SLAC–PUB–5648
LU TP 91–19
September 1991
(T/E)

PARTICLE PRODUCTION AND DECAYS IN AN OBJECT ORIENTED FORMULATION*

Richard Blankenbecler[a]

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

Leif Lönnblad[b]

Department of Theoretical Physics, University of Lund,
Sölvegatan 14A, S-223 62 Lund, Sweden

Abstract

The use of Object Oriented Programming languages and techniques for the development of Monte Carlo simulators for particle production and decay is discussed. For illustration, two specific "template" programs are presented written in Objective-C and in C++, respectively.

Submitted for publication.

---

* Work supported by Department of Energy contract DE-AC03-76SF00515.
[a] rbtheory@slacvm (bitnet) rzbth@thnextb.slac.stanford.edu (internet).
[b] thepll@seldc52 (bitnet) leif@thep.lu.se (internet).

Introduction

In this article, the use of Object Oriented Programming languages [1] and techniques for the development of particle production and decay Monte Carlo simulators will be discussed. Such programs are very important in the analysis of high energy physics data; the coupling of such a program with a detector simulator and a logical data analysis package allows one to examine in detail a particular model of the processes involved and then to see what events look like after detection and processing.

One important feature of decay Monte Carlo simulations is that the number of degrees of freedom, in this case the number and type of final particles, necessarily varies, indeed randomly, from event to event. It is in the ease of handling this feature that OOP provides one of its most important advantages over procedural languages such as Fortran. It has several other useful features as we shall see.

We will first describe the general strategy for applying OOP to this process and then give two specific "template" programs called McOOP and MC++, written in Objective-C and in C++, respectively. The two programs will differ somewhat in concept and goals, and considerably in details. Both, however, will illustrate an important feature of OOP in which the code itself directly mirrors the physics of the process. This implies that a new methodology can be extended and developed for general physics simulation problems; this will be discussed elsewhere.

Particles are characterized by variables and by actions; both are generally unique to the particle type being described. A particular particle has general parameters such as mass, charge, which are identical for all members of its "class" and its own individual value of momentum, energy, position, helicity, etc. Unstable particles have additional parameters such as branching ratios, decay modes, decay matrix elements, etc. Keeping track of all these properties efficiently is generally a difficult design task for the programmer.

We shall see that OOP can be used to greatly simplify this task yet will produce efficient and flexible code.

The general philosophy will be to let the OOP operating system handle memory management (it can undoubtedly do this more efficiently than we can anyway) so that storing and retrieving data from common blocks, and having to write the logic that handles the index to the data, will not have to be done explicitly by the programmer. Other useful features of OOP are that the data for a given particle is encapsulated into a convenient entity called an "object" and thus the structure of the code allows easy and straightforward maintenance and extension. As a consequence, there are NO dynamic data arrays in the code, and no exceeding array limits by accident during execution.

The three basic concepts of object oriented programming relevant here were discussed in the earlier article by Kunz [1]. They are encapsulation, messaging, and inheritance. We shall make extensive use of all of these characteristics. Encapsulation will allow a simple and direct treatment of the (different) physics of each particle type; messaging and its associated polymorphism will clarify the operation of the code; inheritance, defined by a class hierarchy or family tree, will allow more concise code, guarantee consistency between particle types and greatly simplify the task of adding additional particles, new characteristics or new reactions to the program. All of these concepts will allow us to develop code that utilizes one of the main advantages of OOP, namely, that memory management and bookkeeping are the responsibility of the compiler and the run time system, NOT the programmer.

As the decay process proceeds, many particle objects are created and stored in memory by the system. In order to keep track of these particles, a powerful and efficient filing system is needed. This is provided by a List class; this type of object stores an ordered list of pointers to the particle objects that have been created. List objects can add and delete objects from themselves and send messages to all members that are contained therein. The List class plays an important role in OOP, especially in simulation problems.

The OOP code that we have developed has the following characteristics: the program operations mirror in a direct way the physics that is going on, the basic entities of the physical process—particles—correspond directly to the objects of the particle class in the code, which, e.g., respond to familiar physics-related commands such as decay, propagate, setEnergy, chargeConjugate, etc.

At an early stage in the implementation, one must make general design and style decisions and fix the structure of the inheritance hierarchy. Many variations of the choices given below are possible; we have implemented several which work well. One essential difference between particle types, i.e., classes, is the number of branching ratios and their associated decay modes. Once a mode is chosen, the code must produce an instance of each particle class in the mode and assign energy/momentum to each of them according to a specific matrix element. This procedure is treated very differently in the two template programs which will be discussed next.

