SLAC-PUB-5251
June 1990
T/E

OBJECT ORIENTED PROGRAMMING FOR SIMULATION PROBLEMS IN PHYSICS*

RICHARD BLANKENBECLER

Stanford Linear Accelerator Center
Stanford University, Stanford, California 94309

ABSTRACT

In this paper, the use of object oriented programming techniques for physics simulations is discussed, as well as the type of problem for which OOP offers unique advantages. It is argued that object oriented programming can be used to efficiently treat a general class of simulation problems that are difficult, tedious or inefficient to program using standard languages. The methodology advocated here is to represent physical entities (in this case particles) and their dynamical properties by Objects in the sense of code. The advantages of using object oriented programming languages will be illustrated. Our detailed example should be considered as an programming experiment in OOP-an attempt to produce a working template for a particle production and decay Monte Carlo program. Objective-C from Stepstone Corporation, which combines OOP with the efficiency of the C language, will be used in the examples.

Alchemists of old used ciphers to protect their cabal
knowledge, nowadays we simply encode in FORTRAN.

Submitted to Particle World

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

Introduction and Motivation

The origin of the ideas developed here arose from work on the Reason project at SLAC.[1] This effort was aimed at developing an interactive data-analysis program package on a NeXT computer. This is a UNIX machine, making exclusive use of Objective-C in NextStep, its graphical interface.[2] For a discussion of the general language philosophy see the book by B. J. Cox[3] in which an overview (definition) of Objective-C is given. For an introduction to OOP concepts, read P.F. Kunzf

I will describe an experiment in the use of object oriented programming techniques for certain Monte Carlo simulation problems using Objective-C as a language of choice.[5] One characterization of simulation problems is whether or not their 'active' elements, the degrees of freedom, are fixed in number and characteristics,- or whether they vary, even randomly, during the simulation process. I shall argue that especially in this latter range, OOP can offer many unique advantages. The features of the language that may seem foreign to FORTRAN programmers will be highlighted during this discussion.

The analysis, formulation and organization described here is of general utility; it can be used in many types of applications. At this time, I have written simulation templates for several different physical problems in an effort to explore the advantages and disadvantages of OOP. These working templates will also serve as an example of OOP coding that can be 'fleshed' out later by others (those who are interested in the particular application, know what they are doing, what they want, and how the output is to be displayed.). One of them, a simulation code that describes particle production and decay, will be explicitly summarized here; it actually forms a part of a larger physics simulation package that contains, in addition, a quark-gluon cascade generator and a detector model.

We will now show that OOP code can be designed to mirror the ***physics*** of the problem rather than a ***straightjacket*** forced by the language. The new methodology, valid for general simulation problems, is one of the essential points; the decay Monte Carlo problem is a particular important application in particle physics.

Physics Analysis – Particle Data Classification

Particles are characterized by data and by actions such as decay, propagate, flip-spin, interact, etc. A particular elementary particle is described by a set of simple parameters such as mass, charge, other quantum numbers, space-time position, momentum-energy, helicity, etc. These parameters are required for all particles; thus we are lead to the concept of a minimal or generic 'particle' data set, or class. Unstable particles require more parameters; lifetimes and branching ratios for particular decay modes must be given. It is clear that both the stable and unstable particles actually produced in the decay chain form the fundamental ***degrees of freedom*** of the problem, and we shall concentrate on their description. The natural way to proceed is one that has been historically efficacious, namely, to classify the particles into similar groups and analyze their similarities and differences with a particular eye towards the choice of data classes; i.e. we first consider the electron, positron, photon, neutrino, piPlus, piZero, Kplus, . . ., Z_0, etc., as obvious class choices, and then group those with common features.

One very basic physical property, and hence a natural point of division, is whether a particle is classified as a boson or a fermion. Physics demands that we be able to treat these types differently. Thus Boson and Fermion will clearly form a. useful distinction in any classification scheme. The next question is what further subclasses should be defined under these two basic characteristics?

