAN subroutine, that is, neither in COMMON nor passed as parameters, are *encapsulated*. The data is protected from being changed or accessed by another routine. It is this that makes it an *object.* An object has boundaries and clearly limits the accessibility of the variables in the routine. A FORTRAN COMMON on the other hand, has no boundaries. It is more like a fog. It spreads out from where you are and you don't know where it ends. You also don't know what else might be in the fog; or what you might run into.

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.

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. An example of such a new syntax is the Objective-C[1] language, which is derived from the SmallTalk language. On most platforms, it is implemented as a translator that generates C code that works with a run time system to handle the messaging. Objective-C as a language is a proper super-set of C. It adds only one new data type, the object, and only one new operation, the message expression, to the base C language.

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. 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 **@implementation** 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 braces ("{}") can be any amount of plain C code, including calls to C functions, and message expressions. Even calls to other compiled languages, such

```
#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
```

Fig. 2 Objective-C example

as FORTRAN can be placed here. The example in Fig. 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;
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 signaled by an expression starting with the left bracket ("[") and ending with the right bracket ("]"). The syntax may seem strange to a FORTRAN programmer, or even a C programmer; it comes from the SmallTalk language. There is a lot more behind it then can be understood now, so the reader would do best by not questioning it at this point. For the remainder of this section, we'll be using the Objective-C language for the examples so that we can study the object oriented concepts without needing to learn a completely new language at the same time. Some readers may not be familiar with this language, or even the C language. As an aid in following the text, the author offers Table 1., which crudely shows the correspondence between the Objective-C and C languages and FORTRAN. Of special note is the use of colons (":") in the method names. In the first instance above, the colon seems to be a separator between the method name and the parameter; which it does except that the colon is also considered part of the method name. In the second instance in the table, there are two colons, each separating the method name from the parameters. The full name of the method contains the two colons as shown by the pseudo-FORTRAN code: **`setLow:high:`**. This is the style of the SmallTalk language and it is done that way for readability. It is very upsetting to FORTRAN programmers, but once one gets used to it, one begins to appreciate its self describing value.

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

Table 1. Correspondence between C, Objective-C, and FORTRAN.

C	**FORTRAN**
`#include <file>`	`INCLUDE "file"`
`int i;`	`INTEGER I`
`float a;`	`REAL*4 A`
`int bins[100];`	`INTEGER BINS(100)`

`char title[80];`	`CHARACTER*1 TITLE(80)or`
	`CHARACTER*80 TITLE`

`char *title;`	*a pointer*

`if ( x < y ) {`
`}`
	`IF ( X .LT. Y ) THEN`
	`ENDIF`

`for ( i = 0; i < n; i++ ) {`
`}`
	`DO I = 0, (N-1), 1`
	`ENDDO`

`i++;`	`I = I + 1`
Objective-C	**FORTRAN**
`#import <file>` *(include if not already included)*	`INCLUDE "file"`

`- set:(int) i`	`SUBROUTINE SET:( I )`
	`INTEGER I`

`- (int) get`	`INTEGER FUNCTION GET()`

`- setLow:(float) x high:(float) y`	`SUBROUTINE SETLOW:HIGH:( X, Y)`
	`REAL X, Y`

```
#import <objc/Object.h>
@interface Hist:Object
{
    char title[80];
    float xl, xw;
    int nx;
    int bins[100];
    int under, over;
}
- setTitle:(char *)atitle;
- setLow:(float) x width:(float) y;
- setNbins:(int) n;
- acum:(float) x;
- zero;
- show;
@end
```

Fig. 3 Objective-C interface for Hist object

much more concrete example using what we already are beginning to understand.

2.0 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 braces) 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, and thus Objective-C, because these arrays can be dynamically allocated when the histogram is defined, but for our present purposes, we'll avoid introducing features of the C language that are not available in FORTRAN.

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 the following code fragments…

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

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 implementation. By the way, I've written the C code like a FORTRAN programer 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: x ];
[ bHist acum: y ];
```

The first message, “**`new`**”, is sent the class **`Hist`**. This is 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. In the example, we've 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. No other changes are needed to make multiple histograms. All the allocation of objects of class **`Hist`**, i.e. the memory management, is done automatically by the compiler. Part of this comes from the base C language, but the Objective-C factory methods make it even more transparent.

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 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 pointers to the executable code. Thus each object of class **`Hist`** has its private copy of the instance variables, but pointers to the same code. The factory method returns the **`id`** of the object just created. It is actually just a pointer to the object. We can then send messages to this object. The run time support uses the pointers to find the code. Program execution jumps to one of the methods we see in class **`Hist`**. As the code executes, it sees only the instance variables of the object to which we sent the message. 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

At this point, we need to explain more about factory methods and what they do. We have already seen that they will allocate memory space. It is also a place where one may initialize an object to a set of default values. For example, the following code fragment could be the factory method of our histogram object

