Application of Object Oriented Programming Techniques in Front End Computers*

Joseph F. Skelly
AGS Department, Brookhaven National Laboratory
Upton, New York 11973 USA

Abstract

The Front End Computer (FEC) environment imposes special demands on software, beyond real time performance and robustness. FEC software must manage a diverse inventory of devices with individualistic timing requirements and hardware interfaces. It must implement network services which export device access to the control system at large, interpreting a uniform network communications protocol into the specific control requirements of the individual devices. Object oriented languages provide programming techniques which neatly address these challenges, and also offer benefits in terms of maintainability and flexibility. Applications are discussed which exhibit the use of inheritance, multiple inheritance and inheritance trees, and polymorphism to address the needs of FEC software.

1 Introduction

The Standard Model for accelerator control systems [1,2] describes two levels of computers, often called Console Level Computers (CLCs) and Front End Computers (FECs), joined by a network. The purpose of this paper is to discuss the advantages of using an Object Oriented Language in writing software for Front End Computers. The benefits of Object Oriented Programming (OOP) have been much discussed in recent conferences; what this paper focuses on particularly is its use in Front End Computers, and especially on the use of an Object Oriented Language in that environment, which is more novel. The presentation is technical, but at a conceptual level; no code is presented. The reader is assumed to be familiar with the basic concepts of OOP.

A basic difficulty in this endeavor is ensuring compatibility of the Object Oriented Language with the requirements of the Real Time environment. A commercial Real Time Operating System (RTOS) is an attractive solution to the need for providing real time performance in the FEC, but until recently there was no commercial RTOS that supported use of an Object Oriented Language; hence, an effort to achieve the benefits of OOP necessarily employed a procedural language. The use of an Object Oriented Language offers additional advantages, which this paper explores.

2 Historical review

The use of Object Oriented Programming in accelerator control systems was discussed in ICALEPCS'91 and has been revisited at each ICALEPCS since, with increasing enthusiasm. For the most part, the discussions have focused on the use of OOP in applications written for Console Level Computers [2,3,4]. One contribution at ICALEPCS'91 [5] discussed the use of OOP in a Front End Computer, but without benefit of a formal Object Oriented Language; this effort used an Object Oriented approach which was written in the procedural language C, a technical tour de force, motivated by precisely the constraint discussed above, the unavailability of an Object Oriented Language for the real time environment. That report is nevertheless a lucid and complete presentation of the software organization required for the FEC environment. Another report at ICALEPCS'93 [6] discussed prospective development at AGS and RHIC of FEC systems along these lines; the present report provides additional technical description of the AGS effort.

3 Technical context of AGS front end computers

The object oriented software techniques discussed here can be employed in any FEC, regardless of its technical context. For the sake of a concrete perspective, however, the AGS Front End Computer is described in this section.

The AGS control system conforms to the Standard Model, with Unix workstations used for Console Level Computers, and VME systems used for Front End Computers (as well as an inventory of legacy FECs of older design). Computer nodes in the control system are linked by Ethernet. An application program in a CLC communicates with an FEC by means of client-server techniques, using a Remote Procedure Call (RPC) protocol. The FEC nodes employ Single Board Computers with 68040 processors, residing in VME crates. The FEC runs a commercial Real Time Operating System (VxWorks) which is C++ friendly. The FEC-resident C++ objects which implement the accelerator device interface are called Accelerator Device Objects (ADOs) [6].

Accelerator Device Objects may be characterized as containing either homogeneous or heterogeneous data types; AGS ADOs are heterogeneous. This means that a single ADO contains data of multiple types, and provides an Application Programming Interface which communicates multiple data types. All AGS ADOs contain up to 4 command fields and up to 16 status fields; command and status fields are of type "char". In addition, AGS ADOs may contain setpoint and measurement fields which may be of another type, eg. type "int" or "float".

Use of heterogeneous ADOs permits all features of a complex accelerator device (such as a power supply or vacuum gauge) to be integrated into a single ADO.

4 Challenge of accelerator device objects

The fundamental software challenge presented in an FEC environment is orderly management of a broadly diverse inventory of ADOs while taking advantage of the substantial common features they possess. Perhaps the most significant such feature is a common Application Programming Interface, usually implemented via calls into device object methods from an RPC server task.

The diversity found in the ADO inventory is motivated by the diversity in underlying accelerator devices. This diversity is manifested in such characteristics as:

- Hardware interface
- Behavior
- Data content
- Timing requirements

This issue is addressed herein as a series of 5 specific challenges, along with their solutions. Each of these solutions is a simple application of a standard feature in Object Oriented Languages; the feature set assumed here is that of C++. By means of examples for each of these solutions, a set of ADO classes is developed, which implements a model of the device classes needed in an FEC software environment. In this context, only the ADO class software is discussed; there is no discussion of the procedural software required to implement RPC server functionality.

4.1 Challenge 1: Device inventory management; solution: inheritance

The challenge of device inventory management is addressed with inheritance. A base device class is defined, from which all other device classes are derived; this class might be called the "ADO" class. Then a single array of "ADO" objects will serve as the inventory record for all device objects, whether of the base class or of derived classes. The base "ADO" class possesses a set of methods which define the fundamental common behavior and interface for all derived classes.

For example, one might derive the following list of classes from a base "ADO" class.

class ADO
- class PowerSupply
- class TimingDevice
- class VacuumGauge
- class BeamPositionMonitor

4.2 Challenge 2: Diversity in hardware interface and behavior; solution: polymorphism

The challenge of diversity in hardware interface and behavior is solved using polymorphism, ie redefinition (overriding) of methods. The diversity in the classes derived from the "ADO" class can usually be confined to a few methods, and only these few methods need be redefined for each of the derived classes to implement the desired functionality.