One natural group classification is that of SU(3). If we make the pseudoscalar octet a class, for example, we indeed find similarities, but also annoying differences. The major problem with this choice is that the decay modes of these mesons are very different. A unified description of the decays of the octet would be overly complicated. It is more natural to split the octet into particles with similar decay characteristics. After some thought (and trying several possibilities), I have chosen to classify only the particle pairs directly related by charge conjugation as immediate relatives. This will lead to a very simple description of the decay chain process, with the physics and the physical parameters occurring naturally.

Implementing This Physical Description

The next task is to find a natural way to express the physical description outlined above in terms of code. We would like the code, in generating a new particle in the chain, to produce a generic particle, add any needed general properties, and finally add the particular decay parameters specific to the produced particle type.

In addition, we also must add methods to allow a particle to perform an action, such as decay, in its own way. We may also want particles to propagate through a detector with or without a magnetic field, interact, lose energy, etc. Methods to simulate these physical actions will be needed eventually; however for clarity, the simple example described here will assume a perfect, and large, detector.

How should this programming problem be approached? Fortran does not seem to be a natural language to implement the above requirements. C is much better, particularly its use of pointers, recursion, and allowance for data structs, but as we shall see, object oriented programming provides a truly natural dialect for this problem.

Getting Along with Objects and their Actions

Three key concepts of object oriented programming were discussed in ref. 4 ; they are encapsulation, messaging, and inheritance. We shall make extensive use of all three of these features. Encapsulation will allow a simple treatment of the (different) physics of each particle type. Messaging and polymorphism will clarify the operation of the code. Inheritance will play a very important role in shortening, simplifying, and insuring consistency among the physics objects.

The main feature of OOP that we will use is the fact that memory management and bookkeeping are ***not*** the responsibility of the programmer but of the compiler and the run-time system. Indeed, recall that the methods that generate new instances of a class, termed ***factory methods,*** must set aside the proper amount of memory for each generated class object and set up access to the data for the action methods of the class; this is handled automatically and efficiently. As mentioned, to simplify the code, extensive use will be made of inheritance and polymorphism.

Decay Monte Carlo Hierarchy

The highest class, a subclass of Object, will be chosen to be ***Particle.*** In this class, all general and universal attributes shared by every particle will be introduced as instance variables. Action methods of universal applicability and general utility will also be defined; this will thus be the 'largest' class. Two natural subclasses of the Particle class are then introduced, ***Boson*** and *Fermion.*

Next, most of the physical particle types, the electron, pion, etc., can now be introduced as a subclass of one of these two classes. Using charge conjugation invariance, I have arbitrarily chosen the negatively charged member to be a subclass of the positively charged member; thus physical parameters will occur in only one spot in the code-with the relevant positively charged member of the pair. These pairs possess the same number and values of branching ratios but have charge conjugate decay modes.!

This completes the discussion of the particle hierarchy which is illustrated in Figure 1. It is something of a compromise between having many generations for reasons of coding simplicity, and few generations for reasons of efficiency.

Explicit Code Examples

Utilities/Tools: Before I present some sample Objective-C code, note that factory methods may create a number of new instances of any mixture of the defined classes; a powerful and flexible filing system is needed. This is provided by the ***List*** object predefined in NeXT Objective-C .[7] This class can create instances of itself, and add to, delete from, and send action messages to, a list of object id's. ***Lists*** will be an important tool in the OOP approach. Indeed, the decay chain is stored NOT as a linked list, but rather as a list linked by ***Lists.***

In reading the code outline to be given below, the reader should note the following characteristics of the OOP approach: the program operations mirror the physics of the process, particle objects respond to familiar physics-type commands given in English, and parameter values occur only once in the code with the particle that uses it (no conflicts). There are NO ARRAYS and no exceeding array limits by accident. Finally, the programmer does ***no*** explicit memory management ***nor*** other bookkeeping choirs; hence the writing, modification and extension of the code is considerably simplified.