```
+ new
{
    self = [ super new ];
    xl = 0.0;
    xw = 1.0;
    nx = 0;
    strcpy( title, "none" );
    return self;
}
```

The "+" before the method name distinguishes a factory method from an ordinary one, otherwise the implementation proceeds as normal. The full explanation for the first statement will have to wait until the next section. At this

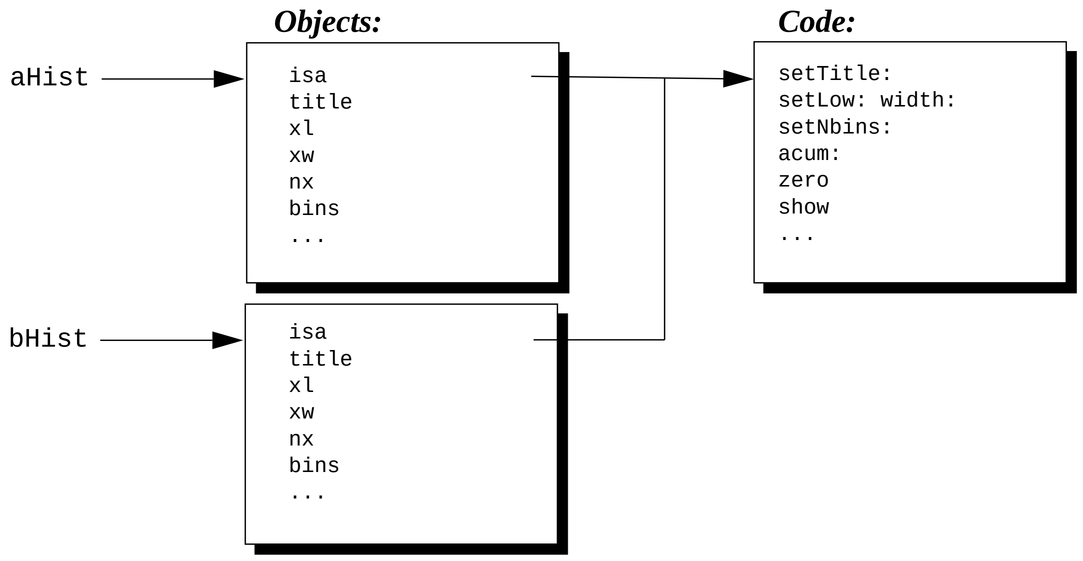

Fig. 4 Allocation of memory for objects

point we'll only say that the message to **super**, a reserved name, does the memory allocation and returns another reserved name, **self**. This name, **self**, is how an object refers to itself. For example, the following two statements are equivalent…

```
xl = 0.;
self->xl = 0.;
```

C programmers will recognize that **self** is a pointer to the memory allocated for the instance variables. The remaining statements in the factory method are straight C code and need no further explanation.

A class can have more than one factory method. This might be used, for example, to simultaneously create an object of a class and set some of the initial values as is done with the following code fragment

```
+ newWithTitle:(char *)aString
{
    self = [super new];
    strcpy( title, aString );
    return self;
}
```

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 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 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* recompiling, one would like to get away from fixed arrays in COMMON blocks. In its place we find a 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 a COMMON block. In short, everything is inside out when compared to the object oriented approach.

Of course as a user of the histogram package, one doesn't really care about the difficulty of the implementation. Object oriented programming, at first, seems to offer no benefit. It's worth mentioning two items along these lines. The first is that what is provided with such packages might be limited by the implementation language. OOP technology frees the programer of a lot of tedious work so he can concentrate on providing a better product. And secondly, the object oriented technology applies itself equally well to physics code since there is a lot of *bookkeeping* code in dealing with tracks, vertices, etc. We'll see examples in the latter part of this paper.

3.0 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 **Hist2**, which will be a two dimensional histogram. The interface file might look like the code shown in Fig. 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.

Note that **Hist2** has some of the same method names as those previously defined in the **Hist** class. Does this mean that one can not have both **Hist** and **Hist2** classes in the same program? No, one can use the same method names in many classes. This is called *polymorphism* and it

```
#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 Xwidth:(float)y;
-setYlow:(float)x Ywidth:(float)y;
-acum:(float)x and:(float)y;
-show;
@end
```

Fig. 5 Interface for Hist2 class

```
#import "Hist2.h"
@interface Lego:Hist2
{
    float plotangle;
}
- setAngle:(float) degrees;
- show;
@end
```

Fig. 6 Interface code for Lego class

allows one to write code that is much easier to understand by re-using the name space for both data and function. With a language like FORTRAN one can only safely re-use the name space for variables local to one subroutine or function. Attempting to have variables of the same name in two different COMMON blocks all too often leads to clashes when both COMMON blocks are needed in the same routine. Also, subroutine and function names must be unique in one program. From this, one can see that the methods in OOP are not just subroutines or entry points.

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 Fig. 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 **`Hist2`**. The use of the word *subclass* is a misnomer. One should not confuse subclass with subset. 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' 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 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 the following code fragments…

```
aLego = [ Lego new ];
[ aLego setTitle:"this plot" ];
[ aLego setXlow: 0. Xwidth: 1. ];
...
[ aLego setAngle: 45. ];
...
[ aLego acum: x and: 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 **`Hist2`** class and the **`Lego`** class. It

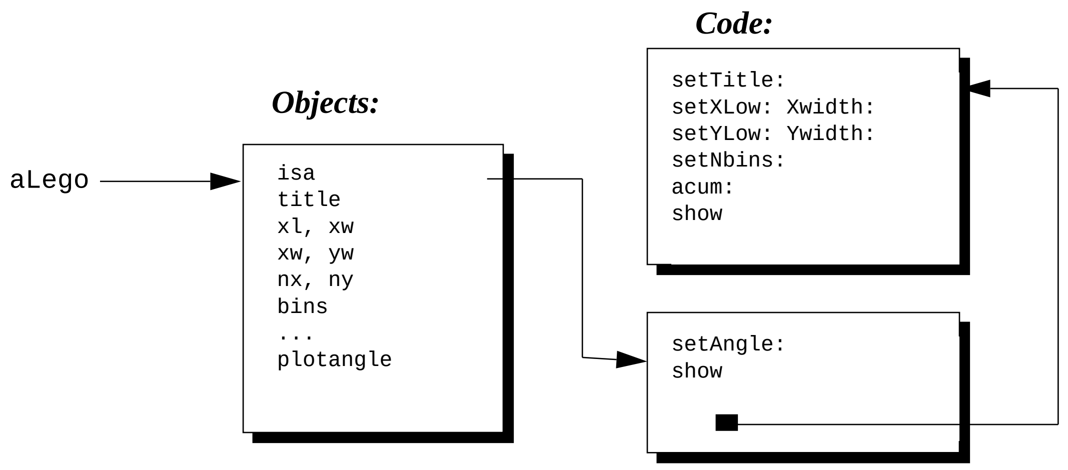

Fig. 7 Memory layout for Lego object

is as if the block of memory for storing the **Hist2** instance variables has been extended to accommodate those of **Lego**. 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 method in the **Lego** class is executed, because the **show** method in the **Lego** class 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. 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, if **Hist2** changes, then **Lego** only needs to be re-compiled. 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.

