

MONTE CARLO AND DETECTOR SIMULATION IN OOP*

W. B. Atwood, R. Blankenbecler, and P. Kunz
Stanford Linear Accelerator Center
Stanford University, Stanford, CA 94309, USA

T. Burnett[†] and K. M. Storr
CERN, ECP Division, CH-1211 Geneva 23, Switzerland

ABSTRACT

Object-Oriented Programming techniques are explored with an eye towards applications in High Energy Physics codes. Two prototype examples are given: McOOP (a particle Monte Carlo generator) and GISMO (a detector simulation/analysis package).

1. Introduction

High Energy Physics (HEP) as a field came into being shortly after World War II. Accelerators replaced cosmic rays and computers were soon required both to gather data as well as assist in the analysis. The FORTRAN language was created to facilitate the translation of algorithms and formulas into machine executable code (FORTRAN stands for FORMula TRANslation language!) and was one of the first "higher-level" languages requiring a compilation step to produce machine code. HEP quickly adopted FORTRAN and today FORTRAN dominates our field. We have come to rely on many FORTRAN programs and packages, from histogrammers to memory managers, in achieving our research goals. Furthermore, entire systems of code such as GEANT [1] for the simulation of particle detectors and the LUND Monte Carlo [2] for gen-

erating events have become accepted starting points in the HEP code world.

FORTRAN code is referred to in computer science circles as a procedural language. The label is obvious and descriptive. Our programs for the most part are repetitively executed algorithms architected along client server lines with a hierarchy of drivers, subdrivers, ... and so on. Our style has changed very little, aside from trying to superimpose a few modern concepts such as memory management on top of FORTRAN.

Meanwhile the rest of the world continued to evolve, rather than refine and embellish. The UNIX operating system and its underlying language C resulted from efforts in the academic and engineering environments to achieve greater flexibility and open access to their minicomputers. The complexities of the graphical user interface employed on most small computers required further evolution. Object-Oriented Programming (OOP) techniques proved themselves equal to this challenge and are now used extensively. It's worthwhile to note that OOP

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

† Permanent address: University of Washington, Seattle, WA 98195, USA.

is driven by industrial needs, not hypothetical requirements from computer scientists.

The applicability of OOP to the computing problems in HEP is the subject of this paper [3]. Two separate projects have begun investigations into what typical HEP codes might “look” and “feel” like in OOP. The first is a Monte Carlo generator called McCOOP, and the second is a detector simulation/analysis program called GISMO. As we shall see, OOP provides a very natural dialect for these problems. Before describing these two projects, we digress with a brief description of OOP especially designed for people having a FORTRAN background.

2. OOP for FORTRAN People

Object-oriented programming is a technique more than a language [4] and usually is based upon an existing language. Our experience has been in Objective-C [5] and as the name implies, it has the C language at its foundation. (There is also an Objective FORTRAN on the market, and others.) The program unit in OOP is something called a *class*. A class is defined by two parts: (1) a *header* file listing and typing the *instance variables* and defining the *method* names and their arguments, and (2) an *implementation* file fleshing out the various *methods*. As the program is run, instances of a class, called *objects*, are created in memory by the operating system using *factory methods*. This is when memory is allocated to the instance variables in contrast to FORTRAN where the memory is usually allocated at the compile-link stage. As the name *factory method* implies, multiple instances of a class may be present in memory simultaneously, each with its own set of instance variables, sharing the same form and methods but acting independently. Each object also contains a pointer to its associated class methods to allow the system to determine which routines to call for a particular method.

In contrast, the data in a FORTRAN program might be in COMMON blocks or in data structures managed by a memory management

package (e.g., ZEBRA, JAZELLE, BOS, etc.). This data is manipulated and operated on by subroutines and functions. Now imagine breaking the data up into pieces along with the routines which operate on those pieces. These “chunks” of data/program might be termed objects. One of the basic ideas in OOP is to closely connect the code to the data that it deals with. For example, histograms and their associated routines could be recast as a class. The bins, bin limits, title, axis definitions, etc., would become the instance variables, while the operations of create, delete, clear, accumulate, and display would be the methods of this histogram class. Multiple instances of this class, i.e., several histograms, could be present simultaneously, each with its own bin size, title, accumulation variable, etc.