McOOP

In order to implement this Objective-C version of the Monte Carlo, we must first decide what the most fundamental class is to be, i.e., what it should contain as instance variables and as methods. The concept of a "generic" particle is clear; it will be realized as the Particle class. Its variables are mass, four-momenta, charge, hypercharge, lifetime, its name, and if it is part of a branching decay process, its parent and a list of its decay products, or children. For a stable particle, this last list will be empty, of course. The action methods of a generic particle should include not only the factory method that produces particular instances of the class (the "objects") but also general utility methods that set and return values of all instance variables, that produces a "family tree" to expose the particular branching of this event, and a "decay" method that allows the particle to decay (if it can) and produces instances of the children, sets their momenta stochastically via some decay matrix element, and then instructs each child to decay in turn.

It is also natural, in order to treat the general case, to have Fermion and Boson classes as immediate subclasses of the Particle class. These classes are quite simple unless one is symmetrizing matrix elements, etc. Under these classes, the familiar particle types such as photon, positron, pion, kaon, etc., will be subclassed.

Before returning to specific particle classes, let us turn to some design questions. Where should the particle property data be stored? One can place the specific data that defines a particle type together with its branching ratios and modes with the individual class. That is the method utilized in the code samples given below; it has the advantage of being very simple to understand—the code reads as it does—but is not flexible. If one wants to modify a mode by adding a particle in the final state, for example, the code for that class must be changed and then recompiled. A more general method is to have a central data cache that contains branching fractions and lists of particular decay modes accessible from each particle class as it is needed; each mode is stored as a List object containing the appropriate class pointers to create the given final state particles. Each class then stores pointers to its own data as instance variables. This central data cache can be modified without changing the code of a class. This technique has many advantages for production code and will be implemented in the next example to be discussed, MC++, although it has been implemented in McOOP as well.

In the particular McOOP code presented below, we have also chosen, rather arbitrarily, to implement the negatively charged member of a charge conjugate pair (such as piPlus and piMinus) as a subclass of the positive member. Thus the data, such as mass, charge, branching ratios, etc., can be stored in only one place in the code—thus avoiding data conflicts. All that is required is a general method that performs charge conjugation. Other choices are possible.

Explicit Code

The McOOP class hierarchy is shown in fig. 1. The interface (header) and implementation files for the generic Particle class is shown in figs. 2 and 3. A few selected subclasses will be discussed as examples of coding philosophy and style.