First, we will insure that all particles can respond to every action message by utilizing inheritance (overridden where necessary). Inheritance will also be used to simplify the action methods of each class. For example, consider the action-method decay; a stable particle will ignore this message. An unstable one will proceed to -decay by first using its branching ratios to randomly decide which particular mode to invoke; it would then proceed to create instances of its decay products and to assign them momenta, etc. The second half of this procedure is the same for all classes; therefore a generic method, called begat: , will be defined to carry out this universal task, thus insuring consistency. An alternative and faster scheme for implementing this entire procedure will also be given.

In contrasting OOP with FORTRAN code, one should compare the cycles needed for Fortran pointer/array arithmetic and branching (written by the programmer) with the time required for an intrinsic OOP message process (~ 2 subroutine CALLS). At this stage, I have opted--to maximize clarity and maintainability instead of running efficiency; this issue can be addressed later in a myriad of obvious ways. The Monte Carlo program uses well-tested FORTRAN routines (e.g. phase space generators) whenever possible to minimize the coding task. Output is easily made available in several formats and will not be discussed explicitly.

The top 'generic' class Particle will have almost all the instance variables and action methods. This class is defined by giving a finite list of physical parameters such as charge, mass, energy, space-time position, etc.,[8] and a list of methods that act on these parameters. Action methods will be placed into the hierarchy at a physically appropriate level, although there is clearly some freedom here. Many utility methods that are not used in the main code but are useful for examination, tweaking parameters, debugging, etc. are included here for general availability.

The main difference between particle types, i.e. classes, is the number and meaning of their branching ratios. These will therefore be defined with the class that uses them; their precise meaning will then be manifest. The interface (.h) and implementation (.m) files of Particle, listed on the left and right respectively, will contain:

```
 #import <objc/Object.h>
 #import <objc/List.h>

 @interface  Particle:Object
 {
   char name[35];
   float charge;
   float  hypercharge;
   float mass;
   float E;     /* energy */
   float px;
   ...........;
   float ptot;
   float  lifespan;
   float  lifetime;
   BOOL  has-decayed;
   BOOL is-Stable;
   id  parent;
   id childList; /* progeny list */
   ............
 }
+ create:sender;
- (char *) name;
- (float) E;
- setE:(float) En;
  ................
- decay;
- familyTree;
- begat:(char *) decayMode;
- setParent:(char *) name;
- saveYourself;
- (BOOL) is-stable;
- (BOOL) has-decayed;
@end
```

```
#import "Particle.h"

@implementation  Particle
+ create:sender
 {
  self = [ super new ];
  parent = sender;
  childList = [ List new];
  lifespan = rand();
  return self;
 }
- (float) parameter // many
 {                  // utility
  return parameter; // methods
 }                  // such as
- setparameter:(float) qx
 {
  px = qx;          // these.
  return self;
 }
 // . . . . ..etc...............

- decay        // for stable
 {             // particles
  return self; // only.
 }
- familyTree;
 { // go thru object list & prt
 } // name & family affiliation.
- begat:(char *) decayMode
 {
   // . . ..details later...
 3
```

The variables are self-explanatory. The ratio (actual lifetime)/(average lifetime) is the lifespan and is set at creation time by a call to a random generator with exponentially distributed output. This sets the propagation distance for an unstable particle. It is not needed for the relatively few stable particles (but can be used to set their mean free path if needed). The begat: method will be discussed in detail later. To repeat, this is the largest class (the largest set of data and methods) that will be defined. Most additions and modifications to the code will-go here. They will then be inherited by every particle type, an efficient procedure.

