



Fermi National Accelerator Laboratory

FERMILAB-Pub-93/079

**The Application of Object Oriented
Programming Methods to Event Delivery in
Experimental High Energy Physics**

Shigeki Misawa

*Physics Department, University of California
Berkeley, California*

*Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510*

March 1993

To be submitted to *Computers in Physics*

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

The Application of Object Oriented Programming Methods to Event Delivery in Experimental High Energy Physics

Shigeki Misawa

Physics Department, University of California, Berkeley, CA 94720

May 26, 1993

Abstract

Object oriented programming is a relatively recent development in software engineering that holds the promise of dramatically reducing the complexity of large computer software systems. Event delivery, that part of experimental high energy physics data analysis designed to convert raw experiment data to a format suitable for analysis, is a large software system that might benefit from the application of object oriented programming methods. This paper describes the results of applying object oriented programming methods to the development of an event delivery system for the E771 fixed target experiment at Fermilab. The paper is targeted at high energy physics experimentalists who are interested in new programming techniques but are unfamiliar with the concepts and vocabulary of object oriented programming.

Contents

1	Introduction	4
2	Caveats	4
3	An Overview of the Experiment and Analysis	5
4	Object-Oriented Programming	6
4.1	Object-Oriented Design	6
4.2	Object-Oriented Programming Languages	7
5	Event Delivery	9
5.1	Hits	9
5.2	Raw Hits	11
5.3	Planes	13
5.4	Readouts	16
5.5	Events	19
5.6	Data sources	20
5.6.1	Tape and Disk	20
5.6.2	Data Clients and Data Servers	21
5.6.3	Monte Carlo	21
5.7	Convenience Classes	22
5.7.1	Readout Lists	22
5.7.2	Groups and Group lists	22
6	Using the Event Delivery System	23
7	Conclusions	24
8	Acknowledgements	26
A	Addresses and Pointers	27
B	Casting	29
C	Complete Spectrometer Decoding	30
	References	34

List of Listings

1	Hit_1d class messages	35
2	Raw_Hit_1d class messages	35
3	Monte_Carlo_Hit class messages	35
4	Plane_1d class messages	36
5	Wire_Plane_1d class messages	37
6	Detector_1d class messages	37
7	Plane_List_1d class messages	37
8	Readout class messages	38
9	Event class messages	39
10	Data_Source class messages	39
11	Data_Server class messages	40
12	Mc_Event class messages	40
13	Readout_List class messages	41
14	Group_1d class messages	41
15	Group_1d_List class messages	41

1 Introduction

Experimental high energy physics (HEP) analysis software is a large and complex software system that is used to extract physics from data generated by an experiment. In order to obtain correct results from this data, it is crucial that the software be correct. Furthermore, to get results in a timely fashion, it is essential that the code be developed quickly and that it process data as efficiently as possible. In the context of a typical experimental HEP collaboration, achieving these goals can be extremely difficult. Therefore, it is vital that the best methods and language be used to develop the analysis software.

Object-oriented programming (OOP), a relatively recent development in the field of software engineering, is a programming methodology that promises to simplify and speed up the development of large software systems. This programming methodology consists of two components. The first component is a method of system design that is different from the traditional methods of structured design and algorithmic decomposition. The second component is the implementation of the resulting design in a computer language that supports the object-oriented programming paradigm (programming model).

In this paper, some of the concepts in OOP are introduced through their application to the design and implementation of the event delivery part of a HEP data analysis system. Event delivery is the “front end” of the analysis system and is responsible for converting raw data from the experiment to a format that is amenable to analysis. The goal of the design was to create an easy to use system from which different analysis programs could be rapidly constructed. The object-oriented (OO) event delivery system was one product of a project to develop a complete OO “drop in” replacement for the official analysis software for the E771 fixed target experiment at the Fermi National Accelerator Laboratory (Fermilab or FNAL). The official analysis software is currently being developed in FORTRAN using standard algorithmic decomposition methods.

This paper is directly targeted at high energy physics experimentalists who are interested in alternatives to FORTRAN and algorithmic decomposition design methods. No knowledge of OOP is required to understand the contents of this paper. However, familiarity with experimental high energy physics data analysis and apparatus is definitely required.

2 Caveats

Several points should be noted before getting into a detailed discussion of the application of OOP methods to event delivery. First, the application of OOP methods was made from “top down”. However, no attempt was made to “push” the OOP paradigm to the smallest details of the system. OOP purists could rightfully argue that the system is, at best, a hybrid design exhibiting only some of the characteristics of a “true” OO implementation. Second, significant time pressures present during the development

of the system impeded attempts to design a system that exhibits one of the desirable characteristics of an OOP system: reusability. The primary goal in the implementation was to get the system working. Third, many features required by “production” event delivery systems may not be present in the system. Finally, the event delivery system is implemented in the C++ programming language [1], a language that extends the C programming language [2] to support object-oriented programming. It is one of several object-oriented programming languages, e.g., Smalltalk80, Eiffel, that support the object-oriented paradigm [3, 4, 5]. Each language supports a slightly different vision of object-oriented programming. Hence, the design of the event delivery system will be influenced by the fact that it is implemented in C++.

3 An Overview of the Experiment and Analysis

Before getting into a discussion of object-oriented programming and the event delivery system, a quick overview of the E771 experiment and the data analysis problem is warranted. The E771 experiment is a fixed target experiment at Fermilab which looks at the interactions of high energy protons with a stationary or “fixed” target. The secondary particles, resulting from an interaction of a proton with the target, are detected with a spectrometer consisting of a series of detector planes. These detector planes come in a variety of types. They include silicon micro-strip detectors, multi-wire proportional (MWPC) chambers, drift chambers, pad chambers [6], scintillators, and muon detectors. An analysis magnet, also present in the spectrometer, provides momentum information about the secondary particles [7, 8].

The detectors in the spectrometer are grouped by the readout electronics to which they are attached. Since the identity of these groups is important in the discussion of the event delivery system, they are mentioned here. The Silicon detector group consists of silicon micro-strip detectors. The PCB and PC detector groups consist of multi-wire proportional chambers. The Drift and CC Wire detector groups are composed of drift chambers. The Pad detector group consists of pad chambers. The RPC and MUON detector groups are composed of muon detectors. (The scintillators are not dealt with in this event delivery system and will not be discussed.)

The data analysis problem for the experiment can be separated into three categories: detector characterization, analysis code characterization, and physics analysis. Detector characterization utilizes experiment data to obtain information about the detectors in the spectrometer. This includes such information as detector efficiency and alignment. Analysis code characterization utilizes data, from both the experiment and from a Monte Carlo simulation of the spectrometer and the interaction physics, to understand and verify the performance of the programs used in physics analysis. This includes such aspects as determining the efficiency of track finding and understanding the effects of various “cuts” used in the track finding process. The final component of data analysis is physics analysis. In physics analysis, data from the experiment is used to determine

particle decay rates, particle production cross sections, and other physical quantities.

A task that is common to all three components of data analysis is the construction of particle trajectories, for a given event, from the data obtained from the spectrometer (or a simulation of the spectrometer and the interaction physics). These trajectories (or tracks) are constructed from the hits registered by the individual detectors in the spectrometer. These hits, however, must first be extracted from the data for each event. For bandwidth and data storage reasons, the event data is stored in a compressed or packed format. Since the packed data is not directly usable, an event delivery system is required to convert the data to a format that is suitable for analysis.

4 Object-Oriented Programming

The creation of an object-oriented software system is a two step process. The first step is to use *object-oriented design* (OOD) methods to design the system. The second step is to implement the design in an *object-oriented programming language* (OOPL). But what is OOD and what makes an OOPL different from a traditional programming language? A detailed answer to these two questions is beyond the scope of this paper. However, a quick overview of OOD and OOPLs will suffice for the purposes of describing the application of OOP methods to the development of the event delivery system.

4.1 Object-Oriented Design

In designing a complex system of any type, software or other, the rule has always been to divide and conquer. A complex system is recursively partitioned into smaller and simpler components. It is hoped that the components that result from this decomposition will be simple enough to implement. Traditional software design methodologies, like algorithmic decomposition and structured design, partition software systems by function or algorithm [9]. A typical program resulting from these design methods consists of a series of nested functions. In contrast, OOD recursively partitions a system into “software objects”. Each software object provides a well defined set of services, with different objects providing different services. The process is similar to the hardware system design of a computer workstation. A workstation can be partitioned into a monitor, keyboard, and processor box. The monitor displays information obtained from the processor box. The keyboard sends inputs from the user to the processor box. The processor box responds to these inputs by changing its internal state and sending information to the monitor. The processor box can be partitioned into a motherboard, peripheral cards, mass storage subsystems, power supply, etc. The complete workstation consists of a collection of interacting components. Similarly, an object-oriented software system consists of a collection of interacting software objects.

The key to object-oriented design is the creation of a physical and/or conceptual model of the system to be built. This model is then partitioned into physical and/or

conceptual subsystems from which software objects are derived. In the case of the event delivery system, a combined physical/conceptual model was used as a basis for the design. The physical model was the E771 spectrometer and its data acquisition hardware. Some of the physical subsystems, from which software objects were derived, are detector planes, readout systems, and tape drives. Among the conceptual additions to this physical model are the notions of an event and a hit.

This association of physical and conceptual objects with software objects is one of the ways in which OOD simplifies software systems. The physical/conceptual model used in designing the system is typically something that is already familiar to the programmer. In this case, the HEP experimentalist is familiar with the spectrometer. In contrast, the subroutines in a traditionally designed event delivery system do not map as well, if at all, to anything that is already familiar to the HEP experimentalist.

Software objects are derived from physical and conceptual objects, but what does this really mean? Software objects are software representations or manifestations of physical and conceptual objects. They are designed to possess the same “behavior” as the physical and conceptual objects they represent. For example, a software detector plane object should be “filled” with “hits”. It should “know” how many hits were detected on the detector plane it represents. It should also “know” about its physical shape and position. Each “hit”, represented by a software hit object, should “know” what detector plane it is “in” and its position in that plane.