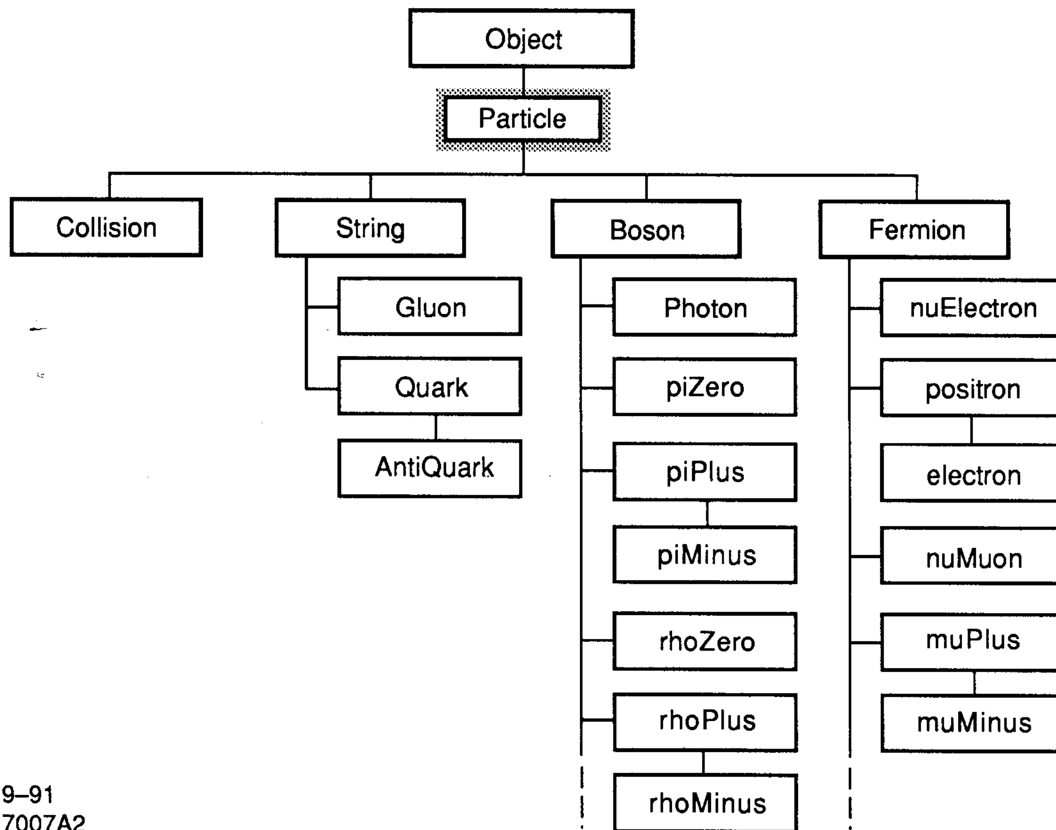

FIGURE 1: The class hierarchy of the McOOP program.

The Boson class does not contain any instance variables to add to those of the Particle class, but it does add methods such as chargeConjugate, which appropriately modifies the instance variables of a class object; this directly allows the instantiation (creation) of a piMinus object, for example, from a piPlus object.

```
@interface Particle: Object  /* Particle.h file */
{  char name[10];               /* instance variables */
   int type;
   float mass;
   int charge;
   int hypercharge;
   int spin;
   boolean isStable;
   boolean hasDecayed;
   id parent;
   id childList;             /* list of decay products */
   float E;                  /* energy */
   float px;                 /* momentum */
   /* helicity, etc......... */ }
   /* declaration of methods for the particle class: */
 + create:sender;    //the creation method with parent's id
 - (char *) name;
 - (float) energy;
 - setEnergy:(float) E;
 - decay;
   /* etc... */
@end
```

FIGURE 2: The header file for the **Particle** class.

```
@implementation Particle                /* Particle.m file    */
 + create:sender                        // the creation method
   { self = [ super new ];
      childList = [ List new];
      parent = sender;
      lifeSpan = rand(); return self; }
 - (char *) name
   {  return name; }
 - (float) E
   {  return E;   }
 - setEnergy:(float) energy
   {  E = energy;   return self; }
 - decay
   /* etc... */
@end
```

FIGURE 3: The implementation file for the **Particle** class.

```
#import  Particle.h                    // piPlus.h  file
#import  Boson.h

@interface  piPlus:Particle
{    float br_muon;}                   // added instance variable
+ create:sender;                      // create method with parent's id
 - decay;
@end

#import  piPlus.h                       // piPlus.m  file
@implementation  piPlus
+ create:sender                         // the creation method
   { self = [ super create:sender ];
      strcpy(name,"piPlus");
      charge = 1.0;               hypercharge = 0.0;
      mass = 0.139;               br_muon = 0.75;
      isStable = NO;              hasDecayed = NO;
      return self; }
 - decay
   {  if (has_decayed || is_stable) return self;
      if (ranflat(&idum) < br_muon) {
         [ childList addObject:[muPlus create:self]];
         [ childList addObject:[nuMuon create:self]];}
      else {
         [ childList addObject:[positron    create:self]];
         [ childList addObject:[nuElectron create:self]];}
      has_decayed = YES;
      [ self setKinematics]; //set kinematics, helicity of decay products
      [ childList makeObjectsPerform:@selector(decay)];
   return self; }
@end
```

FIGURE 4: The header and implementation file for the **piPlus** class.

The specific particle classes are more interesting. The piPlus class has class definitions containing additional instance variables and a customized implementation of certain generic methods in the Particle class, especially the create and decay methods. This is illustrated in fig. 4. The resulting create method for the piMinus class is given in fig. 5.

```
#import piPlus.h                    // piMinus.m file
#import piMinus.h
@implementation piMinus
+ create:sender
   {  self = [ super create:sender ];
      [ self chargeConjugate ];
      strcpy(name,"piMinus");
      return self; }
@end
```

FIGURE 5: The create:parent method for the **piMinus** class.