All boson objects and its subclasses will inherit, in addition, the new instance variables and methods introduced by the Boson class. One of the methods here is ***make-anti;*** this method charge conjugates the quantum numbers of the Boson object that it acts upon. The Boson class contains the two files:

```
#import Particle.h

@interface  Boson:Particle
 { // no new variables.
 }
- make-anti;

- symmetrize: . . . ;   // NOT to be
                        // discussed
                        // further.
@end
```

```
#import "Boson.h"

@implementation  Boson

- make-anti
 {
  charge=-charge;
  hypercharge=-hypercharge;
  return self;
 3
```

The Fermion class is very similar, and will not be explicitly listed.

Creation: The specific particle classes are more interesting. The *piPlus*, for example, has a class definition containing:

```
 #import Boson.h

 @interface piPlus:Boson
 {                // need to add one
  float br-muon; // branching ratio
 }                // here.
+ create:sender;

- decay;
```

```
#import "piPlus.h"

@implementation piPlus


+ create:sender
 {
  self = [super create:sender];
   strcpy(name,"piPlus");
   charge = 1.0;
   hypercharge = 0.0;
   mass = 0.139;
   br-muon = 0.75; // muon-branch
```

```
                                              is-stable   = NO;
                                              has-decayed = NO;
                                              return self;
                                             3
                                            - decay
                                             { ........... later..
                                              return self;
                                             3
@end_ .
```

For the KPlus at least 2 branching ratios must be added, while the KZero needs 5, just for the dominant modes. To set the mass for resonance such as the rho, one could call a gaussian generator with mass $= 0.77 + 0.07 * rangaus()$. The code defining the piMinus class as a *subclass* of piPlus is now straightforward and short; no new variables are needed:

```
#import piPlus.h                      #import "piMinus.h"

@interface piMinus:piPlus             @implementation piMinus
 { // no new variables
 } // needed                          + create:sender
+ create:sender;                       {
                                        self = [super create:sender];
- decay;                                 strcpy(name,"piMinus");
                                         [ self make-anti I;
                                        return self;
                                       3
                                      - decay // . ..still later...
@end
```

Decaying and Begating: The decay and ***begat*** methods are perhaps more interesting; they are the guts of this version of the code and generate the decay chain. For the piPlus, the decay method in the class Particle must be overridden. Once the pion chooses to decay into a particular mode, the instances of the child-objects must be created, the piPlus childList must be updated, and finally, the children must be ordered to ***decay*** (note the form of the action message to the childList object given below). These tasks are accomplished by the methods:

```
     @implementation piPlus

     - decay
      {
       char  *mode;
   -- float rx;
               if (has-decayed || is-stable ) return self;

         rx = ranflat( between 0 and 1 );

            if (rx < br_muon )
                           mode = "muPlus+nuMuon";
            else
                           mode = "positron+nuElectron";

       [ self begat:mode ]; // create the mode list of particles.

       has-decayed = YES;

       [ childList makeObjectsPerform:@selector(decay);
                // sends decay message to each member of the List.
       return self;
      3
 Qend
```

The ***begat:*** method is utilized to create instances of all decay products. It can create an arbitrary number of new instances of objects from the full set of available classes. This method must take the (randomly) chosen decay mode, create the offspring, and add each of them to the childlist. Inside ***begat,*** we may also assign them their dynamic variables, such as momenta, or do it at a later stage. One form of this action method is:

```
   @implementation  Particle

   - begat:(char *) decayMode
    {
     char *ptype;              // the particle type to be created.
     char deCayMode[80];       // a disposable copy of the string.
     id childid;               // the id of the created object.

                    strcpy(deCayMode,decayMode);
          ptype = strtok(deCayMode,"+");       // parse out particle
     while ( ptype != NULL ) {                  // type to be created.

// Find factory method for class 'ptype' & then create an instance

       childid = [ [ self findClass:ptype ] create:sender 1;

                                              // and set its parent's id.

       [ childList insertObject:childid ];         // add to childlist.

          ptype = strtok(NULL,"+");                // NeXT!
     }
                                            // childList is now complete
             [self setKinematics];          // so set its kinematics.

     return self;
   }
 @end
```

An alternative form that eliminates both string parsing and the begat method is by direct messaging; in each decay method, one creates an instance of each particle class demanded by the chosen mode, sets the parent id, and forms the childList by invoking the single compound message

`[ childList addObject:[ piPlus create:self 1 1 ,`

where `self is` the parent id parameter passed to the create method. This method will increase the size of each decay method, but is straightforward and readable.