One programming note should be mentioned now. Note that the interface file for **Lego** (Fig. 6) includes the interface file for **Hist2**. This is necessary so that when **Lego** is compiled, the compiler can know what is inherited. This is one reason why the interface for a class is kept in a separate file; the so-called header file. Another reason is dealing with messages. If in some object one has a message like

```
[ aLego setAngle: 45. ]
```

then the compiler needs to know the type of the parameter (e.g. int, float, etc.) and the type of the return value, if any. Thus, in the source code of the object that sends the above message, the interface file of **Lego** must be made visible to the compiler by including the **Lego** class header file. In practice, the header file is a very good place to put documentation in the form of comments. In the ideal world, the user of an object would understand what operations are performed by the object just from well chosen method names. In practice, however, some documentation is necessary and in many cases, the user might have to read the implementation code as well.

To further illustrate the use of inheritance, let us consider another example. Take a drawing program such as MacDraw or Canvas on the Apple Macintosh. With the mouse, the user can pick from a number of pre-defined shapes such as circles, rectangles, polygons, etc. One can move and resize these shapes in the drawing window. Object oriented programming is a natural way to implement such a program. One could define a generic shape class from which one could sub-class each of the particular shapes. In the shape class, one would find instance variables such as the x-y position on the screen and the size of the shape. Thus the methods to re-position the shape on the screen, or change its size would be implemented in the generic shape class. Any particular shape such as a rectangle, however, would have its own class derived from the generic shape class and contain its own drawing method. A rectangle would fill its height and width. Methods to set the line width and color, to fill the area or not, etc. would be found in the generic shape class.

The class hierarchy that one should use in an application is sometimes difficult to design. The association of physical objects with programming objects is frequently a good rule to follow, but is not always the case. For example, one could argue that a square should be a subclass of a rectangle since it is a kind of specialized rectangle. However, making one's class hierarchy in this manner only confuses the program design. If one tried to define a square class as a subclass of the rectangle class, then how does one decide what the length of the square's side should be? That is, which inherited instance variable, height or width, should one use? It is better to leave a square shape as an instance of the rectangle class; one which happens to have equal height and width.

Another example of confusion results in trying to make the rectangle class as subclass of a polygon. If the polygon class responded to a message to increase the number of its sides, what should the rectangle subclass do if it received such a message? It can't add one side to its shape because then it would no longer be a rectangle. Over-riding the inherited add side method to do nothing doesn't seem natural either. The answer is the rectangle class should not be sub-classed from a polygon class. The rectangle class is simple, it fills its area with a rectangle. It's the polygon class which is more specialized and complex. It must not only fill its area, but have an additional instance variable to know the number of sides it has.

Although arriving at a good class hierarchy for an application may be difficult, especially for the beginner, it is generally not difficult to change the hierarchy. That is, once the problem becomes better understood, one can move methods and instance variables up or down the class hierarchy. For example, if one discovered a set of classes that

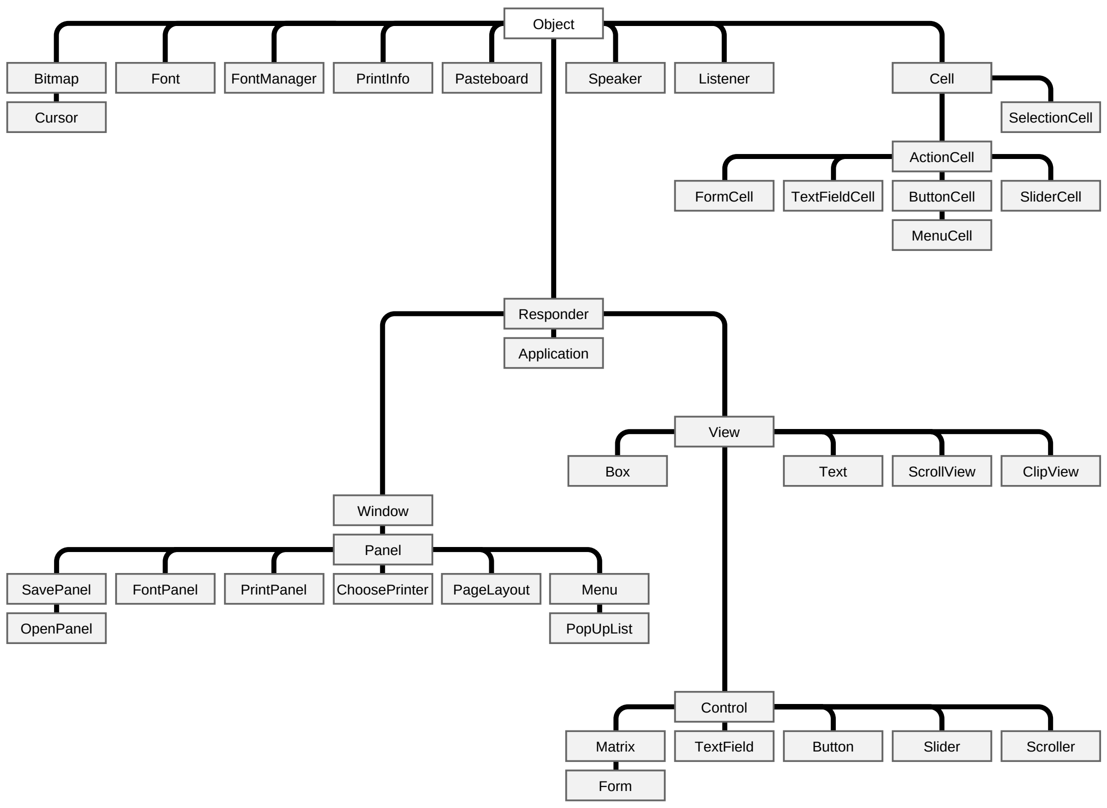

Fig. 8   Graphical User Interface class hierarchy

all shared the same or nearly same properties, then one could add a super class for this set in which those properties could be stored and the methods for operating on them defined. This super class would become a abstract class, like the generic shape class of a drawing program, and the subclasses would become more specific, like rectangles, circles, etc. of a drawing program.