In the decay method, the call to the random number generator, ranflat, determines the decay mode, the setKinematics method in the present code calls a FORTRAN subroutine that assigns the final state momentum (an illustration of re-using old code!). The final line is the recursive message that keeps the generator going; it sends the childList the message "makeObjectsPerform" which sends the message "decay" to each decay product in turn.

The decay method in this code is customized for each particle class since each has its own individual branching ratios and decays. For example, the String class, a quark-antiquark pair plus string, decays into itself at high masses, and into ordinary hadrons as its mass decreases. Thus its decay method is quite analagous to the decay method for physical particles; the matrix element is, of course, very different.

In the alternative data cache method discussed earlier, each instance of a class would contain pointers to an array of branching fractions and to an array of decayModeList objects, each of which contains a list of factories that create instances of each decay particle in the final state. There is then no need for a customized decay method for each particle class.

MC++

We feel that this particular "McOOP" example shows clearly the benefits of OOP when writing a particle generation/decay program. It is, however, very far from a finished "product" from a user's point of view. The user would typically want to be able to easily change branching fractions, switch on or off, add or remove decay channels and also, e.g., in B-decay where far from all branching fractions are known, switch between different models of the decay, i.e., different decay methods. To do this in the explicit McOOP example given above, the user would have to go into the code to do changes and then recompile, which is not a satisfactory situation.

Therefore, in the next example, the goal will not be to write *an* event generator, but rather to try to define the structure of a general event generator using OOP. Or in other words, to take an OOP language, in this case, C++ [2] (see also Appendix A), and customize it to create a "dialect" specially suited for event generators, i.e., a *toolbox* or a *class library* for particle decay.

A first attempt along these lines is a "template" program which we call MC++. As it stands, it can only handle ordinary particle decays, but it is written to be expandable to a complete event generator. In designing the code, we defined a number of goals. These can be divided into two groups: requirements from the "user's" point of view and requirements from the "model builder's" point of view. The first group contained things like:

1. The user should easily be able to add, remove and change decay channels, branching fractions and decay methods of particles as well as adjust their mass and lifetimes, etc.
2. The user should just as easily be able to change between different models, e.g., partonic showers and fragmentation.
3. The code should be portable between different machines.
4. The code should be easily interfaced to a GUI.

From the model builder's point of view, we required that:

1. The code should be modular, so that a model just describing a small part of the event generating chain is easily added.
2. The code should provide utilities to facilitate development of new "modules."

The result is presented schematically in fig. 6 which describes the class hierarchy of MC++.

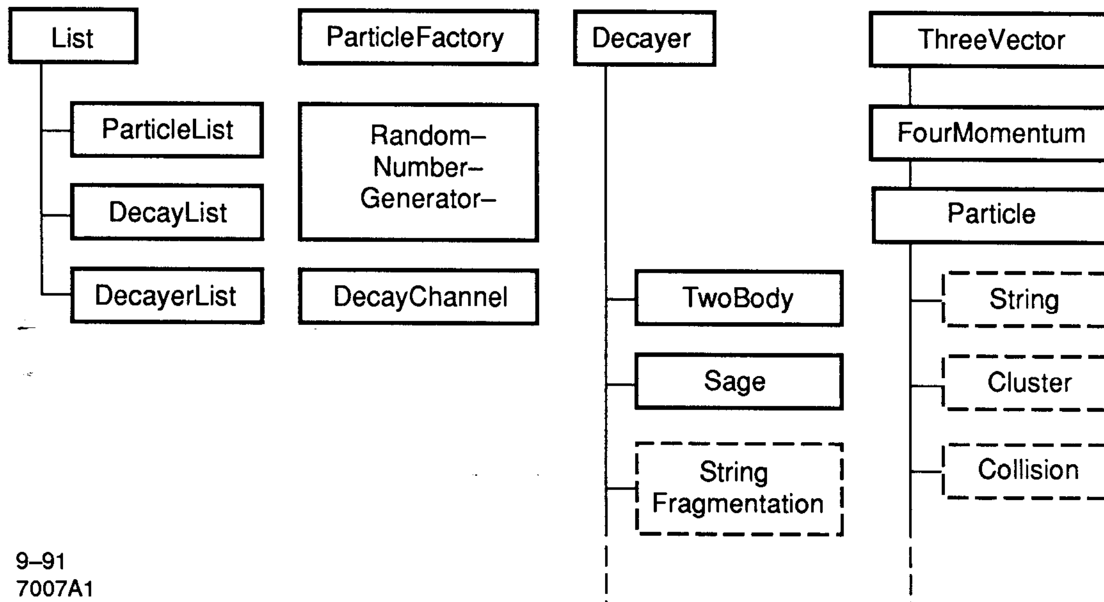

FIGURE 6: The class hierarchy of the MC++ program.