A perhaps faster method (no class searching) is to define as extern variables of type (id) PHOTON, ELECTRON, PIPLUS, etc. and to initialize them to the appropriate factory pointers by code such as

`PIPLUS = [ self findClass: "piPlus" ]`

before invoking the creation of the decay chain. To create a particle, one messages `PIPLUS` with the create method.

Driver: The driver for the program must create the initial state, set its variables to correspond to the physical situation and then generate the required number of events. The driver for generating a single Z_0 decay chain event would contain:

```
   #import <all that is needed>
   #import "Particle.h"

   void main ()
   {
    id initialState;

      initialState= [ zZero create:self ];
                                                  // create a ZO.
    [ initialState setParent:"Original" ];
                                                  // this is the original!
    [ initialState setE:92.7 ];
                                                  // set its energy in
    [ initialState setpx:0.0 ];
                                                  // its rest frame, and .
    [ ...................... ];
                                                  // repeat for py,pz,ptot.

// Now comes the step that starts the entire chaining process:
                          [ initialState decay ];

// The decay chain formed, and the Linked list generated
//              using the branching ratios and random coin tosses.

// Add some code to either exhibit the event by tracing the decay chain
// And/Or save it in a file to be examined by Reason.
                  [ initialState exposeYourself ];
                  [ initialState saveYourself ];
    exit(0);
   3
```

Several different strategies of event generation can be implemented by simply modifying the driver. If a loop is added to generate multiple events, the memory used by the decay chain should be freed after each event is generated.

Conclusions

At this point in the experiment, I have found that it is possible to write Monte Carlo code in a very CLEAR and UNDERSTANDABLE form; indeed, clarity of code still leaves freedom of choice. This type of code should therefore be easy to write, maintain, extend, and modify. The work required to add new classes, 'parameters, or new methods is minimal (by virtue of OOP ***inheritance).***

The particle production and decay simulation program described here forms the central part of a more useful trio of interrelated hierarchies that could be labeled: *field* (quark, gluon, string) $\rightarrow$(via hadronization) $\rightarrow$ ***particle*** (production and decay of named resonances and particles) $\leftrightarrow$ ***detector*** (simulation of tracking, secondaries, magnetic fields, energy loss, etc.). The arrows indicate the direction of message passing between the three separate families.

The output of this particular program can be a data set to be read and analyzed by the Reason data analysis program. Thus the operation of the program can be directly and simply visualized using the tools available in Reason. Alternatively, one can choose to print out, for study and debugging, a 'family tree' that exposes the decay chain for each event separately. Other presentations, graphical for example, are straightforward.

In summary, ***when bookkeeping and memory management DOMINATE the*** *coding* ***tusk,*** then OOP offers many advantages over conventional languages. Likewise, when maintainability and expandability are important, then object oriented programming techniques may well be the most logical choice. There is a wide range of interesting and even novel simulation problems that can be formulated naturally and easily using an object oriented programming techniques ; this general approach will become an ***essential*** programming tool, useful for a broad range of simulation problems in physics and engineering.

ACKNOWLEDGEMENTS

The author wishes to thank all the contributing members of the Reason project for putting up with a neophyte to group programming and for providing many helpful criticisms. A special thanks to Paul Kunz.

REFERENCES

1. 'The Reason Project', by W. B. Atwood, Richard Blankenbecler, Paul F. Kunz, Benoit Mours, A. Weir (SLAC), G. Word (Vanderbilt U.), SLAC–PUB-5242, April 1990. llpp. Contributed to the Rencontre de Moriond, Spring 1990, (by P. F. Kunz), and the Santa Fe conference, see ref. 4 .