For example, the "ADO" class might define the following methods

<u>class ADO Methods</u>

GetImmediateReport	- report state of device
GetDeviceDetails	- report device configuration details
RequestReport	- request reports on ensuing accelerator cycles
SendReport	- send previously requested report
Watchdog	- check device state, send error report if any
AcquireState	- obtain state information from hardware
CommandToDevice	- send new command to hardware
WriteArchive	- write cache record for device
ReadArchive	- read cache record (restore state after reboot)

The first 4 or 5 of these methods mainly implement client-server features, and the base class methods are probably valid as well for all derived device classes; one might debate the issue for the "Watchdog" method, based on behavioral considerations, and override on a case-by-case basis as needed. The "AcquireState" and "CommandToDevice" methods deal with the hardware interface, and certainly would be overridden. The last 2 methods help implement device persistence when an FEC is rebooted; the base methods here are probably adequate for derived classes as well.

The combined use of inheritance and polymorphism promotes extensive reuse of common methods, enhancing efficiency and maintainability.

4.3 Challenge 3: Diversity in data content; solution: templates

For each of the classes derived above from the base "ADO" class, one would probably employ different data types for the setpoints and measurements, eg:

Power Supply	Signed int
Timing Device	Unsigned int
Vacuum Gauge	Float
Beam Position Monitor	Signed int

This challenge can be addressed by using a templated ADO class, with the data type of the setpoint and measurement specified by the templated type. In this way, the code which handles setpoints and measurements need be written only once, resulting in substantial code reuse, and therefore efficiency and maintainability. The notation for such a templated class is, eg, ADO<signed int>.

4.4 Challenge 4: Semi-homogeneous subsets of devices; solution: multiple inheritance

A hypothetical FEC might have an inventory of Power Supply devices and VacuumGauge devices which were all interfaced using the General Purpose Interface Bus (GPIB); certain types of operations would be common for all GPIB devices. The challenge here is to deal efficiently with this commonality. An effective solution is the use of multiple inheritance. A new base class can be defined to deal with the common properties of GPIB-interfaced devices, perhaps named "GpibUnit"; device-specific information such as its gpib address would be private to each "GpibUnit" object.

In this case, one might define the following methods for the "GpibUnit" class:

class GpibUnit Methods

TransmitCmnd - transmit a command (over gpib)
ReceiveState - request state information, receive it (over gpib)

Then the "PowerSupply" and "VacuumGauge" classes are derived from both the "ADO" and "GpibUnit" classes. Methods in the "PowerSupply" and "VacuumGauge" classes could invoke the Transmit Cmnd and Receive State methods as necessary. The inheritance relationship then looks like this:

```
ADO<signed int>        GpibUnit          ADO<float>
               \           /          \          /
             PowerSupply        VacuumGauge
```

As above, use of multiple inheritance promotes code reuse.

4.5 Challenge 5: Chains of semi-homogeneous subsets of devices; solution: inheritance trees

A not uncommon circumstance is the need to support two (or more) almost identical power supplies, with only minor differences. For example, two sets of supplies from the same vendor, with one set operating in a monopolar mode and the other in a bipolar mode. The vendor has provided subtly different command strings to accommodate this situation. The solution here is to develop an inheritance tree which reflects the situation. An intermediate class eg "VendorPowerSupply", is provided to implement the common features, and then derived classes, eg "BipolarPS" and "MonopolarPS", handle the distinctive features. The inheritance tree looks like this:

```
ADO<signed int>                    GpibUnit
                 \                          /
                  VendorPowerSupply
                  /                          \
           BipolarPS                    MonopolarPS
```

Through the use of polymorphism, the final derived classes, "BipolarPS" and "MonopolarPS", can be extremely terse, since most of their features are implemented in the "VendorPowerSupply" class. Yet again, this approach promotes code reuse.

5 Conclusion

Use of each of these techniques leads to substantial efficiency in development and maintenance of software, due to the extensive reuse of code in Accelerator Device Object classes. It is equally important to note that these techniques also make the software more readable and comprehensible.

Development of such an FEC software environment began at the AGS in 1993, and commissioning began in 1994. These systems have been in operational use at the AGS since 1995. At present, 17 such FEC nodes are operational, supporting a total of some 8250 device objects, representing 125 different device classes. There are also some 30 legacy FECs of older design which will be either retired or replaced by FECs of the new standard. Beyond this goal, deployment of new FEC services (eg more capable RPC server features) as well as protocol improvements is envisioned.

* Work performed under the auspices of the U.S. Department of Energy.

Reference

[1] B. Kuiper, "Issues In Accelerator Controls", Proc. ICALEPCS 91, Tsukuba, 1991

[2] V. N. Alferov, et al., "The UNK Control System", Proc. ICALEPCS 91, Tsukuba, 1991

[3] J. Skelly, "Object-Oriented Programming Techniques for the AGS Booster", Proc. ICALEPCS 91, Tsukuba, 1991

[4] J. Chen, et al., "CDEV: An Object-Oriented Class Library for Developing Device Control Applications", Proc. ICALEPCS 95, Chicago 1995

[5] A. Götz, et al., "Object Oriented Programming Techniques Applied to Device Access and Control", Proc. ICALEPCS 91, Tsukuba, 1991

[6] L. T. Hoff and J.F. Skelly, "Accelerator Devices as Persistent Software Objects", Nucl. Instr. and Meth. in Phys. Res. A 352(1994), 185-188