4.0 Graphical User Interfaces And Oop

An example of the use of object oriented programming for the graphical user interface toolkit is shown in Fig. 8. This example is the class structure of NeXTstep, the GUI developed for NeXT computers. 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 or keyboard 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. Classes which are sub-classed from the View class all know how to draw themselves inside a window or panel. Classes which are sub-classed from the Control class have the additional property that when they receive an event from mouse or keyboard, they may send a message to another object, even one that does not present itself on the screen. For example, a slider that is clicked and dragged, will send a message to some object, perhaps with the current value of the slider as a parameter.

The use of the NeXTstep class structure can also illustrate 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 hierarchy. The panel shown in Fig. 9, for example, is one used to display particle properties stored in a data base object. It was developed for the Gismo project[2] and later used within the Reason project[3] as well. Note already that this panel object is used by another object to display its val-

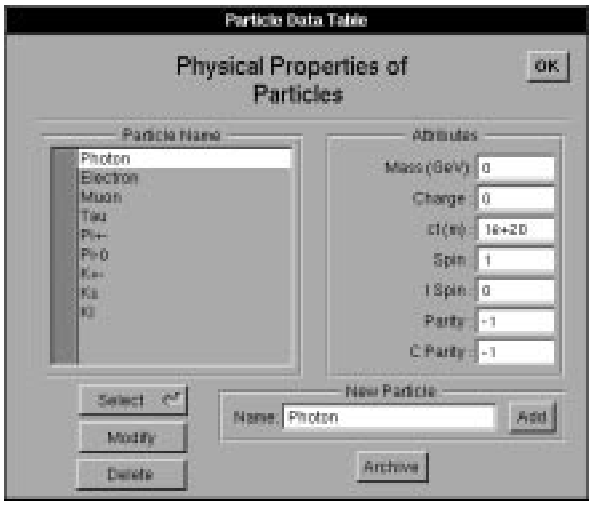

Fig. 9 Panel object with imbedded view objects

ues. This implies that the data base object has an instance variable that is the **id** of the panel object. The panel contains buttons, text fields, scroll-able views, 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. The panel is effectively a container object which controls its position on the screen. It contains a number of view objects which draw themselves within the panel. Those that are also control objects can send messages between themselves, the panel, or the database object that owns the panel, depending on user input. Likewise, the database object can send messages to the panel object or any of the view and/or control objects contained by the panel. Each object manages its own little world as defined by the instance variables of its class and its inherited instance variables. The application works by the objects sending messages to each other.

The lesson to learn from this complex panel object is to make the distinction between the class hierarchy and other hierarchies that might appear in an application. The former defines for a subclass what methods and instance variables are inherited, while the latter defines how various objects are linked together in an application. For example, the panel object contains a view object which holds its contents. Within that content view there are a number of different kinds of sub-views which may also contain sub-views. This forms a view hierarchy. But each one of these sub-views derives their class definition from the same point within the class hierarchy.

5.0 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 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 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 the message expressions. 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; a limitation will be removed in future implementation.

It is the author's belief that an object oriented FORTRAN has more value to HEP than FORTRAN '90. Not that

```
   INCLUDE "appkit.inc"   ! Include AppKit
   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            ! line width of 2.0
   suspend = .false.      ! with cube rotating
   [Timer newTimer: @0.02D0
  +       target: self
  +       action: Selector("display\0")]
   newView_ = self ! Return, by convention, self
   @end

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

Fig. 10 Example of object oriented FORTRAN

```
class ThreeVec {
    float x;
    float y;
    float z;
public:
   ThreeVec( float px = 0.0, float py = 0.0, float pz = 0.0) {
            x = px; y = py; z= pz;}
    ThreeVec( ThreeVec& );
    inline float px(){ return x; }
    inline float py(){ return y; }
    inline float pz(){ return z; }
    inline void setx( float px ){ x = px; }
    inline void sety( float py ){ y = py; }
    inline void setz( float pz ){ z = pz; }
    inline void setVec( float px, float py, float pz )
       { x = px; y = py; z = pz; }
    inline float p2(){ return x*x + y*y + z*z; }
    inline float p(){ return sqrt( p2() ); }
    inline float pt2(){ return x*x + y*y; }
    inline float pt(){ return sqrt( pt2() ); }
    float phi();
    float theta();
    ThreeVec operator+( ThreeVec& );
    ThreeVec operator-( ThreeVec& );
    float operator*( ThreeVec& );
    friend ThreeVec operator*( float, ThreeVec& );
    friend ThreeVec operator*( ThreeVec&, float );
    friend ThreeVec operator-( ThreeVec& );
    void rotatePhi( float );
    void rotateTheta( float );
      /* etc ... */
}
```

Fig. 11 C++ header file to 3-Vector Class

FORTRAN '90 doesn't have some interesting features, as we will see in the next section, but that FORTRAN falls short of being an object oriented language.

6.0 Introduction To C++

There are a number of reasons for giving this short introduction to C++. Although C++ is the most widely used of all object oriented languages, its syntax is initially a bit difficult to understand. In this section, we hope to understand enough of it, by making analogies with Objective-C, to take away a bit of the fear of C++. Besides supporting OOP, C++ also supports abstract data types and operator overloading. These are very useful features but are sometimes confused with object oriented programming, so we need to understand them to see how they differ from OOP. C++ is a very powerful language but its syntax can be tedious as a result. Only certain aspects of the language can be covered here and the reader is referred to the ample supply of books on the language for further information (see Refs. [4] to [6]).

An example of a C++ class header file is shown in. Fig. 11. The class defined is that of a 3-vector as it might be used in HEP. The class definition consists of two parts; the class head with the keyword **`class`** followed by the class name, and the class body which is enclosed in braces ("{}"). This corresponds directly with the Objective-C interface section. In the class body, we have the definition of instance variables which are called data members in C++. The class **`ThreeVec`** has three data members; **`x`**, **`y`**, and **`z`**. They are followed by methods which are called member functions in C++. Unlike Objective-C, the member functions are not visible to objects of different classes unless they are explicitly declared to be visible. The keyword **`public:`** has this effect.

Instead of factory methods, C++ has special member functions called constructors which have more or less the same effect. A constructor member function is distinguished from the others by having the same name as the class. Thus, the function **`ThreeVec()`** is the constructor of the **`ThreeVec`** class. With these pointers, the header file shown in the example should start making a bit more sense.

In C++ each class is considered a new abstract data type which adds to the data types, such as **`float`** and **`int`**, that are built into the language. Thus, the **`ThreeVec`** class can be used just like the built-in data types can be used. The following code fragment illustrates allocating a **`ThreeVec`** and sending messages to it