2.-See, for example, Doug Clapp, 'The NeXT Bible,' Brady Books (1990), and Bruce F. Webster, 'The NeXT Book,' Addison-Wesley (1989).

3. Brad J. Cox, 'Object-Oriented Programming, An Evolutionary Approach,' Addison-Wesley (1986). Objective-C is available for many platforms.

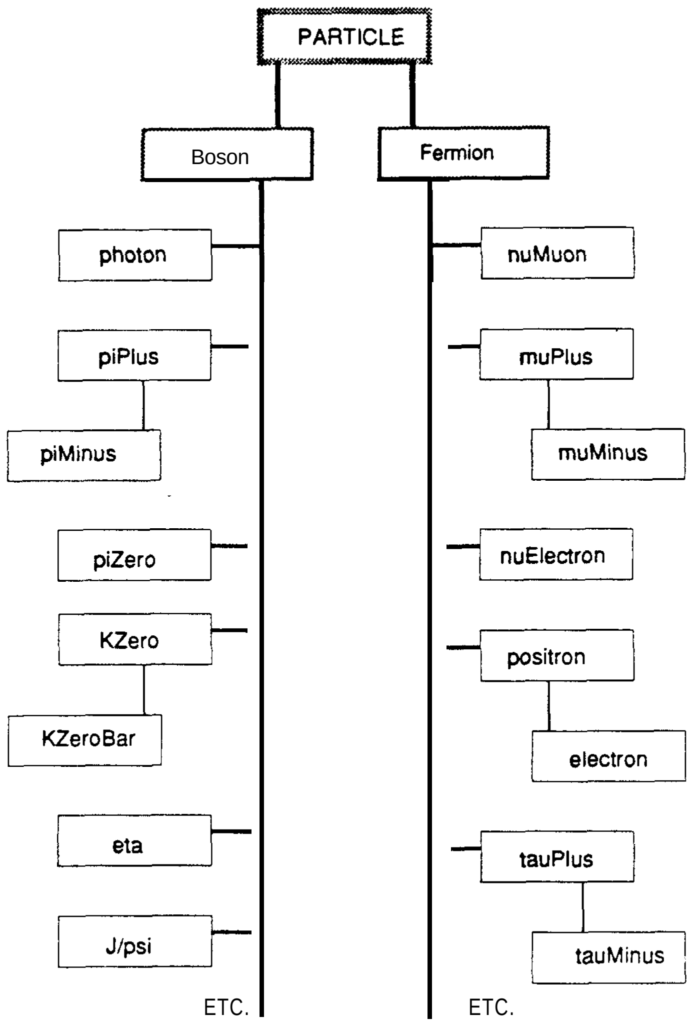
4. 'Object Oriented Programming', by Paul F. Kunz (SLAC), SLAC–PUB–5241, Apr 1990. 12pp. Invited paper given at 8th Conference on Computing in High Energy Physics, Santa Fe, NM, Apr 9-13, 1990.

5. Other efforts that I am aware of, for example K. Wilson, Ohio State, in Quantum Chemistry calculations, J. Dreitlein, Colorado, and independently P. LePage, Cornell, in field theory simulation, are using C++ . These applications do not seem to be using the features of object oriented programming that will be essential here. Indeed, it is perhaps not surprising to find that the optimum language depends upon the problem of interest.

6. For the neutral K (or D or B) systems, an interesting alternative class structure is to define a subclass of Boson called NeutralK, with subclasses KShort and KLong. This can be used to simplify the decay methods by tossing a coin at create time to determine CP and the relevant subclass.

7. List objects are defined in native Objective-C as well, but with different names.

8. Recall that a given class may have several ***instances*** of itself generated during execution. The data in each instance, i.e., each ***object*** , may be different; a variable in the data structure is termed an ***instance variable.***



MONTE CARLO -- **CLASS** HIERARCHY
FIGURE 1