The behavior of a software object is manifest in its ability to respond to “messages”. By “sending” messages to a software object, it is possible to get a software object to answer questions, to store information, to process information, and to complete tasks. (In C++ terminology, the process of sending a message to an object is referred to as “invoking a member function for an object”.)

Finally, it is clear that different physical/conceptual objects possess different behaviors or characteristics. Similarly, different software objects “understand” different sets of messages. For this reason, software objects are placed into different “classes”. The class of an object determines the complete set of message understood by that object.

4.2 Object-Oriented Programming Languages

Object-oriented design outlines a means of partitioning and organizing a software system. Nothing prevents a system designed with OOD techniques from being implemented in a standard programming language. In fact, many OO software systems have been implemented with traditional programming languages, e.g., PenPoint [10], Xt Intrinsics [11]. However, the full benefits of OOD are not realized when this is done. As an analogy, consider a system designed using dynamic memory allocation and user defined data structures and implemented in a language that does not support these features. Both features can be simulated: user defined data structures with groups of arrays accessed with a common index and dynamic memory allocation with pre-allocated arrays and memory management routines. However, these ersatz constructs are not as “clean”

as the real construct and do not provide the complete benefits of the real constructs. The resulting system will not be as easy to read, modify, or maintain as a system implemented in a language that supports these features.

Intrinsic support for *classes* and *objects* is one of the features that distinguish an OOPL from a conventional language. In an OOPL, classes are extensions of types (e.g., `INTEGER`, `CHAR`, `REAL`) found in standard languages. Similarly, objects are simply variables of these “extended” types. In fact, in some OOPLs, no distinction is made between types and classes and between variables and objects. How classes are like types and objects are like variables is best illustrated through an example.

In FORTRAN, a variable can be declared to be of type `REAL`. The type of the variable defines how much memory is allocated for the variable, in this case enough to store a floating point number. Similarly, a class defines the amount of memory that is allocated for an object (of that class). More correctly stated, a class defines the types and numbers of state variables possessed by an object of that class. For example, a “hit” object needs to store position and measurement uncertainty information for a hit. The “hit” class would specify that an object of the hit class would possess (or consist of) two floating point variables. One variable would hold the position information and the other would hold the measurement uncertainty information. For this reason, a class is similar to a user defined type and an object is similar to a variable of that user defined type.

The similarity between types and classes can be extended to their “behavior”. The type of a variable defines what operations can be performed on a variable just as the class of an object defines what messages can be sent to an object. For example, a `REAL` variable can be truncated and negated (i.e., have its sign changed). In a pure OOPL, a `REAL` variable would be replaced by an object of the class `REAL`. Among the messages that would be defined for the class are `negate` and `truncate`. The process of negating a number would be accomplished by sending the object the `negate` message. In the case of the hit class, messages accessing information about the hit would be defined by the class.

The value of thinking of classes as extended types and objects as variables should not be underestimated. An object of a class can be passed to any function that takes an object of that class as a parameter in the same way that a `REAL` variable can be passed to any function that takes a `REAL` variable as a parameter. Multiple objects of a class can be created in the same way that multiple `REAL` variables can be created. Different messages are understood by objects of different classes in the same way that different operations can be performed on variables of different types. For example, consider the different operations that can be applied to `REAL` variables and `CHARACTER` variables in FORTRAN.

At this point, the basic OO design methodology and the OOP ideas of class and object have been introduced. This is sufficient information to start a discussion of the event delivery system. It should be noted that the important OOP (and OOPL) concepts of *class hierarchy*, *inheritance* and *derivation*, *polymorphism*, *dynamic binding*, *encapsulation*, and *data abstraction* have not been discussed. However, these concepts

are best explained in the context of their use in the event delivery system.

5 Event Delivery

Before getting into a discussion of the event delivery system, it should be pointed out that the discussion is tailored to users of event delivery systems, i.e., data analysis programmers. Little if any of the discussion will involve details of the implementation of the system (those aspects that are of interest to the designer and implementer of event delivery systems).

5.1 Hits

The simplest object of interest to the data analysis programmer is the hit. Physically, it is the interaction point between a detector plane and a particle. Conceptually, it is information about this interaction point, e.g., the position of the hit in the plane, the uncertainty in the measurement of its position, etc. Physical hits generated by the E771 spectrometer fall into two distinct conceptual categories which will be called 1D and 2D hits. 1D hits are generated by the MWPCs, drift chambers, and silicon micro-strip detectors. Physical hits in these detectors can be localized to a finite width line within the detector plane. That is, the position of a particle can be determined along only one axis that is transverse to the beam axis. The width of the line is determined by measurement uncertainty. Hits of the 2D variety are generated by the pad chambers, scintillators, and muon detectors. These detectors can localize a physical hit to a rectangular region in the detector plane, i.e., they provide position information along two orthogonal axes that are transverse to the beam axis. The size of the rectangular region is determined by the resolution of the detector.

The 1D and 2D hits are represented in the event delivery system, respectively, by objects of the `Hit_1d` and `Hit_2d` classes. Since these two classes differ in only “minor” details, only the `Hit_1d` class will be discussed. (This extends to the discussion in the rest of this paper. The `Hit_1d` class is one class in a set of classes to be discussed that work exclusively with 1D detector information. These other “1D” classes all have “2D” counterparts that are not mentioned in the paper.)

Since an object of the `Hit_1d` class is the software representation of a physical hit, it should provide information about the hit. This information is obtained by sending messages to the object. The messages understood by a `Hit_1d` object are shown in Listing 1. (Actually, this should be “*an object of the Hit_1d class*”, but it is much easier to say “*a Hit_1d object*”.) The format of the message declaration used in the listing is:

```
<return type> <message name> (<argument list>)  
    /* optional comment describing the message */
```

The word `void` in the argument list means that the message takes no arguments. Similarly, `void` as a return type indicates that the message returns no value.

Of the messages defined by the class, two are particularly important: `raw_hit()` and `plane()`. Respectively, they return the address of the `Raw_Hit_1d` object from which the hit was derived and the address of the `Plane_1d` object containing the hit. (See appendix A for a discussion of addresses.) These classes will be discussed later.

Now that the messages recognized by `Hit_1d` objects have been introduced, the following question arises, how is a message sent to an object? The following code fragment shows how the `x_pos()` message is sent to a `Hit_1d` object.

```
double x_coord;  
:  
x_coord = hit_object.x_pos();
```

The first line defines `x_coord` to be a double precision variable. The `{ ... }` contains the code that creates the `Hit_1d` object, `hit_object`. (The exact procedure for creating a `Hit_1d` object is left out because the end user never creates a `Hit_1d` object. All the `Hit_1d` objects are created by objects of classes to be discussed later. The process of creating objects will be outlined at that time.) The second line sends the `x_pos()` message to `hit_object` and stores the returned value in `x_coord`.

There are two important points to note about the `Hit_1d` class. First, the end user is completely ignorant of the implementation of the `Hit_1d` class. Given a `Hit_1d` object, it is not possible to determine how the information returned by the `x_pos()` message is stored within the object. The message interface hides the implementation details of the class. This is an example of the important concept of *data abstraction*. With data abstraction, it is possible to alter the implementation of a class without affecting any of the programs, written by the end user, that use the class. (This is assuming that the message interface is unaltered, i.e., the “names” of the messages remain the same.) Second, the message interface restricts the user’s ability to access and alter information contained in a `Hit_1d` object. This is an example of *encapsulation*. In the case of the `Hit_1d` object, only the `set_flag()` message alters the “internal state” of the object. This message sets an integer variable within the `Raw_Hit_1d` object associated with `Hit_1d` object to the value specified by the argument to the message. This variable is provided for whatever use the end user desires. (One possible use is to tag hits as “used”.) This ability to restrict access to information goes a long way in making programs easier to understand and more immune to bugs.

For simple classes, the benefits of encapsulation and abstraction achieved with the message interface may not be obvious. However, for more complex classes, the benefits are more apparent. The greater degree of encapsulation and abstraction that is obtained with messages is one of the advantages classes have over user defined data structures found in traditional programming languages.

5.2 Raw Hits

The `Hit_1d` object provides information about a hit that is of the most use to the data analysis programmer. However, some data analysis programs require hit information that is more closely related to the information obtained from the data acquisition hardware. This “raw” hit information takes the form of an integer identifying the individual detector wire in the detector plane that registered the hit. (This is for hits from the drift chambers and MWPCs. For hits from the silicon micro-strip detectors, the “wire” number identifies the individual strip in the detector plane that registered the hit.) Raw hit information for a hit is provided by the `Raw_Hit_1d` object that is created for the hit.

Listing 2 shows the messages recognized by `Raw_Hit_1d` objects. The `monte_carlo()` message is used to determine if a `Raw_Hit_1d` object represents a Monte Carlo generated hit. This ability to distinguish Monte Carlo hits from “real” hits is necessary because the testing of data analysis code sometimes involves overlaying a Monte Carlo generated event “on top” of a “real” event. If a `Raw_Hit_1d` object represents a Monte Carlo generated hit, additional information is available about the hit. This information is obtained simply by sending the `track_info()` message to the object. This message returns a `Monte_Carlo_Hit` object which provides “tagging” information for the hit. This tagging information is accessed with the messages defined by the `Monte_Carlo_Hit` class. (see Listing 3)

If raw hit information were restricted to just a wire number and Monte Carlo tagging information, then every hit would be adequately represented by a `Raw_Hit_1d` object. However, 1D hits are not homogeneous. For example, additional information in the form of drift distance data is available for each hit from a drift chamber. In the case of silicon micro-strip detector hits, the status of an rf flag, set by the readout electronics, is available for each hit. It would appear that `Raw_Hit_1d` objects are inadequate for representing hits from these two types of detectors.

A programmer familiar with a language that provides user defined data structures would immediately suggest setting aside extra space in a `Raw_Hit_1d` object to hold this additional information. There are two problems with this approach. First, it requires the user to be cognizant of the interpretation of the additional information, which may be different for different detectors. Second, if a new detector is installed that provides more information about a raw hit than can be stored in the allocated space, changes have to be made to the definition of the data structure.