```
int main()
{
    ThreeVec a;
    a.setVec( 1., 1., 1. );
    phi = a.phi();
}
```

The variable **`a`** is of abstract data type **`ThreeVec`** and its space is allocated at compile time in the same manner as data types **`float`** or **`int`**. The member functions are invoked with the syntax that looks like accessing a data member of a C struct. In the example, the member functions h as **`a.setVec()`**and **`a.phi()`** illustrate this. The authors of C++ were concerned about run-time efficiency. Thus, inline functions were included in the language to allow access to data members while preserving their encapsulation. The keyword **`inline`** declares this as is shown in Fig. 11 for the member function **`setVec`**. Inline functions do not invoke a function call, rather code is placed at the point the appear to be called. In addition, the implementation of inline functions can be placed directly in the body, as is shown for **`setVec`**. This, however, is recommended only if the implementation is short.

The implementation of the member function **`phi()`** in the example is not inline. Thus it is usually done in another file, sort of like the Objective-C implementation file. A possible implementation is shown in the example below.

```
float ThreeVec::phi()
{
    if ( x == 0.0 )
        if ( y >= 0.0 )
            return 0.5*M_PI;
        else
            return (-0.5*M_PI;
    float arctan = atan2( y, x );
    return arctan;
}
```

The double colon (“**`::`**”) separates the class name from the member function name, otherwise the function is declared as one would in the C language. Also note that one can declare the type of a variable anywhere before it is first used as was done with the variable **`arctan`**, i.e. they do not need to be at the head of the function. Otherwise, a C++ member function is pretty much like a C function. The inline declaration of new variables is one example of many improvements that C++ brings to the C language. There are many others, this paper is not the place to discuss them.

In C++, functions distinguish themselves from others with the same name not only by which class they are a member of, but also by what is called *signature*. The signature is the number and type of arguments and the return type. Thus the function **`int max(int,int)`** is distinct from **`float max(float,float)`**. This aspect of the language is known as *function name overloading*. There are two constructor member functions in the **`ThreeVec`** class. Which one is invoked is controlled by the signature used. In addition, constructors may be invoked with a variable number of arguments and default values can be applied to missing arguments. The following code fragment shows four ways that a **`ThreeVec`** constructor might be invoked

```
ThreeVec a;             // sets a to 0., 0., 0.
ThreeVec b(1.);         // sets b to 1., 0., 0.
ThreeVec c(1., 1., 1.); // sets c to unit vector
ThreeVec d = c; // invokes ThreeVec( ThreeVec& )
```

The last method is similar to allocating and initializing a built-in type (**`int i = 1;`**).

To complete the data abstraction aspect of the language, C++ also supports *operator overloading*. An operator is simply that character symbol in the language that causes an operation to be performed on one or more operands. For example, the plus character (“+”) is an operator to invoke the addition of quantities. Operator overloading means that for the abstract data types defined by the programmer, one can supply the meaning of the operators, i.e. overload the operator. For example, the + operator for the **`ThreeVec`** class might be defined as follows

```
ThreeVec ThreeVec::operator+( ThreeVec& b)
{
    ThreeVec sum;
    sum.x = x + b.x;
    sum.y = y + b.y;
    sum.z = z + b.z;
    return sum;
}
```

Reading the first line above can be confusing at first. Let’s decipher it. The function name is “**`operator+`**”. It is a member function of the class **`ThreeVec`** as seen by double colons ("**::**"). It takes one argument which is a reference to a instance of the **ThreeVec** class, or we could say that the argument is of type **ThreeVec**. The ampersand ("**&**") indicates the argument is passed by reference; without it the argument would be passed by value. It also returns a value of type **ThreeVec** which is indicated by the first "**ThreeVec**" on that line.

One might be immediately puzzled about what happened to the other operand for the operator "+". It turns out that the instance of the class, i.e. the object which was messaged, will be the other operand, and it will be the operand on the left of the + sign. With this knowledge, we can see how the implementation works. When we see the use of the data member **x**, for example, it means the data member of the object on the left of the + sign.

At this point it is worth noting that an object of a class has direct access to the instance variables of other objects of the same class. This would seem to violate the protection of the data. But main purpose of protecting the data is to hide its structure, i.e. where and how it is stored, so that should one change the structure, other classes don't need any changes. But if we change the structure of the data in a class, that class is going to be re-compiled, so there's no harm giving direct access to members of the same class.

There is another mechanism to implement operator overloading which must be used when the left operand is of a type that one does not have control over. For example, if one wants to multiply our **ThreeVec** by a **float**…

```
ThreeVec v1, v2, v3;
float a, b;
    ...
    v2 = a*v1;
    v3 = v1*b;
```

then the type **float** may be the left operand. The *friend* mechanism was invented to handle this situation. It allows functions that are not members of a class to have access to the protected data members of the class. In our **ThreeVec** example, the overloaded operator ***** was implemented this way. It was declared as a friend non member function in the class body. The implementation might look like