A method executes other methods in its own or other classes by sending *messages*, which are analogous to subroutine or function calls in FORTRAN. A message must contain an identifier of the class (i.e., a pointer to the target object), the name of the method, then possibly followed by arguments. In Objective-C, for example, the syntax is similar to Smalltalk:

[histogramaccum : x]

where *histogram* is a pointer to the target object, *accum*: is the accumulation method of the histogram class and *x* is the (single) value to be binned. The “:” is syntactical and separates methods and arguments from each other; messages are contained within brackets, “[.. ..]”.

An important feature of OOP is a property called “inheritance.” This is jargon for the fact that in OOP, the data, as well as the methods, are layered (i.e., form a hierarchical tree). This layering is much more extensive than in familiar procedural codes and does not refer to layering in the client server sense. In OOP, classes are *subclassed* to other classes. When this is done the new class “inherits” all the instance variables *as well as* the methods of ALL the classes lying above it in the hierarchy. This could be illustrated in the previous histogram example by subclassing our histogram class to a root

class which allocates and deallocates memory. The “create” and “delete” methods of the histogram class would then be inherited from the root class. There would be no need for explicit code inside the histogram class to enable this functionality. On the other hand, if an action method needs to be modified for the subclass, it may be *overridden* by simply including new code in the subclass definition.

This property of inheritance is perhaps one of the most difficult aspects of OOP (for us old FORTRAN types) to use effectively. It requires (of us) a new level of abstraction in the creation of our programs not directly encountered in procedural code. It is believed by some “OOPers” that abstractions rarely evolve from the top down, but instead emerge from the bottom up. This is to say that the programmer is usually not aware of a new abstract class until he perceives common properties of several existing classes. He then creates a new abstract class with these common instance variables and methods and “bumps them upstairs.” In spite of the difficulty (novelty?) in developing abstract classes, they usually prove to be quite valuable and efficient since they serve to establish the framework of the program as well as provide many useful reusable utilities which may be inherited by new classes.

Another OOP buzz word is “polymorphism.” What this refers to is that the same function in different classes can, and usually should, have the same method name. For example, to deallocate an object and free up some memory, we might have a “free” method. The principle of polymorphism is that all objects can share the same name of the deallocation method, say, “free,” but each class may implement it in its own way. This should be a familiar concept to us... it replaces “naming conventions” in procedural languages.

We end this section with a brief discussion on memory management and how data organization is handled in Objective-C. One important feature of OOP is the fact that memory

management and bookkeeping are *not* the responsibility of the programmer but of the compiler and the run-time system. (The programmer must free objects appropriately, however, in order to keep memory usage under control.) First of all, data is bound to objects in OOP. Since many instances of various classes may be present simultaneously, a bookkeeping aid for objects is needed. This is provided in Objective-C by a class called List. List objects allow looping over the objects they contain, globally messaging each of them in turn, and providing edit functions to add, delete, and insert new objects into the List. An example usage of a List object might be to manage a group of histogram objects. This would allow global clearing of these histograms with a single message. Another more primitive data structure management tool is the Storage class. This object manages a group of identical data structures, similar to the concept in JAZELLE of *families* [6].

Examples of OOP programs are familiar to most of us although we may not have recognized them as such. These include popular drawing programs on personal computers with the windowed desktop screen layout, etc. Once aware of OOP it is easy to imagine that most of the things created on the screen are in fact objects, and indeed, multiple objects!

We shall see that OOP provides a very flexible coding framework, and now briefly describe our present formulation of two different applications. Keep in mind that this is prototype code; these examples can, however, provide the basis for a full implementation.

3. McOOP

One important characterization of numerical simulation problems is whether or not the “active” elements, or degrees of freedom, are fixed in number and characteristics, or whether they vary, perhaps even randomly, during the simulation process. We shall argue that OOP can offer many unique advantages in this latter situation [7].

The methodology described here, valid for general simulation problems, is one of our general points; the decay Monte Carlo application is one important example in particle physics. There are many other possible applications. The OOP characteristics discussed above and their uses here are: encapsulation, which will allow a simple treatment of the physics of each particle type, messaging and polymorphism, which will clarify the operation and writing of the code, and inheritance, which will play a very important role in shortening, simplifying, and insuring consistency among the physics objects.

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. In addition to these variables, we also must add methods to allow a particle to perform an action, such as decay, in its own unique way.

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. To minimize the coding task, we will insist that the program use well-tested FORTRAN routines (e.g., phase space generators) whenever possible.