In a language that does not support user defined data structures, raw hits would typically be stored in a set of arrays. One array would store the wire number for the hits, another would store the Monte Carlo track number for the hits, and another would store the Monte Carlo vertex number for the hits. A common index would be used to access the information for a particular hit from the different arrays. In such a system, the natural method for handling the additional information provided by the silicon microstrip and drift detectors is to assign additional arrays to hold this information. The problem with this approach is that the connection between the original set of arrays and the new

arrays is only conceptual. (In addition, the connection between the original arrays is also only conceptual.)

The object-oriented method for handling the differences between the different raw hits is to utilize the *inheritance* and *polymorphism* mechanisms present in OOPLs. Inheritance is an OOP concept which deals with the relationship between similar classes. Inheritance allows new classes to be built from pre-existing classes. Almost all the functionality required for a drift chamber hit is provided by the `Raw_Hit_1d` class. It would be convenient to be able to create a class for drift chamber hits by taking the implementation of the `Raw_Hit_1d` class and adding the features that are needed. By doing this, all the work involved in implementing and testing the functionality of the `Raw_Hit_1d` class will not have to be repeated for the drift chamber hit class. In OOP, this is achieved by “deriving” a new class, in this case the `Raw_Drift_Hit` class, from the old class, in this case the `Raw_Hit_1d` class. The `Raw_Hit_1d` class is called the base or parent class of the `Raw_Drift_Hit` class. The derived class “inherits” all the behavior of its parent class. A `Raw_Drift_Hit` object will therefore, recognize (i.e., respond to) the same messages as an object of the `Raw_Hit_1d` class. However, it will also respond to any new messages defined by the `Raw_Drift_Hit` class. This class defines the new message `drift_distance()` which returns drift distance information for the hit. In addition, it defines what additional memory is to be allocated for a `Raw_Drift_Hit` object to hold the drift distance information. Similarly, the `Raw_Silicon_Hit` class is derived from the `Raw_Hit_1d` class. The `Raw_Silicon_Hit` class defines the `rf_flag()` message, which returns the status of the rf flag for a silicon detector hit, and the state variable required to store the status of this flag. The relationship between these three classes constitutes a *class (or inheritance) hierarchy*.

At this point, a programmer familiar with only traditional programming languages would point out that inheritance has provided a means of adapting a pre-existing class to create new classes with additional functionality, only to create a new problem. In the eyes of such a programmer, this “problem” is in the relationship between a `Hit_1d` object and a `Raw_Hit_1d` object. The `raw_hit()` message defined by the `Hit_1d` class returns the address of a `Raw_Hit_1d` object. It would appear that the existence of the `Raw_Drift_Hit` and `Raw_Silicon_Hit` classes would require the existence of `Drift_Hit` and `Silicon_Hit` classes, with each of these latter classes defining a `raw_hit()` message, respectively returning the address of a `Raw_Drift_Hit` object and a `Raw_Silicon_Hit` object. However, the `Drift_Hit` and `Silicon_Hit` classes are not needed because OOPLs support the OOP concept of *polymorphism*.

`Raw_Silicon_Hit` and `Raw_Drift_Hit` objects understand all the messages that a `Raw_Hit_1d` object understands. Therefore, from a functional perspective, a function that expects a `Raw_Hit_1d` object should not care whether it is passed a `Raw_Hit_1d`, `Raw_Drift_Hit`, or `Raw_Silicon_Hit` object. (Stated in another way, an object of a derived class cannot be distinguished from an object of the base class by sending the object the messages defined by the base class.) Because of this, OOP languages allow derived objects to be used wherever base class objects are allowed. This is *polymorphism*.

From the user's perspective, this ability makes "sense". For all intents and purposes, a `Raw_Drift_Hit` object behaves "just like" a `Raw_Hit_1d` and should therefore, be allowed anywhere that a `Raw_Hit_1d` is allowed.

In C++, polymorphism is manifest in the ability to use the address of an object of a derived class anywhere that the address of an object of the base class (from which it was derived) is expected. This means that a `Hit_1d` object can "legally" return the address of a `Raw_Drift_Hit` object when it receives the `raw_hit()` message. Conversely, a function that expects the address of a `Raw_Hit_1d` object as an argument can be passed the address of a `Raw_Drift_Hit` object. One drawback with polymorphism in C++ is that the address returned by the `raw_hit()` message is classified as the address of a `Raw_Hit_1d` object. Therefore, the `drift_distance()` message cannot be sent to the object at the returned address, even if it is a `Raw_Drift_Hit` object. However, this is not a fatal flaw. If the returned address is "really" the address of a `Raw_Drift_Hit` object, the address need only be "cast" into an address of a `Raw_Drift_Hit` object. (More information about this procedure is given in appendix B.) Once this cast is applied, the `drift_distance()` message can be sent to the object located at the returned address. Currently, the "real" class of the object, at the address returned by the `raw_hit()` message, can only be determined by knowing the detector type of the plane containing the hit, i.e., silicon detector plane, drift chamber plane, etc. This information can be obtained from the `Plane_1d` object returned by the `plane()` message defined by the `Hit_1d` class.

Several questions present themselves at this point. First, where do the `Hit_1d` objects come from? Second, why create two distinct classes, `Raw_Hit_1d` and `Hit_1d`, to hold hit information? Third, how are `Raw_Hit_1d` and `Monte_Carlo_Hit` objects placed into their respective "container" objects. The answers to these questions will be given much later in this paper. At this point it is more advantageous to discuss the classes that hold the `Hit_1d` objects for particular detector planes in the spectrometer.

5.3 Planes

A single `Hit_1d` object is perfectly nice, but a typical fixed target system generates hundreds of hits per event. The question that arises is, how will all these hits be managed? A programmer proficient in a language without user definable data structures would immediately suggest an array of `Hit_1d` objects. However, the question then arises, should there be a single array containing all the hits or should some other system be used? Clearly, a logical grouping of hits would be by detector plane. This would suggest that the logical data structure should be a two dimensional arrays of hits. Another possibility is to have a single one dimensional array, with hits from the same detector plane located in a contiguous section of the array. This leads to the next question, in what order should the planes be stored? Finding a single answer to this question is one of the problems with using arrays to store hit information. Another is that the use of arrays does not address the problem of storing other information that may be associated with a detector plane, for example, the Z position of the detector (i.e., the position of the

detector along the beam line), the type of detector (i.e., silicon detector, drift detector), the number of hits in the detector plane, etc.

The OO solution is to create a class, the `Plane_1d` class, to represent detector planes. This fits in nicely with the physical spectrometer, which is built of detector planes from which hits are extracted. In addition to providing information about a detector plane, a `Plane_1d` object also serves as a repository for the hits detected on the plane. A `Plane_1d` object behaves as if it contains an array of `Raw_Hit_1d` objects and an array of `Hit_1d` objects. The messages defined by the `Plane_1d` class are shown in Listing 4.

A peculiarity in the above description of the `Plane_1d` class is the statement that a `Plane_1d` object behaves like it contains an array of `Raw_Hit_1d` objects. This gets back to the question about the presence of the `Raw_Hit_1d` class. As was stated previously, not all data analysis programs require the hit information that is provided by the `Hit_1d` class. For these programs, the event delivery system allows hits to be decoded into only `Raw_Hit_1d` objects, thus reducing processing time. These `Raw_Hit_1d` objects are stored in the `Plane_1d` objects and are made accessible in the same array-like manner as the `Hit_1d` objects.

One final peculiarity is the presence of the `num_hits()` and `num_raw_hits()` messages. One would expect that these numbers would be equal; however, this is not true for three reasons. First, `num_hits()` will return zero if hits have only been decoded into `Raw_Hit_1d` objects. Second, a single raw drift chamber hit can map to two transverse coordinates because of directional ambiguities in the drift distance information. Thus, one `Raw_Drift_Hit` object can result in the creation of two `Hit_1d` objects. (Two `Hit_1d` objects derived from the same `Raw_Hit_1d` object will return the same `Raw_Hit_1d` object when sent the `raw_hit()` message.) Third, the current system allows hits on adjacent wires or strips to be consolidated should there be a desire to do so. This consolidation is made only with the `Hit_1d` objects, not with the `Raw_Hit_1d` objects.

As with the `Raw_Hit_1d` class, the `Plane_1d` class serves as a base class for other classes. The most important of these is the `Wire_Plane_1d` class. This derived class exists because the Drift, PCB, PC, and CC Wire detectors provide additional information about their respective geometries that the other 1D detectors do not. This information is accessed with the messages shown in Listing 5.

Two OOP features, not previously encountered, are used by the two `in_plane()` messages defined by the `Wire_Plane_1d` class. The first feature, *function overloading*, allows two messages to have the same name. The message `in_plane()` has been “overloaded” with two definitions. The compiler is able to determine which definition to use by the number and types (or classes) of the arguments passed to the message. The `in_plane()` messages were overloaded because they accomplish the same task, albeit with different arguments. The second OOP feature used by both `in_plane()` messages is *dynamic binding* (which is unrelated to function overloading). In order to discuss dynamic binding, it is necessary to discuss the hierarchy of plane classes and the detector planes they represent in slightly greater detail.

The Drift, PCB, PC, and CC Wire detector planes all contain dead regions around

the beam axis where particles cannot be detected. However, the shape of the dead region varies from detector to detector. For example, it is either a circle or square for the Drift detectors, it is a square for the PCB, and it is a rectangle for the CC Wire detectors. In a standard programming language, these differences are typically handled with a `switch` statement or a computed `goto` or some other multi-way branch that calls different functions depending on the type of plane being examined. Each function would determine if the point of interest was in the dead region for a different dead region shape. The drawback with this method is that each addition of a new detector with a different dead region shape would require the updating of this multi-way branch.