```
ThreeVec operator*( float& a, ThreeVec& b)
{
    ThreeVec mul;
    mul.x = a * b.x;
    mul.y = a * b.y;
    mul.z = a * b.z;
    return mul;
}
```

Note that only because this non-member function has been declared to be a friend of the **ThreeVec** class (cf. Fig. 11) can it have access to the protected data members of that class. Note also that operator overloading functions, the non member function requires one more argument then the member function. One can use either the member function or non-member function methods of implementing operator overloading, but not both for the same class. This is because the compiler would find an ambiguity, which it could not resolve.

In both examples of operator overloading, the function returned a result of type **ThreeVec**. This object is in fact only temporary. When the function is invoked with a code fragment like

```
d = a + b*c; // add and scale ThreeVec's
```

the object returned is the temporary the compiler must produce to form the result. The operator "=" for the class **ThreeVec** must also be overloaded by a member function of the form

```
ThreeVec& ThreeVec::operator=(const ThreeVec& a)
{
    x = a.x;
    y = a.y;
    z = a.z;
    return *this;
}
```

which does the assignment of **d** to the result. The compiler has this member function pre-defined for all abstract data types, but the programmer can over ride it by supplying his own. By the way, the **this** variable in C++ corresponds to the **self** variable in Objective-C, namely, its a reserved name for the object at hand.

The above description of adding two vectors may make it appear that there is a large overhead in the procedure. In fact, it is not. We are just looking at the step by step process a compiler must do to handle the addition of any two data types, even the built-in types. That is, operate on two quantities and store the result as a temporary, finally make the assignment to the variable on the left of the equal sign by copying it from the temporary space. As with any good compiler, a good C++ compiler will optimize these sequences of operations.

There are many member functions in our **ThreeVec** example. Many of them simply access the vector in different ways. For example, in HEP one frequently deals with the transverse momentum. Thus when the **ThreeVec** class

```
class FourMom : public ThreeVec {
    float energy;
public:
   FourMom(float px = 0.0, float py = 0.0, float pz = 0.0, float pe = 0.0)
      :(px, py, pz){ energy = pe; }
    friend FourMom operator+( FourMom&, FourMom& );
    friend FourMom operator-( FourMom&, FourMom& );
    friend float operator*( FourMom&, FourMom& );
    inline float e(){ return energy; }
    inline void sete( float pe ){ energy = pe; }
    inline float p2(){ return energy*energy - ThreeVec::p2(); }
    inline float mass(){ return sqrt( p2() ); }
    inline void setFourMom( float px, float py, float pz, float pe ){
        sete( pe );
        setx( px );
        sety( py );
        setz( pz );
    }
    virtual void boost( double, double, double );
    virtual void boostToCMOf( FourMom& );
    virtual void boostFromCMOf( FourMom& );
    virtual void print( int, char* form = "%8.4f" );
       // etc. ...
};
```

Fig. 12 FourMom class declaration

is used for momentum, we have the member functions **`float pt()`** and **`float pt2()`** to access the transverse and square of the transverse components of the vector. There are also member functions to rotate the vector in phi or theta. It may be tedious to implement all the variations that we may want to use. But by doing so, we will make the code that uses the **`ThreeVec`** class much easier and much more self describing. Most of these additional functions are declared **`inline`** thus there will be no lost in run time efficiency since they are not real function calls. If we did not include member functions like **`float pt()`**, then we would wind up implementing it explicitly each time we needed it. One could consider inline member functions of this type as something like a macro, but it is much safer then a macro because the compiler handles it.

Everything we've said about C++ in this section so far, is the data abstraction aspect of the language. Other languages that support data abstraction with operator overloading as well. Ada and FORTRAN '90 are examples. Each have their own syntax for accomplishing more or less the same features. C++ distinguishes itself from these languages in supporting object orientation as well as data abstraction. To understand the significance of this, consider the **`FourMom`** class shown in Fig. 12. This class is declared as a subclass of the **`ThreeVec`** class, thus it uses one of the important aspects of object oriented programming; *inheritance*. The keyword **`public`** in the class head is used to make public to the **`FourMom`** class the private members of the **`ThreeVec`** class. The other options, **`protected`** and **`private`** are too detailed to be discussed in this introduction to C++.

The **`FourMom`** class has one additional instance variable, or data member; the energy. Thus, the **`FourMom`** class differs from a vector with four components in that it is a Lorenz vector. It consists of a ordinary 3-vector component, which it inherits from the **`ThreeVec`** class and the energy component. Note how the member function **`p2()`** differs from an ordinary vector of four dimensions…