The first thing to note is that since the decay channels and decay methods, etc., are not fixed in the code, there is no longer any great difference between, e.g., a π^+ and a K^-. Hence there is no reason to have the "deep" hierarchy of McOOP where each particle is a class of its own. Instead they are all members of the same class called "`Particle`." Also `String`, `Cluster` and `Collision` are classes which are derived from `Particle`. In this way, the complete chain in the event generation can be described in terms of `Particles` decaying into `Particles`. For example, we can have the (generalized) `Particle` "`e+e- Collision`" which is an object of the `Collision` class, decaying into a *"Z0"* which decays into a `String`, which, in turn, decays into `Particles`.

```
class Particle: public FourMomentum {

/* instance variables describing the particle type: */

        char name[10];
        int type;
        float mass;
        int charge;          // charge is in units e/3
        int spin;            // unit 1/2
        boolean isStable;

    DecayList* decayTable;

/* etc...
    instance variables describing this instance of the particle: */

        ThreeVector creationPoint;
        ThreeVector decayPoint;

        ParticleList childList;
        Particle* parent;

        boolean hasDecayed;

/ * etc...
    declaration of methods for the particle class: */

public:

        Particle();                 //the creation method
        virtual void decay();
        virtual void addChild(Particle* child);
        float charge();
        float spin();
        char* name();
        int type();
        void boost(double bx, double by, double bz);
        rotatePhi(float phi);
        rotateTheta(float theta);
        void setDecayPoint(ThreeVector point);
        void setCreationPoint(ThreeVector point);
        Particle* copy();
        void print();
        ~Particle();                // the destruction method

/ * etc... */

 };
```

FIGURE 7: The definition of the `Particle` class.

The main structure of the code is contained in the definition of the **Particle** class and we will therefore begin by describing its features.

The **Particle** Class

The **Particle** class is derived from the **FourMomentum** class in an attempt to make life easier when developing new methods. The **FourMomentum** class has, besides its variables p_x, p_y, p_z and E, methods for boosting, rotating, etc., which are inherited by the **Particle** class. In this way, a **Particle** knows automatically how to be rotated and boosted [3].

The instance variables of the **Particle** class (see fig. 7) can be divided into two groups. One group describes the generic features of the particle type: name, charge, mass, decay channels, etc.; and one group describes the particular instance of a particle: fractional lifetime, creation point, a list of its children, etc. (its four-momentum is inherited). The decay channels are described by a pointer to a **DecayList** object, which is simply a list of objects of the **DecayChannel** class. Each **DecayChannel** contains a branching fraction, a pointer to a **ParticleList** containing a list of the decay products and a pointer to a **Decayer** object.

When a **Particle** receives a message to decay (see fig. 8), it will tell the **DecayList** to select a **DecayChannel** according to the branching fractions. Then it will send a message to the **Decayer** pointed to by the **DecayChannel** to perform the decay, giving the pointers to itself and to the list of decay **products**. Finally, the particle sends messages to its children to decay as well.

In this way, the whole event generation is defined. The procedure would be to first define the different particles, their masses, the way they decay, etc., and then to generate, e.g., a LEP event you would simply create the generalized particle **e⁺e⁻Collision** giving it a mass of 91 GeV and tell it to decay.

```
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();

   }
```

FIGURE 8: The definition of the **decay** method in the **Particle** class.

The problem here, as compared with the McOOP example, is that since the particles' properties are not defined in the code, it is difficult to let the operating system completely administer the creation of new particles in the program. Instead, we have defined a class called **ParticleFactory** to do this.

The ParticleFactory Class

The **ParticleFactory** class keeps a list of "templates" of all available particles and all available decay methods, and it is by passing messages to a **ParticleFactory** object that the event generating chain is defined. The **Particle** class itself is not equipped with any methods to change the generic features of a particle type such as decay channels and methods; instead, this is handled completely by the **ParticleFactory** through methods like the ones listed in fig. 9. This encapsulation is to ensure that everything is set up consistently so that, e.g., a π^0 is not set up to decay according to string fragmentation.