In an OOPL, dynamic binding, which is closely tied to inheritance, is the solution to the problem. With inheritance, a derived class can redefine messages defined by its parent class. For an object of a given class, the definition (of a redefined message) that is executed is the definition that is appropriate for the class to which the object belongs. The situation would seem to be less clear in the case of a pointer to an object. (FORTRAN programmers should read appendix A before continuing. The word pointer, in C++, has a definition that is different from its definition in FORTRAN.) Because of polymorphism, a pointer to a base class may contain the address of an object of a class derived from that base class. If a message that is defined by the base class and redefined by the derived class is sent to the object at the address contained in the pointer, which definition of the message gets executed? With dynamic binding, the answer is the version that is correct for the object at the address, irrespective of the class of the pointer. An object implicitly “knows” the class to which it belongs. Now that dynamic binding has been explained, we are ready to apply it to the problem of the `in_plane()` message.

With dynamic binding, the solution to the `in_plane()` function is to give each detector plane type, i.e., Drift, PCB, PC, etc., its own class, derived from the `WirePlane1d` class. Each individual class redefines the `in_plane()` message in a way that is appropriate for the shape of its dead region. (Note: This does not mean that every detector plane needs its own separate class. Clearly, all planes with square dead regions could be represented by the same class. The different square sizes could be handled by state variables possessed by individual objects of the class. The same holds true for other shapes.) With dynamic binding, any function that takes an address to a `WirePlane1d` object as an argument can also be passed the address of an object of a class derived from the `WirePlane1d` class. If the function sends the object (at the passed address) the `in_plane()` message, the definition of the message that is correct for the passed object is automatically to execute. The function is completely ignorant of the existence of any derived classes and the fact that they have redefined the `in_plane()` message.

In addition to hiding the diversity of dead region shapes, dynamic binding also eliminates the need to recode, as well as recompile, any function (that expects the address of a `WirePlane1d` object as an argument and sends that object the `in_plane()` message) should a detector with a different shaped dead region be added to the spectrometer. The maintainer of the event delivery system need only derive a new class from `WirePlane1d` that redefines `in_plane()` for the new dead region shape. Another benefit of dynamic

binding is that the information about the shape and size of the dead region has been localized within the plane object, along with the other information about the plane. If a standard multi-way branch had been used, this information would not have been as well localized.

Now that the `Plane_1d` class has been introduced, it is possible to answer the question: what created the hit and plane objects and placed the appropriate information in them?

5.4 Readouts

In the E771 spectrometer, groups of detector planes are connected to readout systems containing the data acquisition (DA) electronics. The DA electronics in each readout system compress the information obtained from the attached detector planes and send the data to a central event processor. This central event processor collects data from all the readout systems and combines them into a logical bundle called an event. This event is then written out to tape. This system, running in reverse, served as the physical/conceptual model for the remaining objects in the event delivery system. In this model, event objects, holding data for single events, are retrieved from data source objects. The event objects are then passed to readout objects which decode the data into hit objects and put them into plane objects.

The readout object must accomplish three tasks. First, it must create `Plane_1d` objects (or in the case of readouts connected to 2D detectors, `Plane_2d` objects) for each detector plane connected to the physical readout system the object represents. Second, it must fill each `Plane_1d` object with `Raw_Hit_1d` objects and `Hit_1d` objects representing hits contained in the physical detector plane represented by the `Plane_1d` object. Third, it must provide access to these `Plane_1d` objects.

In the E771 system, the detectors connected to any particular readout system are either all 1D or all 2D. No readout system contains a mixture of both detector types. This prompted the development of the `Detector_1d` and `Detector_2d` classes. The messages defined by the `Detector_1d` class are shown in Listing 6. This class provides access to the planes connected to a particular readout through an object of the `Plane_List_1d` class. The messages defined by this class are shown in Listing 7. A `Plane_List_1d` object is designed to look like an array of `Plane_1d` objects. One problem with the `Plane_List_1d` class is that the ordering of `Plane_1d` objects cannot be specified or altered by the end user. The `Group_1d` and `Group_1d_List` classes, to be introduced later, solve this problem.

The `Detector_1d` class defines how planes are bundled and accessed. However, a `Detector_1d` object does not create `Plane_1d` objects nor does it fill them with hit data. This is the responsibility of Readout objects. A Readout object “undoes” the work of a readout (DA) system. That is, it takes an event, extracts the data for the readout that the object represents, decodes this data into hits, and places the hit information into `Plane_1d` objects. The Readout object is also responsible for creating the `Plane_1d`

objects which represent the planes connected to the readout that the `Readout` object represents. The messages defined by the `Readout` class are shown in Listing 8.

The design of the `Readout` class is sufficiently interesting to warrant some discussion. The `unpack_event()` messages are responsible for extracting hit information, for the readout the `Readout` object represents, from the event data. This data is obtained from an object of the `Event` or `Mc_Event` class. (Both classes will be discussed later.) These messages also create the `Raw_Hit_1d` object for each hit and place these objects in the appropriate `Plane_1d` object. The creation of `Hit_1d` objects from `Raw_Hit_1d` objects is accomplished with the `decode_event()` message.

The `Readout` class, like several other classes in the event delivery system, implements a two tier error reporting system. The first level is for the novice/casual user, i.e., the category into which most users fall. The second tier is for the expert user, e.g., the designer/maintainer of the readout electronics represented by the particular object. For the expert user, the `unpack_num()` and `decode_num()` messages return an integer code specifying the exact error that occurred during the unpacking and decoding processes. For the novice user, information on whether or not the error is fatal is most important. For this reason, the class maps the integer error code into error levels or states that tell the users if an error is fatal and how to recover from an error if it is not fatal. The `unpack_event()`, `unpack_status()`, `decode_event()`, and `decode_status()` messages return this type of error information. (As an example, the return type of the `unpack_event()` message is `Unpack_Status`. The valid values for this type are of the form `unpack_xxx` where `xxx` can be `okay`, `bad_event`, `bad_tape`, `bad_device_data`, or `other_fatal_error`. The `bad_event` state tells the user that the current event is “corrupted” but that the rest of the events in the file are “okay”. The `bad_tape` state tells the users that the current event, along with the rest of the events in the file are corrupted. The `bad_device_data` state tells the users that the current event is okay but the data for the requested device or readout is corrupted. Finally, the `other_fatal_error` state tells the user that some other error has occurred that prevents the system from continuing.) It should be noted that an error will also trigger the output of an error message describing the error to a log file.

In order to accomplish its assigned tasks, the `Readout` object requires information about the configuration of the readout system it represents and about the detectors connected to it. Since this information is subject to change, each readout system has a set of parameter files associated with it that tracks its configuration. (The number of parameter files for each readout system is different, as are the contents of the files. The parameter files are inherited from the official FORTRAN data analysis system. However, the official FORTRAN routines that read these files are not used by the OO event delivery system.) These files are “dated” by tape set number, i.e., each set is valid only for a range of tape set numbers. (Data tapes are grouped into tape sets. There are twelve tapes per set. Each set is given a tape set number. Experiment data is written to tape sets, i.e., sequential events are written to different tapes in the tape set.) The `update()` message is used to tell the `Readout` object to configure itself, i.e., read the

parameter files appropriate for the tape set number that is passed as a parameter to the `update()` message.

From the description of the `Readout` class, several questions naturally arise. First, how are `Plane_1d` objects transferred to `Detector_1d` objects so they can be accessed? Second, how are the differences, e.g., different data encoding schemes, different parameter files, etc., in the (physical) readouts handled? The answer to these two questions is inheritance and derivation.

The first question is easily answered by using the OOP feature of *multiple inheritance*. With multiple inheritance, a class can be created that inherits the behavior of multiple base classes. A class called `Readout_1d` was derived from both the `Readout` class and the `Detector_1d` class. The `Readout_1d` class inherits the messages for handling the unpacking and decoding of hit information from the `Readout` class and inherits the messages for handling the distribution of hit information from the `Detector_1d` class.

The differences in the different readout systems are handled by deriving a new class from `Readout_1d` for each readout system. These classes redefine those messages that need to be re-implemented to cope with the differences in the physical readout systems. Currently, there are 8 derived classes, `Silicon`, `Drift`, `Pc`, `Pcb`, `Cc_Wire`, `Rpc`, `Muon`, and `Pad` classes, corresponding to the different readouts present in the spectrometer. (Some are derived from the `Readout_2d` class.) Since none of the 8 derived classes defines any new messages, the processes of unpacking, decoding, and delivering hit information to the end user are the same for all the readout systems. Thus, once the end user has learned how to work with one readout system, he/she has learned how to work with all of them.

Unlike the other classes encountered so far, when writing a data analysis program, an end user must create objects of these eight derived classes. Objects of the other previously discussed classes are all “pre-assembled” for the end user by objects of these eight derived classes. That is, the user only needs to create pointers to objects of these pre-assembled classes. (The syntax for doing this is shown in appendix A.) This leads to the question, how are objects created?

Creating objects in C++ is syntactically very similar to creating variables of a built-in type like `double` or `int`. The only difference is that the object name may be followed by a parenthesized list of parameters. A special message defined by the class, called the *constructor*, defines the number of parameters in the list. It also defines what is done during the creation of an object. (Constructor is C++ terminology, not object-oriented terminology.)

The constructor can be distinguished from the other messages defined by a class because it is given the name of the class, e.g., `Readout(const int tape_set_number)` is the constructor for a `Readout` object. Also, the constructor message does not specify a return type because it returns no value. (In reality, the constructor for this class takes no arguments. However, the classes derived from `Readout`, e.g., `Pcb`, `Drift`, `Muon`, etc., all define constructors with the same argument as this fictitious constructor.) The following code fragment creates a `Readout` object with the name `my_readout` :

```
Readout my_readout(2456);
```

The number **2456** is passed to the constructor of the **Readout** class. Once the object has been constructed, it is ready to receive messages.

Finally, some complex constructors can fail. The correct method for dealing with this possibility is to utilize the exception handling facilities provided by the C++ programming language. (Many OOPs provide language supported exception handling.) However, since most currently available compilers do not support this feature, an alternate method for handling constructor failure is used by the event delivery system. The `init_okay()` message is defined by all classes in the system that possess constructors that can fail. This message returns `false` if the constructor fails, otherwise it returns `true`.