The highest class, a subclass of the abstract root class *Object*, will be denoted as *Particle*. In this class, all general and universal attributes shared by all particle types will be introduced as instance variables. Action methods of universal applicability and general utility will also be defined; this will therefore be the "largest" class. Many utility methods that are not used in the main code but are useful for examination, tweaking parameters, debugging, etc., are also included here for general availabil-

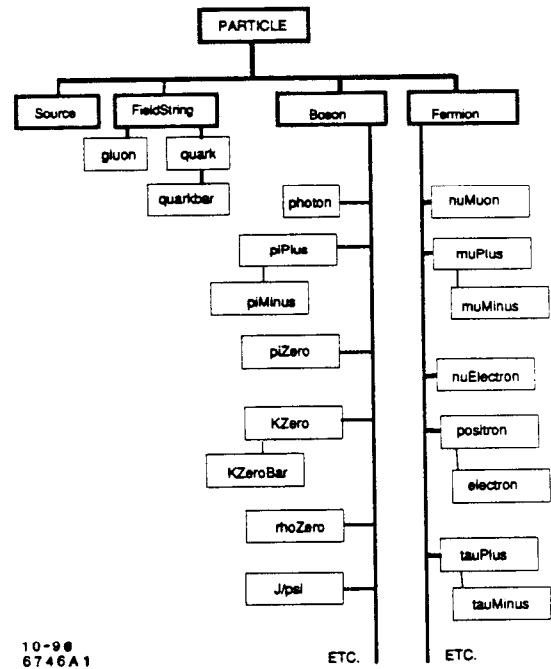


Figure 1. The decay Monte Carlo, McOOP, class hierarchy.

ity. The general structure of the hierarchy is shown in Figure 1.

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., 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. Polymorphism will be extensively used.

Let us now move down the hierarchy and first describe the classes that contain the physical particles. Two natural subclasses of the *Particle* class are introduced, *Boson* and *Fermion*.

Next, most of the physical particle types, the photon, electron, pion, proton, etc., can now be introduced as a subclass of one of these two classes. Since ordinary particles decay sequentially and independently (except for mixing which can be separately handled), the decay methods are straightforward to implement.

<pre> #import <objc/Object.h> #import <objc/List.h> @interface Particle:Object { char name[36]; 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; @end </pre>	<pre> #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. } prints out decay chain. </pre>
<pre> @implementation piPlus + create:sender { self = [super create:sender]; strcpy(name, "piPlus"); charge = 1.0; hypercharge = 0.0; mass = 0.139; br_muon = 0.75; is_stable = NO; has_decayed = NO; return self; } // end of create: </pre>	<pre> - decay { float rx; if(has_decayed is_stable) return self; rx = ranflat(0 < rx < 1); if (rx < br_muon) { //mode->muPlus+nuMuon [childList addObject:[muPlus create:self]]; [childList addObject:[nuMuon create:self]]; } else { //mode -> positron+nuElectron. [childList addObject:[positron create:self]]; [childList addObject:[nuElectron create:self]]; } has_decayed = YES; [childList makeObjectsPerform:@selector(decay)]; // sends decay message to each child on childList. return self; } // end of decay. </pre>

10-90
6746A2

Figure 2. The (.h) and (.m) files of the Particle class, listed on the left and right, respectively, are shown at the top. The (.m) file for the class piPlus is shown at the bottom. In the decay method, one creates an instance of each particle class in the chosen mode, sets the parent id, and forms the childList by invoking a single compound message.

Using charge conjugation invariance to further simplify the code, the negatively charged member is chosen to be a subclass of its positively charged partner; 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.

The class *FieldString* is introduced to handle the special properties of colored fields. For

example, the decay properties of quarks and gluons is a collective phenomena in most models of hadronization; the decay of a rapidly separating quark-antiquark pair and their accompanying gluons is controlled in many models by the properties of the colored string(s) that are formed. Thus in the general case, the decay must be treated coherently. This correlated hadronization process is handled in the FieldString class, again introduced as a subclass of Particle as shown in Figure 1.

The class *Source* is introduced to describe the physical origin of the fields. This is specialized to the originating collision, i.e., proton-proton or electron-positron, etc. For the electron collider case, it includes the initial beam characteristics, momenta, widths and polarization, and randomly chooses the final state, i.e., a particular type of lepton or quark pair. It then messages the relevant class to produce instances of the produced particles, assigns them their correct kinematics, and finally, orders them to decay. For $p - p$ or $p - \bar{p}$ collisions, multiple instances of quark/gluon, etc., must be created according to the hadronic interaction model at hand.