```
inline float p2() {
    return energy*energy - ThreeVec::p2(); }
```

The use of the function name **`p2()`** for both the **`ThreeVec`** and **`FourMom`** classes is making use of the polymorphism that is available to us. To force the call to the **`ThreeVec`** function **`p2()`**, we had to use the scope operator (double colon "**`::`**"). As with the **`ThreeVec`** class example, we've put into the **`FourMom`** class many auxiliary member functions to make the class easy to use. For example, if we want to boost a four momentum to the center of mass frame of

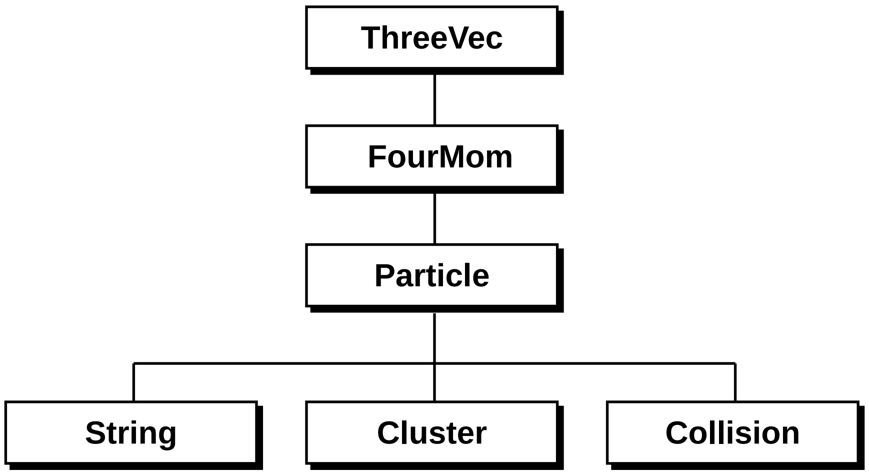

Fig. 13 Particle class hierarchy

another four momentum we have the member function **`void boostToCMOf()`** to do it.

Many of the member functions need to over-ride the inherited member functions of the same name, including the operator functions. While, if we want to get the transverse momentum of a four momentum vector, we have the inherited **`float pt()`** member function of the **`ThreeVec`** class available to us. This latter features brings out the difference between using the pure data abstraction and the object oriented approach. With data abstraction, a four momentum abstract data type might have a three vector component of it and thus look roughly like the same as objects with inheritance. However, with objects one inherits functions such as the transverse momentum functions, while with data abstraction alone, one would have to explicitly write code in the four momentum abstract data type to extract that information. Thus we see that data abstraction is quite distinct from object orientation even though for light weight objects, such as vectors, matrices, etc., they can be easily confused.

7.0 Particle Production Monte Carlo With Oop

In this section the use of object oriented techniques for particle production and decay simulation will be studied. OOP is well suited for such simulations for many reasons. The creation of particles in the real world corresponds to the creation of objects in the programming context. Particle properties correspond to data members of a class. When a particle undergoes a physics process, it's just like an object responding to a message. Encapsulation allows one to store the particle properties and program implementation of physics processes together.

We'll look at prototype code from the MC++ project that is under development at University of Lund and SLAC[7]. The key idea in the MC++ project is to formulate the event generation chain as generalized "particles" decaying into other generalized "particles". For example, at LEP one would start with an "e^+e^--collision" particle which decays into a "Z^0" particle. It in turn decays into perhaps a "$q\bar{q}$-dipole" particle which may decay into a "$q\bar{q}$-string" particle. Further down the chain one would eventually create the hadrons and leptons which will also decay until we reach stable particles at the end of the chain. Each such generalized particle will have a list of ways it can decay and each element of the list will have a branching ratio, a list of decay products, and a pointer to a class which implements the decay.

Let's first look at the class inheritance hierarchy in which the particle class is imbedded as shown in Fig. 13. The **`Particle`** class is derived from the **`FourMom`** class that was discussed in the previous section. Thus energy and momentum aspects of a particle will be fully taken care of by its superclasses. The instance variables or data members in the particle class are divided into two groups. The first group describes the generic features of the particle type, such as charge, mass, a pointer to its decay channels, etc. The second group describes a particular instance of a par-

```
class Particle:public FourMom
{
    float mass;
    int charge;    // charge is in units e/3
    int spin;      // in units 1/2
    boolean isStable;
    DecayList* decayTable;
    /* etc... */
    Particle* parent;
    ParticleList childList;
    float lifeTime;
    boolean hasDecayed;
public:
    Particle();    // constructor
    inline float charge() { return (charge/3.); }
    inline int icharge() { return charge; }
    inline float spin() { return (spin/2.); }
    virtual void decay();
    virtual ParticleList* decay();
    /* etc... */
}
```

Fig. 14   Particle class head and body

ticle such as its lifetime, creation values, a pointer to a list of its decay products, etc. Further specialization from the generic **`Particle`** class will be achieved by sub-classing. Thus one will handle the special attributes of strings, clusters, collisions, etc.

Figure 14. shows a code fragment of what the class definition might look like. Some interesting design choices are worth calling out. For example, the data member **`charge`** is stored as an integer with units of 1/3 the electron

To the theorists developing this code, these choices are very natural. However, to avoid confusion to an experimentalist who may want to access the charge, the inline function **`charge`** returns the charge of the particle in normal units.

Of particular note are the decay methods in the **`Particle`** class. Their implementations are shown in Fig. 15. and they are the heart of the code. The first method, **`void decay()`**, I call the theorist's method. It decays the particle into its children, then has each of the children decay. Since

```
void Particle::decay() {
   if ( isStable || hasdecayed ) return;
   DecayChannel channel
          = decayTable->selectChannel( random.flat);
   (channel->decayer)->decay(this,channel->products);
   hasDecayed = YES;

   Particle* child = childList.top();
   while( child = childList++ ) child->decay;
}
ParticleList* Particle::decay() {
   if ( isStable || hasdecayed ) return;
   DecayChannel channel
          = decayTable->selectChannel( random.flat);
   (channel->decayer)->decay(this,channel->products);
   hasDecayed = YES;
   return childList;
}
```

Fig. 15   Decay methods of Particle class

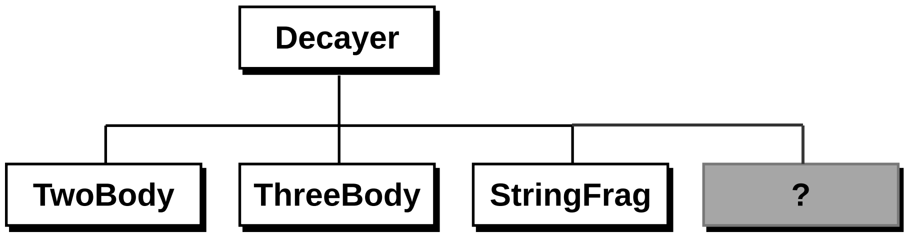

Fig. 16 Decayer class hierarchy

the children are particles, the decay method is invoked recursively. Since stable particles just return when the message is received (as seen by the first line of the code), recursion ends with stable particles. Theorists normally don't care about detector simulation, so this decay method suits their needs.

In the second line of code, a decay channel is created by randomly selecting one from the available channels in the decay table. The line of code says it about as well as my words. The decay channel object that one receives back from the decay table has a pointer to a decayer object. The third line of code invokes the decay member function of the decayer with two arguments; the particle being decayed and the list of decay products. The former is needed in order to access the four momentum and perhaps other properties. Obviously the latter is needed as well for similar reasons and to know what particles to create. The remaining lines of code in the **`decay()`** function recursively causes the decay products to decay.

The second decay method I call the experimentalist's method. This method decays the particle and returns the list of decay products. Thus, if the experimentalist is tracking a particle through a detector and finds that it needs to decay, he can invoke this decay method to get the decay products and then track them further through the detector. This method is identical to the theorist' method with the recursive decays removed.

A key component of the MC++ program is the decayer class and their hierarchy. A representative sample of them is shown in Fig. 16. The main aspect of the decayer, as shown in Fig. 17, is that its **`decay()`** method is presented with two arguments; the first gives it the particle that is to decay and the second a list of particles that will be produced as has already been described above. The **`Decayer`** class is just an abstract place holder which is meant to be sub-classed by a class that represent a real physics processes. The keyword **`virtual`** tells the compiler about this. That is, that the function is meant to be over-ridden by a subclass. For example, the **`TwoBody`** class would implement pure kinematics for most particle decays. A **`ThreeBody`** decayer might just use phase space. Rather then rewriting 3-body phase space in C++, a **`ThreeBody`** class might be implemented by presenting the relevant parameters to a FORTRAN function or subroutine.