The next logical question that arises at this point is, how is event data delivered to the **Readout** objects? The answer to this question is the topic of the next section.

5.5 Events

A **Readout** object obtains event data from an **Event** object which, in turn, gets the data from a **Raw_Event** object. The **Event** object holds the data for one event. It also provides access to a few items that are embedded in a section of the event data called the event header. The messages defined by the **Event** class are shown in Listing 9.

One peculiar aspect of the constructor for the **Event** class is the tape number argument. The **Event** class requires tape set dependent information to accomplish its assigned tasks. (These tasks are not apparent from the messages defined by the class.) However, unlike the **Readout** class, the **Event** class does not define an `update()` message to update this information. Instead, the `get_event()` message automatically updates this information if it is passed a **Raw_Event** object containing an event from a new tape set.

Like the **Readout** class, the **Event** class provides a two tier error reporting system. The `event_state()` returns a simple five state error code suitable for the novice user. The `status_num()` returns an integer code identifying the exact error encountered during the execution of a message.

The **Event** object obtains event data from a **Raw_Event** object. This object, in turn, obtains event data from one of the data sources to be described next. The **Raw_Event** class is completely uninteresting from the end user's perspective. It is an implementation artifact that could have been eliminated at the cost of increasing the implementation time by a few days. This is because the elimination of the class would have increased the complexity of the **Event** class, albeit only slightly. The end user sends no messages to a **Raw_Event** object.

5.6 Data sources

The last major component of the event delivery system is the data input section. The event delivery system currently provides four sources of event data: tape, disk file, Monte Carlo, and event server. This last source allows a group of networked workstations to run analysis code on event data distributed by a single workstation.

The behavior that was desired for all the data source objects was that of a single virtual file containing a series of events. The adoption of this common abstraction simplifies working with objects of all four classes. It should be noted that this virtual file model deviates from the actual storage format of the event data in the four data sources, sometimes significantly. This idealized behavior is yet another example of data abstraction.

The logical way of obtaining identical behavior from all four classes of data source objects is to derive them from a common base class. The base class would then define the messages implementing the single file behavior. However, for historical reasons, this was not done.

5.6.1 Tape and Disk

The first two classes that were developed were the `Tape` and `Disk` classes. Objects of these classes respectively represent data tapes and disk files. Identical behavior from objects of these two classes is obtained by deriving the two classes from the `Data_Source` class. The messages defined by this latter class are shown in Listing 10. Since the `Tape` and `Disk` classes define no new messages, their behavior is completely defined by the `Data_Source` class.

It should be noted that the `Tape` and `Disk` classes are quite different from the actual sources of data. The disk file contains event data intermixed with FORTRAN format information. Also, at the beginning of the disk file is a special block of data containing what is called, in E771 terminology, the tape header. (This file format is defined by the official FORTRAN analysis system.) On the other hand, data on tape is contained in multiple tape files, with each file segregated into 65536 byte blocks, each containing multiple events. Like the disk file, at the beginning of the tape is the tape header information. Despite the radically different formats, to the end user, both sources objects “look” like a simple file containing a series of `Raw_Event`'s. (Note : The `tape_set()` message is the only message that accesses information from the tape header. Although all the tape header data is read and “decoded”, the `Data_Source` class does not provide messages to access this information. This can be changed in a few minutes, but the information has never been needed.)

The constructors for the `Disk` and `Tape` classes are defined to take two arguments. The first argument to the `Disk` constructor is the file name of the disk file. The first argument to the `Tape` constructor is the device name for the tape drive. The second argument to both constructors is the input buffer size.

5.6.2 Data Clients and Data Servers

The `Data_Client` class defines an object that reads events distributed by an object of the `Data_Server` class. This latter class defines an object that reads event data from a tape and distributes events to different computers running analysis code. Each computer running analysis code uses a `Data_Client` object to retrieve the event data that has been sent to it by the `Data_Server` object. With these two classes, an analysis program that is built with the event delivery system can be easily modified to utilize the processing power of several workstations to analyze events while requiring the presence of only one physical tape drive.

Like `Disk` and `Tape` objects, a `Data_Client` object behaves like a single virtual file containing a series of events. However, the lack of a suitable response to the `rewind()` and `skip_record()` messages prevents the class from being derived from the `Data_Source` class. (Actually, it could have been derived from the `Data_Source` class if the `rewind()` and `skip_record()` messages were redefined to either do nothing or return with an error.) The messages recognized by the `Data_Client` are the same, i.e., have the same name, as those of the `Data_Source` class except the `rewind()` and `skip_record()` messages are not defined. Thus, for most applications, a `Data_Client` object works just like a `Disk` or `Tape` object. However, not deriving `Data_Client` from the `Data_Source` class has some drawbacks. For example, the address of a `Data_Client` object cannot be stored in a pointer to a `Data_Source` class.

The constructor for a `Data_Client` object takes two arguments. The first argument is a character string used to identify the `Data_Client` object from other `Data_Client` objects in data analysis programs running on other workstations. The second argument is the input buffer size.

A `Data_Server` object distributes the events to objects of the `Data_Client` class. The messages declared by the `Data_Server` class are shown in Listing 11. The program which distributes events is very simple, it creates a `Data_Server` object, verifies that it was correctly constructed, sends it the `initialize()` message, and finally, sends it the `start()` message. (Note: Tape header data is distributed to all clients, as should be apparent from the fact that the `Data_Client` class defines the `tape_set()` message.)

5.6.3 Monte Carlo

Monte Carlo data is handled differently from data obtained directly from the experiment. (Recall the existence of a special `unpack_event()` message, with `Mc_Event` as an argument, for the `Readout` object.) There are three reasons for this fact. First, Monte Carlo data is stored in a format that is significantly different from experiment data. (However, when decoded, it provides the same information that real data provides, along with some additional information.) Second, the other data sources were implemented before the possibility of reading Monte Carlo data was even considered. Finally, the design goals for the Monte Carlo system were completely different from those of the other data

sources. Because of these differences, the system for reading Monte Carlo data will not be discussed completely.

The `Mc_Event` class is the main class built for handling Monte Carlo data. From the end user's perspective, it combines the functionality of the `Data_Source`, `Raw_Event`, and `Event` classes. The messages defined by the class are shown in Listing 12. In addition to the message defined to return the next Monte Carlo event, the class also defines messages accessing specific information about the event. This information is useful in verifying the performance of data analysis code. Finally, the `Mc_Event` class defines the mysterious message `append()`. This message is used in the process of initializing the `Mc_Event` object. This process will not be discussed.

5.7 Convenience Classes

The classes discussed in the previous sections are all the classes (with the exception of the "2D" classes) needed to decode the entire spectrometer. In addition to these classes, several auxiliary classes are defined to simplify the task of working with the event delivery system.

5.7.1 Readout Lists

The simplest convenience class to describe is the `Readout_List` class. As its name implies, an object of the class holds a list of `Readout` objects. The object is designed to behave as if it contains an array of `Readout` objects. It is used to simplify the process of sending messages to the various readouts. Instead of explicitly sending the `unpack_event()`, `decode_event()`, and `update()` messages to each readout object, a looping construct can be used to send these messages. This is shown in the code sample shown in appendix C. The class also serves as a convenient "holder" for readout objects. The messages defined by the `Readout_List` class are shown in Listing 13.

A list of readout objects is created by creating a `Readout_List` object and appending `Readout` objects to the `Readout_List` object with the `append()` message. Once all the desired `Readout` objects have been appended, the `Readout_List` object is sent the `make_table()` message, which generates an array access table. This table implements the array-like behavior.

5.7.2 Groups and Group lists

The final set of convenience classes provided by the event delivery system organize the `Plane_1d` objects into more convenient groups. Due to time pressures, these classes do not exhibit the functionality that was originally envisioned for them. However, they are useful even in their current forms.

`Group_1d` objects store `Plane_1d` objects in an array-like format. However, the class only defines the messages needed to access the list of `Plane_1d` objects. (see Listing 14)

It is the responsibility of the classes derived from this class to define the messages that decide which `Plane_1d` objects will be stored. (It is the implementation of this aspect of the `Group_1d` class that is not what was originally envisioned.) Some of the more useful derived classes group planes by view (i.e., the orientation of the individual wires or strips in the detector), by their position relative to the analysis magnet, and in order of their position along the beam line.

The `Group_1d_List` class provides the same functionality as the `Group_1d` class except that it holds `Group_1d` objects instead of `Plane_1d` objects. The messages defined by the `Group_1d_List` class are shown in Listing 15. As with the `Group_1d` class, classes derived from the `Group_1d_List` class are responsible for determining what `Group_1d` objects to hold. Currently, the system provides classes that hold groups of silicon micro-strip detector (Si) planes, i.e., those Si planes that are downstream of the target (The incoming proton beam is defined to flow from upstream to downstream.); groups of beam Si planes, i.e., those Si planes that are upstream of the target; groups of upstream non-Si planes, i.e., upstream of the analysis magnet; and groups of downstream non-Si planes, i.e., downstream of the analysis magnet. Each group within these group lists contains planes of a single projection or view. That is, the wires or strips on the different planes in a group are oriented in the same direction.

6 Using the Event Delivery System

A description of the event delivery system does not show the complete benefits of the object-oriented system. For this reason, a sample program using the event delivery system is shown in appendix C. An additional reason for providing a sample program is to verify that the “mental picture” of an OO system, developed in the mind of the reader, resembles reality. For this latter reason, it is essential that the sample program be examined.

The sample program uses the event delivery system to decode all the detectors in the E771 spectrometer. The code is fully commented and should be understandable even to those unfamiliar with C and C++. (In order to keep the length of the appendix short, error messages have been deleted and non-critical message “passing” has been omitted in the sample program.) One point to emphasize is the building block quality of the program. The process of constructing the program consists of three steps. First, the classes to be used in the program are defined with the `#include` directives found at the beginning of the program. Each “included” file contains the definition of a class. (Unfortunately, due to poor design, some contain the definition of more than one class.) It should be emphasized that these files are not like the files containing `COMMON` blocks found in FORTRAN. No objects or global variables are created in these class definition files. Second, objects of the classes are created. Third, messages are sent to the objects to accomplish the desired tasks.