An edited version of the class code is shown in Figure 2 for Particle and π^+ . Note in particular the *create*: and the *decay* methods. Note also that the decay chain is stored NOT as a linked list, but rather as a "list linked by Lists."

This completes the discussion of the particle hierarchy 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.

4. GISMO

GISMO (Graphical Interface for Simulation and Monte carlo with Objects) is a recent undertaking centered at CERN [8]. The objective of this project is to explore the applicability of OOP to particle detector modelling and production data analysis in HEP. Specifically we hope to learn if the modelling/analysis task is more easily broken up into pieces than is presently the case. Also, with GISMO, a goal is to "re-use" modelling code to perform functions in the analysis chain. Both of these goals are supposed to be features of OOP and could make an improvement on the way HEP collaborations write their software.

GISMO will have a Graphical User Interface (GUI) allowing for detector modification in a drawing program environment. The same GUI will serve as the event display.

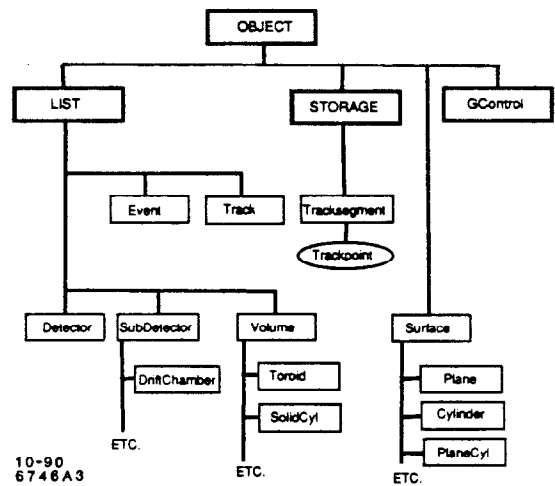


Figure 3. GISMO's class hierarchy.

The classes for GISMO are shown in Figure 3. Detector geometry is organized with the List class mentioned in Section 2. Detector is a "list" of SubDetectors, SubDetectors are a list of Volumes, and Volumes are a list of Surfaces which enclose them. Each of these classes is intended to be a superclass of specific types of detectors, volumes, and surfaces as indicated in the figure.

Inheritance plays a crucial role in this hierarchy. For example, Volume has instance variables for its title, a material-type index, and the maximum allowed step size for tracking. Variables to describe particular types of volumes are found in the classes subclassed to it. The Toroid class, for instance, has variables for its center, principle axis, length, inner and outer radii. No need to include title, material type, etc., as these attributes are inherited from Volume.

Recall that inheritance applies to not only the data, but to the methods as well. Volume is subclassed to List, hence anything a List can do (any messages a List will accept), Volumes can also do. This subclassing was done since most Volumes can be conveniently described by a list of surfaces (Volume is a List of Surfaces!). Volumes can be messaged to tell if a space point lies inside, how far along a helical trajectory to the boundary, as well as to be created and

deleted. Most of these methods can be generalized (with the exception of the create method) from all Volumes and hence appear in the Volume superclass. For example, to determine if a space point lies inside a Volume, the Volume messages each of its Surfaces to tell if a radial vector passing through the space point intersects it. If the number of intersected surfaces is odd, the point lies inside. This is quite general and usually won't have to be superceded by code in each specific type of volume subclasses to Volume.

In GISMO, subdetector definitions are also treated in a hierarchy. At the top there is the List class to manage the Volumes in the SubDetector. Then comes SubDetector itself. It provides methods common to all subdetectors such as track propagation and management of geometry related questions such as arclength to swim, "inside," etc. SubDetector is a superclass for real detectors such as DriftChamber. In DriftChamber, for example, are the details of a typical axial/stereo wire chamber. This in turn could serve as a model from which to build drift chamber variants, each providing specialized instance variables and methods peculiar to themselves. The original code, List—SubDetector—DriftChamber, need not be touched (i.e., rewritten) but simply appended to and superceded by the subclassing process using inheritance.