The **ParticleFactory** is also equipped with methods to read and write information to and from files. In particular, it should be able to use machine-readable and user-modifiable files with standard particle properties, decay channels and branching fractions. The procedure would be to have a file with

```
class ParticleFactory{

    DecayerList decayers;
    ParticleList particles;

public:

    ParticleFactory(char* fileName = "pdg.ptab");
    readFromFile(char*);
    saveToFile(char*);

    void addParticle(char* particleName, int particleNumber);
    void setParticleMass(int particleNumber, float mass);
    void setParticleMass(char* particleName, float mass);
    void setParticleSpin(int particleNumber, int spin);
    int addDecayChannel(int particleNumber, float branchingFraction, \
    int decayerNumber, int numberOfChildren, ... );
    void removeDecayChannel(int particleNumber, int channelNumber)
    void setBranchingFraction(int particleNumber, int channelNumber, \
                                  float branchingFraction);

    void addDecayer(char* genericName, int number, char* name);
    void setDecayerOptions(int number, char* options);

    Particle* getA(char* particleName);
    Particle* getA(int particleNumber);

/* etc... */

    ~ParticleFactory;

    };
```

FIGURE 9: The definition of the **ParticleFactory** class.

a standard set of additions and modifications and then a set of files with changes corresponding to different experiments.

When the event generating chain is completely set up, the **ParticleFactory** object can be made to produce copies of its template particles in the same way as the operating system does in the McOOP example. In this way, the

ParticleFactory can be thought of as an extension of the operating system, and we actually prefer to view the whole structure of MC++ as an extension of the programming language C++, customized for event generation in particle physics.

The **Decayer** Class

The generic **Decayer** class is very simple. As seen in fig. 10, it has only three instance variables: a generic name for the type of decayer and a name and number for a particular instance of a decayer. Besides the creation and destruction methods, it has only two methods: **decay** and **isAllowed** which are virtual in the sense that for this generic class they do nothing except declares the method names to be inherited by the derived decayer classes.

```
class Decayer{

    char genericName[10];
    char name[10];
    int number;

public:

    Decayer(char* name, int number);
    virtual boolean isAllowed(Particle* parent, ParticleList* children);
    virtual int decay(Particle* parent, ParticleList* children);
    ~Decayer();

    };
```

FIGURE 10: The definition of the **Decayer** class.

The procedure for developing new decay methods would then be to write a new class derived from the generic **Decayer** class, modifying the **decay** and **isAllowed** methods. A very simple example, the **TwoBody** decayer which simply decays a **Particle** into two according to phase space, is shown in fig. 11. The **isAllowed** method simply checks that the list of decay products

```
class TwoBody: public Decayer{
public:
TwoBody(char* name, int number);
boolean isAllowed(Particle* parent, ParticleList* children);
int decay(Particle* parent, ParticleList* children);
 TwoBody();
    };
boolean TwoBody::isAllowed(Particle* parent, ParticleList* children){
    return children->count() == 2 ;
 }
int TwoBody::decay(Particle* parent, ParticleList* children);
    Particle* child1 = factory->getA(children[1]);
    Particle* child2 = factory->getA(children[2]);
    if( child1->mass() + child2->mass() > parent->mass() ) {
    delete child1;
    delete child2;
    return 0;
   }
    float mass12 = (child1->mass())*(child1->mass());
    float mass22 = (child2->mass())*(child2->mass());
    float energy = parent->mass();
    float energy1 = (energy*energy - mass22 + mass12)/(2*energy);
    float pz = sqrt(energy1*energy1 - mass12);
    *child1 = FourMomentum(0.0, 0.0, pz, energy1);
    *child2 = FourMomentum(0.0, 0.0, -pz, energy - energy1);
    float theta = acos(random.flat(-1.0, 2.0)); // random.flat(-1.0,2.0)
    child1->rotateTheta(theta);                 // gives a random number
    child2->rotateTheta(theta);                 // between -1 and 1.
    float phi = random.flat(2*PI);              //  random.flat(2*PI)
    child1->rotatePhi(phi);                     // gives a random number
    child2->rotatePhi(phi);                     // between 0 and 2 pi.
    child1->boostFromCMOf(parent);
    child2->boostFromCMOf(parent);
    child1->setParent(parent);
    child2->setParent(parent);
    parent->addChild(child1);
    parent->addChild(child2);
    return 1;
 }
```

FIGURE 11: The header and implementation files for the **TwoBody** decayer class.

consists of exactly two particles. The **decay** method performs the decay and we like to think that it is written in such a transparent manner that even readers with very poor knowledge of C++ will need no further comments.