The example in appendix C shows a standard use of the event delivery system. Other

conventional uses of the system are to create programs that analyze the performance of individual detectors. These other programs can be rapidly constructed because of the building block characteristics of objects. These programs can then be turned into on-line detector monitoring programs by replacing one line of code. Instead of creating an object of one of the supplied data sources, an object of a new “on-line” data source class would be created. This object would retrieve event data directly from the central event processor in the spectrometer readout system. Since it is possible to have the new class understand the same messages as the supplied data source objects, no other changes would have to be made to the program.

The above programs are relatively ordinary uses of the event delivery system that are easily duplicated in a system that is designed and implemented using standard techniques and languages. However, there is one program that would be hard to duplicate with such a system without significant planning before the system was implemented. This program depends almost completely on the idea that classes are just like types and objects are just like variables. Because of this fact, it is possible to have multiple objects of the same class. Therefore, it is possible to create multiple `Silicon`, `Rpc` and other readout objects in a single program. Since each such object stores the hit information from a single event for the detector system it represents, it is easy to construct a program that works with hit information from multiple events, for each detector, at the same time.

7 Conclusions

The following is a short list of some of the benefits of the application of OOP to the event delivery system.

1) The event delivery system looks like the E771 spectrometer and data acquisition electronics. Both consist of hits, detector planes, readout systems, etc. Furthermore, all the components of the event delivery system “work” just like their counterparts in the E771 spectrometer and data acquisition electronics. All the “pieces” in the event delivery system are components with which the experimenter is familiar.

2) The use of classes and objects has made the task of decoding selected detector groups as easy as creating variables.

3) Through the use of derivation, similar objects (i.e., objects of related classes) “look” and “act” the same. For example, all the readouts are updated, decoded, and unpacked in the same way.

4) Differences between similar objects are “hidden” until they are needed. For example, `RawDriftHit` objects work just like `RawHit_1d` objects. However, if drift distance information is desired, it is easily obtained from the `RawDriftHit` objects.

5) Write access to information is restricted to “authorized” users by the message interface. For example, the hit information contained in `Hit_1d` and `RawHit_1d` objects cannot be altered by the end user. Access to plane information contained in `Plane_1d` objects is similarly restricted.

6) Implementation details are well hidden from the user by the class/object construct. For example, parameter information required for decoding and unpacking hits is completely hidden within the `Readout` objects. The `update()` message is the only clue that additional information, besides the event data, is required by the `Readout` object.

7) Information that is “logically” connected is localized within a common object. For example, information about a detector plane like detector position, detector geometry, and hits on the plane are all contained within a `Plane_1d` object.

8) Classes and objects have eliminated the need for `COMMON` blocks and global variables.

9) The class hierarchy provides a means of maintaining continuity between currently implemented components and future additions to the system. For example, the `Readout` class provides a template from which new readouts can be derived.

A discussion of OOP, in general, is now in order. First, the comment most programmers make about a description of an OO software system is: there is nothing here that can't be accomplished in language `XXXX`, where `XXXX` is the language that the programmer is currently using. This is typically followed by the comment: the object-oriented system looks just like a well designed system obtained using traditional design techniques. There are several responses to these comments.

First, as was stated before, it is definitely possible to implement a system designed using OOP techniques with a traditional programming language. However, the questions that need to be answered include: how clean and intuitive is the resulting system? How hard is it to construct such a system and how easy is it to modify it once it is written? In other words, is this an unnatural use of the given language? Could things have been simpler if the language provided support for the “programming paradigm”? If the resulting system is cluttered by implementation details then it is clearly not optimal. (The sample program in appendix C is provided to show the level of “cleanliness” that can be achieved with an OOPL.)

Second, it is true that object-oriented systems look like well designed systems obtained using traditional design techniques. However, one can clearly turn the statement around: well designed systems created using traditional design techniques look like object-oriented systems. For example, the UNIX file system can be interpreted as a system designed around a generic file class with messages like `ioctl`, `open`, `read`, `write`, `close`, etc., being recognized by objects of the file class [2, 12]. Different physical devices are then “derived” from the base file class. However, the real question is: which design methodology more consistently generates a well designed system? For the reasons outlined in this paper and others reasons that are not mentioned here, many software engineers believe that the answer to this question is object-oriented design.

8 Acknowledgements

This work was partially supported by the National Science Foundation under Contract No. PHY-9121416. The author was also supported by a Department of Education fellowship during part of the development of the system.

A Addresses and Pointers

Addresses and pointers are an integral part of C++. “A pointer is a variable that contains the address of another variable.” [2, Page 93] The primary use of pointers in C++ programs is to provide an alternate method of accessing a variable or object. The closest analogue to a pointer in FORTRAN is the dummy argument of a subroutine or function. For example, consider the following subroutine :

```
line 1 : SUBROUTINE SETVAR(X)
line 2 : REAL X
line 3 : X = 10.0
line 4 : RETURN
```

Within the SETVAR subroutine, the dummy variable name X serves as an alternate “connection” or “handle” to the variable that was passed to the subroutine. The subroutine sets this latter variable to the value 10.0. Consider the following code fragment.

```
line 1 : REAL A
line 2 : REAL B
line 3 :
line 4 : CALL SETVAR(A)
line 5 : CALL SETVAR(B)
```

In the first call to SETVAR, the dummy argument X provides an alternate way of accessing the variable A within SETVAR. The value of A is 10.0 after the call to SETVAR. Similarly, in the second call to SETVAR, X serves as another “name” for B. The value of B after the call to SETVAR is also 10.0. It should be stressed that a pointer should NOT be thought of as the index of an element in an array. One reason is that an index to an element of an array is an incomplete reference to that element. An index is useless unless the array (or arrays) into which the index “points” is accessible. This implies that any function (or subroutine) that utilizes an index must have access to the array (or arrays). But this exposes the contents of the entire array to the function, making it vulnerable to accidental or malicious alteration.

The following fragment of code shows how pointers are created and used in C++ :

```

line 1 : double a, b;
line 2 : double *x;
line 3 :
line 4 : x = &a;
line 5 : *x = 10.0;
line 6 : *x = *x + 5.0;
line 7 : x = &b;
line 8 : *x = 7.5;

```

Line 1 defines the variables `a` and `b` to be double precision variables. Line 2 defines the variable `x` to be a pointer that can hold the address of a double precision variable. In line 4, the address of the variable `a` is stored in the pointer `x`. The address is returned by the operator `&`. At this point, variable `x` is said to “point to `a`”. Line 5 sets the variable `a` to 10.0. (Note the asterisk in front of `x`. Without the asterisk, `x` is a pointer containing the address of `a`. The construction `*x` is an alternate “name” for the variable whose address is contained in `x`, in this case, the variable `a`. For example, in line 6, `*x` is equivalent to `a`. In line 8, `*x` is equivalent to `b`.) Line 6 sets the variable `a` to 15.0. Line 7 makes `x` point to `b`, i.e., places the address of the variable `b` into `x`. Line 8 sets `b` to 7.5. This code sample shows what pointers are and how they can be used. (However, these are not very interesting uses of pointers.) In C++, pointers are particularly important because they are the primary means by which objects are passed to other objects and to and from functions.

One additional piece of information that is of use is the method of sending a message to an object that is being pointed at by a pointer to an object. Consider the following code sample :

```

line 1 : Hit_1d *hit_ptr;
line 2 : double x_coord;
           :
line 3 : hit_ptr = & hit_object;
line 4 : x_coord = hit_ptr->x_pos();

```

The first line defines `hit_ptr` to be a pointer to an object of the `Hit_1d` class. The second line defines the double precision variable `x_coord`. The `{ ... }` contains the code that creates the `Hit_1d` object `hit_object`. The third line sets `hit_ptr` to point to the `hit_object` object. The fourth line sends the `x_pos()` message to the object pointed at by `hit_ptr`. It then places the returned transverse coordinate for the hit in `x_coord`. This last line should be compared to the syntax for sending the `x_pos()` message to a `Hit_1d` object shown in section 5.1. Note the use of the `->` here, in place of the `.` used in the example in section 5.1.

B Casting

In the discussion about `RawHit1d` and `Hit1d` objects, it was mentioned that the `raw_hit()` message, defined by the `Hit1d` class, returns the address of a `RawHit1d` object. However, some `Hit1d` objects will return addresses to objects of classes derived from the `RawHit1d` class. This presents no problem if the messages sent to the returned object are restricted to the messages defined by the `RawHit1d` class. However, if the returned address is actually the address of an object of a class derived from `RawHit1d`, it is sometimes necessary to send that object one of the messages defined by the derived class (but not by the base class). These messages can be sent to the object only if the address is first “cast” to an address of an object of the derived class. The following code shows how this is accomplished.

```
line 1 : RawHit1d *hit_ptr;
line 2 : RawDriftHit *drift_hit_ptr;
line 3 : double drift_dist;
        :
line 4 : hit_ptr = hit_object.raw_hit();
line 5 : drift_hit_ptr = (RawDriftHit *) hit_ptr;
line 6 : drift_dist = drift_hit_ptr->drift_distance();
```

The first two lines define pointer variables to `RawHit1d` and `RawDriftHit` objects respectively. The third line defines a double precision variable. The `{ ... }` contains the code that creates the `Hit1d` object, `hit_object`. The fourth line retrieves a `RawHit1d` object from the `Hit1d` object (it is assumed that what is returned is really the address of a `RawDriftHit` object). The fifth line casts (or changes) the address contained in the pointer `hit_ptr` into an address of a `RawDriftHit` object. This address is then placed in the pointer variable `drift_hit_ptr`. The `drift_distance()` message can now be sent to the object. This is done in the sixth line.