GISMO simulates events as follows. Detector ingests a Monte Carlo event and creates an Event object which is a List of the Tracks to be processed. The Track class is similar to Particle in McCOOP but also includes instance variables describing its spatial location, its present momentum, and so on. (In the next version of GISMO it would be interesting to subclass Track to the McCOOP Particle class!) Track is subclassed to List because if it decays or interacts it will be a list of its daughter Tracks. The Track class has the "stepby" method (i.e., "swim") and takes as arguments a distance to step by and the material (index) in which to step (necessary to compute multiple scattering, interactions and energy loss). The physics of

```

/* ----- Propagating the MCTrack through this subdetector ----- */
- propagate: (Particle *) MCTrack
{
    id trackseg, volume;
    float arclength;
    unsigned material, endptcode;
    BOOL alive;

    track = MCTrack;
    volume = [self locateVol];

    if(volume != nil) {
        trackseg = [Tracksegment new: track];
        [tracksegList addObject: trackseg];
        [trackseg addTrackPoint: ENTRY];

        material = [volume getMaterial];
        alive = YES; endptcode = ENTRY;
        while(alive & (endptcode != LEAVE)) {
            [self chooseStep: volume :&arclength :&endptcode];
            [track stepby: arclength in: material];
            alive = [track checkStatus];
            [trackseg addTrackPoint: (alive) ? endptcode: ENDPOINT];
        }
    }

    return self;
}

```

10-90
6746A4

Figure 4. The *propagate* method from the SubDetector class.

particles propagating in matter is part of the Track class definition.

Detector messages the first SubDetector with the id of the first Track in the event with the message to *propagate*. The Subdetector propagates this Track until it has decayed, interacted, or left the subdetector as illustrated by the code sample (see Figure 4). Detector proceeds to the next SubDetector and so on until it reaches the end of its list. Detector then processes the next Track until none are left "alive."

SubDetector creates a Tracksegment Object to manage the set of track points that are computed along a particles trajectory. Tracksegment is subclassed to the Storage class as track points are a fixed format data structure. Of course the number of track points in a Tracksegment is arbitrary. The Tracksegment class can display these track points for the event display and provides an implicit "loop" over trackpoints for modelling the subdetector's response. Tracksegments are managed in SubDetector by a List object.

An arclength is computed by taking the minimum of the maximum allowed step for the Volume, the distance to the wall, or some other criterion provide by a specific subdetector type.

Track is then messaged to “stepby” in the material of the Volume. At the end of each step the Tracksegment is messaged to add a track point to itself. The process is continued until the particle has interacted, decayed, or left the Sub-Detector. Once all the Tracks have been propagated through the Detector each subdetector will be messaged *simulateResponse* to compute and produce raw data facsimiles.

The analysis phase of GISMO has yet to be designated in detail. The idea will be to use the same object as in the simulation phase, adding methods to them to compare hypothetical tracks and interactions generated by a pattern recognition to those recorded either during simulation or actually measured in a real detector.

5. Conclusion

The OOP programmer does *no* explicit or detailed memory management *nor* other book-keeping chores; hence, the writing, modification, and extension of the code is considerably simplified. Inheritance can be used to simplify the class definitions as well as the instance variables and action methods of each class; thus the work required to add new classes, parameters, or new methods is minimal.

The software industry is moving rapidly to OOP since it has been proven to improve programmer productivity, and promises even more for the future by providing truly reusable software. The High Energy Physics community clearly needs to follow this trend.

REFERENCES

- [1] R. Brun et al., “GEANT: Simulation Program for Particle Physics for Particle Physics Experiments,” CERN-DD/78/2 (July 1978).
- [2] T. Sjostrand and M. Bengtsson, *Comp. Phys. Com.* **43**, 367 (1987).
- [3] For a data analysis application, see: W. B. Atwood, R. Blankenbecler, P. F. Kunz, B. Mours, A. Weir, and G. Word, “The Reason Project,” SLAC-PUB-5242 (April 1990).
- [4] See, for example, P. F. Kunz, “Object-Oriented Programming,” SLAC-PUB-5241 (April 1990); Invited paper given at 8th Conference on Computing in High Energy Physics, Santa Fe, NM, April 9–13, 1990.
- [5] See B. J. Cox, *Object-Oriented Programming, An Evolutionary Approach*. Addison-Wesley, 1986. Objective-C is available from Stepstone Corporation, Sandy Hook, CT.
- [6] A. S. Johnson et al., JAZELLE, *Proceedings of the 8th Computing in High Energy Physics Conference, AIP Conference 209*, 1989, p. 285; also see A. S. Johnson and D. J. Sherden, SLAC-PUB-263.
- [7] R. Blankenbecler, “Object-Oriented Programming for Simulation Problems in Physics,” SLAC-PUB-5251 (May 1990).
- [8] W. B. Atwood, T. Burnett, R. Cailliau, D. Myers, and K. M. Storr, GISMO (*Graphical Interface for Simulation and Monte carlo with Objects*), a CERN ECP Division Project.