```
class Decayer {
   char* genericName;
   char* name;
   int number;

public:
   Decayer( char* name, int number );
   virtual boolean isAllowed( Particle* parent, ParticleList* children );
   virtual int decay( Particle* parent, ParticleList* children);
   ~Decayer();
      // etc.
}
```

Fig. 17 Decayer class body

```
main()
{
    ParticleFactory factory; // invokes constructor
                             // which reads disk file
    Particle p;

    p = factory.getA( "e+e-collision" );
    p.decay(); // decay it.
    p.print(); // to see results
}              // that's all folks!
```

Fig. 18. A simple main program

This latter suggestion highlights the difference and similarities between the object oriented approach and a procedural one like FORTRAN. At some low level, the physics calculations are the same; there's no way around that. C++ through its abstract data types allows for operator overloading, thus the calculation can be written in a much easier to understand manner. That is, adding two vectors has the same programming notation as adding two integers; its just like one would add two vectors on the blackboard, with a shorthand notation. Some non-object oriented languages support data abstraction as well. Data abstraction also changes dramatically the way such abstract data types are created and stored. The use of inheritance makes C++ also an object oriented language which further simplifies the way the programming to be done.

One important aspect of the MC++ program will not be shown here. It is the mechanism by which the properties of the generalized particles are initialized. It is done via a **`ParticleFactory`** class of which only one instance exists in the program. It is a reference table, or could be considered an on-line Particle Data Book for the known particles, and a reference model for the less well known or generalized particles. The **`ParticleFactory`** class supports reading in a reference table from disk and interactive modification of the table, in order to study particular decay modes or models by the user of the program. It is also responsible for setting up the list of **`Decayer`** classes that make up part of the program. When all these classes are done, the main program to simulate one e^+e^- collision might be as simple as the code shown in Fig. 18. This program decays one e^+e^- collision but could easily be extended to do more. It is also a prototype for code that would be put into a detector simulation program.

We have just walked through the basics of a particle generation simulation program written in C++ using both its data abstraction and object oriented features. Of course, we only looked at a prototype program and we didn't show a lot of the details, but nevertheless it appears that a full production quality program would retain the simplicity and modularity we have seen. One should also notice the complete lack of dimensioned arrays, only minor use of conditional statements, do-loops are almost completely hidden by the use of lists. All these attributes of object oriented code greatly clean up the implementation, allowing the author to concentrate on implementation of the algorithms.

8.0 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, member functions, overloading, 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 object oriented programming. Generally, programs using these techniques are 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 far more easily achieved with the object oriented approach. Compared to traditional programming, object oriented code has fewer array declarations, thus minimizing the possibility of inadvertently exceeding array boundaries. Through the creation of objects, the system does the kind of bookkeeping that one would need to do by hand 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.

Object oriented programming is made much easier when one starts with a class library well suited to one's needs. For example, the NeXTstep class library is well suited for programming an application with a graphical user interface. High energy physics will need a class library for its specialized applications. It is hoped that out of projects such as Gismo[2], MC++[7], and CABS[8] such a class library will develop.

In an age where one frequently talks of a "software crisis", the object oriented programming approach seems to offer some real solutions. Programmers and scientists who use the object oriented techniques will find themselves to be much more productive.

References

[1] Brad J. Cox, Object Oriented Programming, Addison-Wesley, 1986.

[2] W.B. Atwood, T.H. Burnett, R. Cailliau, D.R. Myers, K.M. Storr, *Gismo: Application of OOP to HEP Detector Design, Simulation and Reconstruction*, Proc. Computing in High Energy Physics '91, Tsukuba, Mar. 11-15, 1991.

[3] W.B. Atwood, R. Blankenbecler, P. F. Kunz, B. Mours, A. Weir, G. Word, Proc. 8th Conf. on Computing in High Energy Physics, Santa Fe, NM, Apr 9-13, 1990, AIP Conference Proceedings ***209***, 320, (1990).

[4] Bjarne Stroustrup, The C++ Programming Language, 2nd Edition, Addison-Wesley (1991).

[5] Stanley Lippman, C++ Primer, 2nd Edition, Addison-Wesley (1991).

[6] Ira Pohl, C++ for C Programmers, Benjamin/Cummings (1989).

[7] Richard Blankenbecler and Leif Lönnblad, *Particle Production and Decays in an Object Oriented Formulation*, Private communication, to be published.

[8] Nobu Katayama, *Object Oriented Approach to B Reconstruction*, Proc. Computing in High Energy Physics '91, Tsukuba, Mar. 11-15, 1991.