Finally, in fig. 12 there is an example of how to use MC++ to produce 10 K^+ decays and to print them.

```
main(){

    ParticleFactory  factory("fileOfParticleProperties");
    factory.readFromFile("MyFavouriteChanges");

    Particle* p;
    for (int i=0; i<10; i++){
        p = factory.getA("K+", 0.0, 0.0, 10.0)
        p->decay();
        p->print();
    }
}
```

FIGURE 12: Example of a main program using the MC++ structure.

As it stands, MC++ is, of course, just as far from a finished product as McOOP. A project is, however, planned by the theory departments at Lund and SLAC, to develop the MC++ idea into a product which (hopefully) may set the standards for the next generation of event generators. The form and time scale of the project is not fixed, and we would like to encourage anyone who is interested to contact us so that the base for this project can be broadened.

Conclusions

We have found that it is possible to write OOP code simulating particle production and decay and utilizing Monte Carlo methods in a very clear and understandable form; the code allows easy and straightforward extensions— the addition of new particle classes and/or decay reactions is simple, straightforward and without hidden pitfalls.

In summary, when bookkeeping and memory management DOMINATE the design of a simulation code, OOP offers many advantages over conventional procedural languages. When code must be flexible, maintainable and extendable, again, OOP should be considered. The slight pain experienced when learning a new computer language and the mastering of the quite different OOP design philosophy will be well worthwhile in the long term.

Appendix A: Objective-C vs C++

Both Objective-C and C++ are object oriented extensions of the C programming language. They differ, however, very much both in notation and in "philosophy." We can only alert the reader to be on guard in this brief appendix.

The notational differences are perhaps the most obvious. In C++, classes are implemented as an extension of the structures in C, which besides containing member variables now also may contain member functions which are accessed in the same way. In Objective-C the classes are implemented as a completely new feature, and the declaration of classes differs quite radically as can be seen in the code examples above.

The notational differences when sending messages to objects are also large. If "**p**" denotes a (pointer to a) **Particle**, to decay it in McOOP one would write:

```
[p decay];
```

While in MC++ the notation would be:

```
p->decay();
```

Which of these notations one prefers is, of course, a matter of taste. A more important difference is that C++ has what is called "strong typing." This means, e.g., that it has to be decided at compile time what kind of object a pointer refers to. So that in the example above, "**p**" must be defined as "a pointer to a **Particle** object." In Objective-C, however, **p** can be defined as a pointer to any object which makes it easier to construct general classes. As an example, the List class in Objective-C may contain any object, whereas in C++ there has to be a separate List class for each kind of object (cf. **ParticleList** and **DecayList** in MC++). This is accomplished in Objective-C because all classes are derived from the common (abstract) base class called "**object.**" The use of virtual functions [3] in C++ can, in fact, accomplish something

similar. Note, e.g., that the **ParticleList** class may also contain objects of classes derived from the **Particle** class.

The advantage of strong typing instead lies in the run-time speed. In Objective-C it first has to be checked if **p** really is an object of the particle class before the decay method is invoked and this, of course, takes time.

Another feature of C++, not present in Objective-C, is called operator overloading. This enables you to redefine any operator to perform different tasks depending on its arguments. As an example, in MC++ the operator "*" is redefined so that if **p1** and **p2** are two objects of the class **FourMomentum**, **p1 * p2** gives the scalar product of the two, etc.

For more details about the two languages, we suggest refs. [3] and [4]. The reader then can make his own comparisons and evaluations.

References

[1] Paul Kunz, "Object Oriented Programming," SLAC–PUB–5241, April 1990. 12pp. Invited paper given at 8th Conference on Computing in High Energy Physics, Santa Fe, NM, Apr 9–13, 1990.

[2] Our experience with the different versions of the program written thus far is the basis for this claim. A test version of the program that can symmetrize the decay chain back through two generations to explore interference effects was straightforward to implement by adding new methods to the Boson class.

[3] See, for example, B. Stroustrup, '"The C++ Programming Language," Addison–Wesley 1987, ISBN 0–201–12078–X.

[4] See, for example, Brad J. Cox, "Object–Oriented Programming, An Evolutionary Approach," Addison–Wesley (1986). Objective-C is available for many platforms.