C Complete Spectrometer Decoding

```
#include <iostream.h>          /* include the declaration of the C++ */
#include <stdlib.h>            /* standard library classes */
#include "defines.h"          /* include the definition of the types */
                               /* (not classes) defined by the event delivery system */
#include "tape.h"              /* include the declaration of the Tape class */
#include "raw_event.h"         /* include the declaration of the Raw_Event class */
#include "event.h"             /* include the declaration of the Event class */
#include "readout.h"           /* include the declaration of the Readout class */
#include "silicon.h"           /* include the declaration of the Silicon class */
#include "drift.h"             /* include the declaration of the Drift class */
#include "pc.h"                /* include the declaration of the Pc class */
#include "pcb.h"               /* include the declaration of the Pcb class */
#include "cc_wires.h"          /* include the declaration of the Cc_Wires class */
#include "pad.h"               /* include the declaration of the Pad class */
#include "muon.h"              /* include the declaration of the Muon class */
#include "rpc.h"               /* include the declaration of the Rpc class */
#include "group_id.h"          /* These three statements include the declaration */
#include "grp_id_types.h"      /* of all the classes in the Group and */
#include "grp_id_list.h"       /* Group_Id_List class hierarchies */

/* This is the start of the main body of the program. argc and argv are, respectively */
/* the number of arguments passed to the program from the command line and the */
/* character array containing the command line arguments. */

main(int argc, char *argv[])
{
    /* Create a tape object connected to the tape drive specified in the command line. */

    Tape data_source(argv[1], 65535);
    if (data_source.init_okay() != true) /* If constructed incorrectly */
        exit(1);                       /* terminate execution */
                                        /* != is like the FORTRAN .NE. */

    int tape_set_num;
    tape_set_num = data_source.tape_set(); /* Get the tape set number of the tape */
                                        /* from the Tape object */

    Raw_Event raw_event;                 /* Create a Raw_Event object */

    Event event(tape_set_num);           /* Create an Event object */
    if (event.init_okay() != true)       /* If constructed incorrectly */
        exit(1);                         /* terminate execution */

    Silicon silicon(tape_set_num);        /* Create a Silicon object */
    Drift drift(tape_set_num);           /* Create a Drift object */
    Pc pc(tape_set_num);                  /* Create a Pc object */
    Pcb pcb(tape_set_num);                /* Create a Pcb object */
    Cc_Wires cc_wires(tape_set_num);      /* Create a Cc_Wires object */
    Pad pad(tape_set_num);                /* Create a Pad object */
}
```

```

Rpc rpc(tape_set_num);          /* Create a Rpc object */
Muon muon(tape_set_num);       /* Create a Muon object */

/* The init_okay message should be sent to all eight of the readout objects */
/* created above. In this example, the message is sent to only two of them */

if (silicon.init_okay() != true) /* If constructed incorrectly */
    exit(1);                    /* terminate execution */
if (drift.init_okay() != true)  /* If constructed incorrectly */
    exit(1);                    /* terminate execution */

Readout_List readout_list;      /* Create a Readout_List object */

readout_list.append(&silicon);  /* Append the readouts to the readout list */
readout_list.append(&drift);
readout_list.append(&pc);
readout_list.append(&pcb);
readout_list.append(&cc.wires);
readout_list.append(&pad);
readout_list.append(&rpc);
readout_list.append(&muon);

if (readout_list.make_table() != success) /* Create the array access table */
    exit(1);                             /* Stop if an error occurs */

/* Create the group list objects. Each group list contains a list of groups. Each group */
/* contains a list of planes of a single projection. The Si group list contains Silicon (Si) */
/* planes downstream of the target. The Beam Si group list contains Si planes */
/* upstream of the target. The Up group contains non Si planes upstream of the */
/* analysis magnet. The Dn group contains non Si planes downstream of the magnet. */

Grp_id_Si_List si_planes;
Grp_id_Beam_Si_List beam_planes;
Grp_id_Up_List upstream_planes;
Grp_id_Dn_List downstream_planes;

/* Define several variables and pointer variables needed later. */

Group_id *group_ptr;           /* Define a pointer to a Group_id object */
int number_of_groups;
int group_number;
Plane_id *plane_ptr;          /* Define a pointer to a Plane_id object */
int number_of_planes;
int plane_number;
Hit_id *hit_ptr;              /* Define a pointer to a Hit_id object */
int number_of_hits;
int hit_number;
Readout *readout_ptr;         /* Define a pointer to a Readout object */
int readout_num;
int number_of_readouts;

```

```

/* Get events from the data_source object until an error occurs. An error */
/* occurs when either a real error occurred or when no more events are found. */
/* The end_of_file() message can be sent to the data_source object to determine if */
/* the error was real or if the end of the "virtual" file was reached */

while(data_source.get_raw_event(raw_event) == success)
{
/* Retrieve a new event from the Raw_Event object. Stop if an error occurs. */

    if (event.get_event(raw_event) != success)
        exit(1);

    tape_set_num = event.tape_num(); /* Retrieve the tape number for the event */

/* Set up a loop over the readouts in the readout list. */

    number_of_readouts = readout_list.num_readouts();

    for(readout_num = 0; readout_num < number_of_readouts;
        readout_num = readout_num + 1 )
    {
/* Get the (readout_num + 1) Readout object from the readout list and send it the */
/* update, unpack_event, and decode_event messages. Stop if an error occurs. */

        readout_ptr = readout_list.get_readout(readout_num);

        if (readout_ptr->update(tape_set_num) != success) /* Actually, update() */
            exit(1); /* needs to be sent only when the tape set number changes. */
        if (readout_ptr->unpack_event(event) != unpack_okay)
            exit(1);
        if (readout_ptr->decode_event(0) != decode_okay)
            exit(1);
    } /* End of the loop over readouts. */

/* Clear the group lists and rebuild them. (This really only needs to be done when */
/* the tape set number for the new event is different from the previous event) */

    si_planes.clear_list();
    beam_planes.clear_list();
    upstream_planes.clear_list();
    downstream_planes.clear_list();

    if (si_planes.append(&readout_list) != success)
        exit(1); /* Error while rebuilding. Stop. */
    if (beam_planes.append(&readout_list) != success)
        exit(1); /* Error while rebuilding. Stop. */
    if (upstream_planes.append(&readout_list) != success)
        exit(1); /* Error while rebuilding. Stop. */
    if (downstream_planes.append(&readout_list) != success)
        exit(1); /* Error while rebuilding. Stop. */
}

```

```

/* As an example, groups will be retrieved from the silicon group list, one by one. From */
/* each retrieved group, planes will be retrieved, one by one. From each plane, hits will */
/* be retrieved, one by one. The information from each hit will then be printed out. */

/* Loop over the groups in the silicon group list */

    number_of_groups = si_planes.num_groups();

    for(group_number = 0; group_number < number_of_groups;
        group_number = group_number + 1)
    {
/* Get the (group_number + 1) group from the silicon group list and */
/* loop over the planes in the group */

        group_ptr = si_planes.group(group_number);
        number_of_planes = group_ptr->num_planes();

        for(plane_number = 0; plane_number < number_of_planes;
            plane_number = plane_number + 1)
        {
/* Get the (plane_number + 1) plane from this group and loop over the hits in it */

            plane_ptr = group_ptr->get_plane(plane_number);
            number_of_hits = plane_ptr->num_hits();

            for(hit_number = 0; hit_number < number_of_hits;
                hit_number = hit_number + 1)
            {
/* Get the (hit_number + 1) hit from the plane and print the information in it */
/* Information from the Raw_Hit_Id object associated with the Hit_Id object */
/* (and retrieved by sending the raw_hit() message to the Hit_Id object) */
/* will not be accessed or displayed. */

                hit_ptr = plane_ptr->get_hit(hit_number);

/* The cout construct below is like a FORTRAN WRITE statement */

                cout << hit_ptr->x_pos()           /* Print the transverse position */
                     << hit_ptr->z_pos()           /* Print the Z position */
                     << hit_ptr->local_z()         /* Print the local Z position */
                     << hit_ptr->sigma()           /* Print the position uncertainty */
                     << hit_ptr->flag() << endl;    /* Print the value of the user */
                                                    /* alterable flag */

            } /* End of the loop over hits in the plane */
        } /* End of the loop over planes in the group */
    } /* End of the loop over group in the group list */
} /* End of the loop over events in the data source */
} /* End of the program */

```

References

- [1] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Publishing Company, Reading, Massachusetts, second edition, 1991.
- [2] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.
- [3] Grady Booch. *Object Oriented Design*. Benjamin Cummings Publishing Company, Inc, New York, New York, 1991.
- [4] Adele Goldberg and David Robinson. *Smalltalk-80: The Language and Its Implementation*. Addison Wesley Publishing Company, Reading, Massachusetts, 1983.
- [5] Bertrand Meyer. *Eiffel : the language*. Prentice Hall Object Oriented Series. Prentice Hall, Englewood Cliffs, New Jersey, 1992.
- [6] L. Spiegel, *et.al.* A combination drift chamber/pad chamber for very high readout rates. Technical Report FERMILAB-TM-1765, Fermi National Accelerator Laboratory, November 1991.
- [7] Harry H. Bingham. Progress on E771 B detection experiment. In *A. I. P. Conference Proceeding No. 261*, page 73. American Institute of Physics, 1992. Santa Monica Workshop on Rare and Exclusive B and K Decays and Novel Flavor Factories.
- [8] T. Alexopoulos, *et.al.* B physics at FNAL E771. In *Nuclear Physics B (Proc. Suppl.)*, volume 27, pages 257 – 262. Elsevier Science Publishers B.V., 1992. 3rd Topical Seminar on Heavy Flavors, San Miniato.
- [9] Edward N. Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Programs and System Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1979.
- [10] GO Corporation. *PenPoint architectural reference*. Addison Wesley Publishing Company, Reading, Massachusetts, 1992.
- [11] Douglas A. Young and John A. Pew. *The X Window System : Programming and Applications with Xt*. Prentice Hall, Englewood Cliffs, New Jersey, open look edition, 1992.
- [12] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley Publishing Company, Reading, Massachusetts, 1992.

```

double x_pos(void)
    /* returns the transverse position of the hit */
double z_pos(void)
    /* returns the position of the hit along the beam axis */
double local_z(void)
    /* same as the previous message but in a translated coordinate */
    /* system (it is needed for numerical reasons) */
double sigma(void)
    /* returns the uncertainty in the transverse coordinate */
Plane_1d *get_plane(void)
    /* returns the address of the Plane_1d object representing */
    /* the plane in which the hit occurred */
Raw_Hit_1d *get_raw_hit(void)
    /* returns the address of the Raw_Hit_1d object from which */
    /* the Hit_1d was derived */
int flag(void)
    /* returns the value of a user settable flag in the Raw_Hit_1d object */
void set_flag(const int flag_value)
    /* sets the value of the user settable flag in the Raw_Hit_1d object */

```

Listing 1: Hit_1d class messages

```

int wire_num(void)
    /* returns the wire number for the hit */
int flag(void)
    /* returns the value of the user settable flag for the hit */
void set_flag(const int flag_value)
    /* sets the value of the user settable flag to flag_value */
Boolean monte_carlo(void)
    /* returns true if the hit was generated by the Monte Carlo */
Monte_Carlo_Hit *track_info(void)
    /* returns the address of the Monte_Carlo_Hit object containing */
    /* the tagging information provided by the Monte Carlo simulation */

```

Listing 2: Raw_Hit_1d class messages

```

int track(void)
    /* returns the track number for the track that created the hit */
int vertex(void)
    /* returns the vertex number for the vertex from which the track, */
    /* which generated the hit, originated */

```

Listing 3: Monte_Carlo_Hit class messages

```

Plane_Id id(void)
    /* returns the name of the detector group to which the */
    /* detector belongs */
Plane_Type type(void)
    /* returns the detector type, i.e., silicon strip, drift, etc. */
View view(void)
    /* returns the orientation of the wires or strips */
double z_pos(void)
    /* returns the position of the detector along the beam axis */
double local_z(void)
    /* same as the previous message, but in a translated */
    /* coordinate system */
double first_wire_pos(void)
    /* returns the transverse position of the "first" wire or strip */
    /* in the plane */
double last_wire_pos(void)
    /* returns the transverse position of the "last" wire or strip */
    /* in the plane */
double cos_xy(void)
    /* returns the cosine of the angle the wires or strips make with */
    /* a standard axis */
double sin_xy(void)
    /* returns the sine of the angle the wires or strips make with */
    /* a standard axis
/ int num_raw_hits(void)
    /* returns the number of raw hits on the plane */
int num_hits(void)
    /* returns the number of hits on the plane */
Hit_Id *closest_hit(const double x_coordinate)
    /* returns the address of the hit object closest to */
    /* the transverse coordinate specified in the argument */
Raw_Hit_Id *get_raw_hit(const int index)
    /* returns the address of the (index + 1) Raw_Hit_Id in the plane. */
    /* The + 1 is present because the class follows the C language */
    /* practice of referencing the first array element with a ZERO index. */
Hit_Id *get_hit(const int index)
    /* returns the address of the (index + 1) Hit_Id in the plane */

```

Listing 4: Plane_1d class messages

```

double base_sig(void)
    /* returns a base uncertainty associated with a hit */
    /* used in calculating the uncertainty in a hit */
double wire_space(void)
    /* returns the spacing between wires */
double x_min(void)
    /* returns the distance to the "left" edge of the detector */
    /* from the beam line */
double x_max(void)
    /* returns the distance to the "right" edge of the detector */
    /* from the beam line */
double y_min(void)
    /* returns the distance to the "bottom" edge of the detector */
    /* from the beam line */
double y_max(void)
    /* returns the distance to the "top" edge of the detector */
    /* from the beam line */
double x_offset(void)
    /* returns the x location of the center of the detector */
double y_offset(void)
    /* returns the y location of the center of the detector plane */
In_Plane_Answer in_plane(const double real_x_coord)
    /* determines if the x coordinate passed as an argument is */
    /* within the active region of the detector */
In_Plane_Answer in_plane(const double real_x_coord,
                        const double real_y_coord)
    /* determines if the (x, y) position passed as an argument */
    /* is within the active region of the detector */

```

Listing 5: Wire_Plane_1d class messages

```

Plane_List_1d *get_planes_1d(void)
    /* returns the address of the Plane_List_1d object */
    /* containing a list of Plane_1d objects. */
int num_views(void)
    /* returns the number of projections viewed by the */
    /* detector planes connected to the readout system */

```

Listing 6: Detector_1d class messages

```

int num_planes(void)
    /* returns the number of Plane_1d objects contained in the list */
Plane_1d *get_plane(const int index)
    /* returns the address of the (index + 1) Plane_1d in the list */

```

Listing 7: Plane_List_1d class messages

```

Readout(const int tape_set_number)
    /* Constructor for the Readout class. The argument is the tape set */
    /* number for the tape containing the events to be processed */
Boolean init_okay(void)
    /* returns true if the object was properly constructed. */
    /* Otherwise, it returns false */
Readout_Name name(void)
    /* identifies the readout system */
Success update(const int tape_number)
    /* updates run dependent information required for decoding */
Decode_Status decode_event(const int option_flags)
    /* decodes raw hits into hits. The option_flag is used to control */
    /* decoding options, like hit consolidation. */
Unpack_Status unpack_event(Event &event)
    /* unpacks raw hits from an event */
Unpack_Status unpack_event(Mc.Event &event)
    /* unpacks raw hits from a Monte Carlo event */
Decode_Status decode_status(void)
    /* returns a status code about the success or failure of decoding */
Unpack_Status unpack_status(void)
    /* returns a status code about the success or failure of unpacking */
int decode_num(void)
    /* returns a error code indicating the exact error encountered */
    /* during decoding */
int unpack_num(void)
    /* returns a error code indicating the exact error encountered */
    /* during unpacking */

```

Listing 8: Readout class messages

```

Event(const int tape_num)
    /* Constructor for the Event class. The argument is the tape set */
    /* number of the tape containing the events to be processed */
Boolean init_okay(void)
    /* returns true if the object was properly constructed. */
    /* Otherwise, it returns false */
Success get_event(Raw_Event &event)
    /* retrieves event data from the raw event object */
Event_Status event_state(void)
    /* return the error status */
int status_num(void)
    /* returns an integer code specifying the exact cause of an error */
int number(void)
    /* returns the event number for the event */
int trig_num(void)
    /* returns the value of the trigger word from the event */
int tape_num(void)
    /* returns the tape number for the tape from which the event was taken */
int drive_num(void)
    /* returns the tape drive that wrote the tape */

```

Listing 9: Event class messages

```

Data_Source(const unsigned int buffer_size)
    /* Constructor for the Data_Source class. The argument */
    /* specifies the size of the input buffer to use. */
Boolean init_okay(void)
    /* returns true if the object was properly constructed. */
    /* Otherwise, it returns false */
Success get_raw_event(Raw_Event &event)
    /* retrieves an event from the data source and places it into the */
    /* Raw_Event object */
Success skip_record(const int number)
    /* skips the next record in the data source */
Success rewind(void)
    /* goes to the first event in the data source */
int tape_set(void)
    /* retrieves the tape set number from the tape header */
Boolean end_of_file(void)
    /* returns true if the end of file has been reached */

```

Listing 10: Data_Source class messages

```

Data_Server(const char *data_file,
            const unsigned int buffer_size)
    /* Constructor for the Data_Server class. data_file contains the */
    /* device name of the tape drive containing the tape to be read */
    /* buffer_size specifies the size of the input buffer to use */
Boolean init_okay(void)
    /* returns true if the object was properly constructed. */
    /* Otherwise, it returns false */
Success initialize(void)
    /* Initialize communications with the Data_Client objects */
void start(void)
    /* Start the distribution of events */

```

Listing 11: Data_Server class messages

```

Mc_Event(const int tape_num)
    /* Constructor for the Mc_Event class. The argument specifies */
    /* the tape set number for the events in the Monte Carlo data file. */
    /* The file that is read is hardwired into the FORTRAN code */
    /* used by the class to access event data. The FORTRAN */
    /* routines are modified versions of the routines */
    /* by the official FORTRAN analysis system. */
Boolean init_okay(void)
    /* returns true if the object was properly constructed. */
    /* Otherwise, it returns false */
Mc_Event_Status get_next_event(void)
    /* retrieves the next event from the Monte Carlo data file */
int num_vertices(void)
    /* returns the number of interaction vertices in the event */
int num_tracks(int vertex_num)
    /* returns the number of tracks originating from the vertex */
int num_hits(int vertex_num, int track_num)
    /* returns the number of detector planes that detected the track */
    /* specified by vertex_num and track_num */
double vtx_coord(int vertex_num, int proj)
    /* returns the coordinate of the vertex specified by vertex_num */
    /* The coordinate is specified by proj */
double trk_momentum(int vertex_num, int track_num, int proj)
    /* returns the projection of the momentum of a particle */
    /* specified by vertex_num and track_num, along */
    /* the axis specified by proj */
Success append(Mc_Data *readout)
    /* Adds a Mc_Data object to the object. Used during initialization */

```

Listing 12: Mc_Event class messages

```
void append(Readout *readout)
    /* Appends a readout to the readout list */
Success make_table(void)
    /* Create the table that implements array-like access to the readouts */
int num_readouts(void)
    /* returns the number of readouts contained in the list object */
Readout *get_readout(const int readout_num)
    /* returns a pointer to the (readout_num + 1) readout */
```

Listing 13: Readout_List class messages

```
Plane_id *get_plane(const int plane_num)
    /* returns a pointer to the (plane_num + 1) Plane_id object */
int num_planes(void)
    /* returns the number of planes in the list */
```

Listing 14: Group_id class messages

```
void clear_list(void)
    /* removes all the Group_id objects from the list */
Group_id *group(const int group_num)
    /* returns a pointer to the (group_num + 1) Group_id object */
int num_groups(void)
    /* returns the number of groups in the list */
```

Listing 15: Group_id_List class messages
