Object Oriented Programming and High Energy Physics

David R. Quarrie
Lawrence Berkeley National Laboratory, Berkeley, CA, USA.

Abstract

This paper discusses some aspects of the use of object oriented programming in high energy physics. The first section covers the goals and some fundamental concepts and asks whether OOP is merely a refinement of previous experience or whether it is a radical paradigm shift. The following sections illustrate the use of OOP through the use of some case studies. These focus on identifying the underlying abstractions and deriving a class hierarchy from them. Finally the support for OOP concepts that the C++ and Eiffel programming languages provide, as used in the case studies, is discussed.

1 OOP in HEP: Evolution or Revolution?

1.1 What is OOP?

Object oriented programming (OOP) is an approach to software development that encompasses all phases, from analysis through design and implementation onto testing and maintenance. I will use the definition from Booch [1]:

Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

Unfortunately this definition uses terminology and concepts that are specific to OOP itself and which will be discussed later. However, it does not restrict itself to issues of programming language, but addresses the complete development cycle.

1.2 The goals of OOP

In the conventional model the software development cycle starts with requirements analysis, continues with a detailed design that addresses these requirements, then coding of an implementation that meets the design goals and finally testing and maintenance take place. This waterfall model is based on the underlying assumption that our knowledge during the requirements analysis phase is complete and the resulting implementation is coupled very tightly to that understanding. Variants of this model support the notion of some iteration between adjacent phases, but the basic principles remain unaltered. The resulting software is very tightly tied to our original understanding of the problem which is assumed to be complete.

Unfortunately experience teaches us otherwise; at the beginning of a project we almost certainly don't really understand the full implications or scope. In a survey of several hundred software projects [2], 42% of the maintenance costs (changes in the product following initial release) resulted from unforeseeable changes in the user requirements. A further 18% of changes resulted from changes to the data formats.

We frequently experience such changes within the HEP environment. Most modern detectors undergo some upgrade to their capability during their lifetimes, or their physics goals change, perhaps as a result of improvements in the accelerator luminosity or other outside influences. The goal of our software should therefore not only be correctness, the ability to match the requirements and specification, but should also be flexibility or extendibility, the ability to be adapted to changes in the specifications.

Another goal is to produce software that is reusable because it is not tightly coupled to a particular project. Reusability can lead to significant improvements in overall productivity, not necessarily for a single project, but by applying the same software across multiple projects.

1.3 Fundamental Concepts

The following concepts are fundamental to OOP and the object oriented approach.

- *Objects* are entities that combine the attributes of both data and code and which can be manipulated or changed. They thus embody state and have state variables or attributes. They also exhibit an interface that describes how they may be manipulated. Finally, when they are manipulated they should do something useful; they exhibit behaviour. Note that it is what they do that is important, not how they do it. Their inner workings should be of no concern to us, as long as their overall behaviour conforms to the specifications. The interface to a clockwork stopwatch is essentially identical to that for an electronic one and as long as either performs correctly, it is irrelevant for us to know the details of the internal springs and levers or electronics components. In summary, an object has attributes to describe its state and operations to describe its behaviour.
- *Abstraction* is *"the elimination of the irrelevant and the amplification of the essential"* [3]. The stopwatch is an abstraction that eliminates all details about its internal workings. We can start it, stop it and reset it. All stopwatches obey this abstract interface. Anything else is irrelevant. The focus should be on *what* an object does, not *how* it does it.
- *Encapsulation* or *information hiding* is a related concept to abstraction whereby portions of an object are hidden from the user. This results in an overall simplification since the programmer only has to deal with the interface or visible portion of the object, but also in additional security since the object is protected from inadvertent tampering. Thus the possibility of side effects from modifications are greatly reduced and the software becomes both more amenable to change, and more robust under conditions of change. The programmer wishing to make a change needs to understand a smaller portion of the overall system than would otherwise be the case. Finally, problems are easier to localize because the possible causes of corruption are reduced.
- *Classes* form the specification of an object. An object is an instance of a class or an instantiation from this specification. Objects that share the same structure and behaviour belong to the same class.
- *Inheritance* is the key concept that distinguishes object oriented from object based systems. In conjunction with abstraction it is the main provider of the extendibility and flexibility of OO systems. A class can be a specialization or extension of another class. A "dog" is a specialization of a "mammal", which is itself a specialization of an "animal". This is the inheritance relationship and is perhaps where the most confusing terminology appears. The parent class is sometimes called the ancestor or superclass, whilst the child class is sometimes called the descendant or subclass. Single inheritance is the situation when a class has a single direct ancestor, whilst multiple inheritance describes the case where a class has two or more direct ancestors (which might themselves have other ancestors).
- *Polymorphism* describes the relationship between classes that have identical interfaces but different behaviours. A favourite example of this that is often quoted is a collection of

drawable objects, each of which has a draw operation, which performs the appropriate drawing operation, a circle, rectangle etc.

- *Genericity* is the term used to describe parameterized interfaces. Thus a push-down stack may contain integers or floating point numbers, or other quantities. However, all stacks exhibit the same interface, they just operate on different data types.
- *Object Lifecyle Management* is the term used to describe the ability to dynamically create new objects or destroy them. A related concept is that of garbage collection; the ability to automatically destroy objects that are no longer referenced by any other objects and are hence no longer needed.
- *Relationships* can exist between classes. A client-server or “uses” relationship is one whereby one class, the server, provides services to another class, the client. A containment or “has-a” relationship is one whereby an instance of one class is held by an instance of another class. Containment is used to build classes that are composites of other classes.

1.4 The OOP Development Cycle

At the coarse level the object oriented development cycle is similar to that of the conventional, structured approach. They both begin with a requirements analysis phase, followed by design, implementation and finally testing and maintenance. However these phases both differ in their details and the iterative development cycle is different. OOA consists of identifying objects and their relationships from the problem domain. Structured Analysis consists of identifying the data items and the processes that act on them. OOD creates an architecture for the implementation by extending the identification of objects to the solution space, identifying common policies for error detection and handling, memory management and generalized approaches to control etc. Structured Design identifies a hierarchical decomposition into modules that perform specific operations.

One major difference between the two methodologies is the emphasis that OOP places on incremental development, sometimes called rapid prototyping. This has several advantages including the early and extensive testing of major system interfaces, obtaining early feedback from end users and the ability to see early results from a working system. Another strategy is the identification of class categories or clusters that have a close internal relationship. Development of such clusters can proceed in parallel, following their own internal development and testing cycle. Thus they themselves are subjected to their own iterative development, resulting in more robust software at the completion of the project. The concept is illustrated in Figure 1.

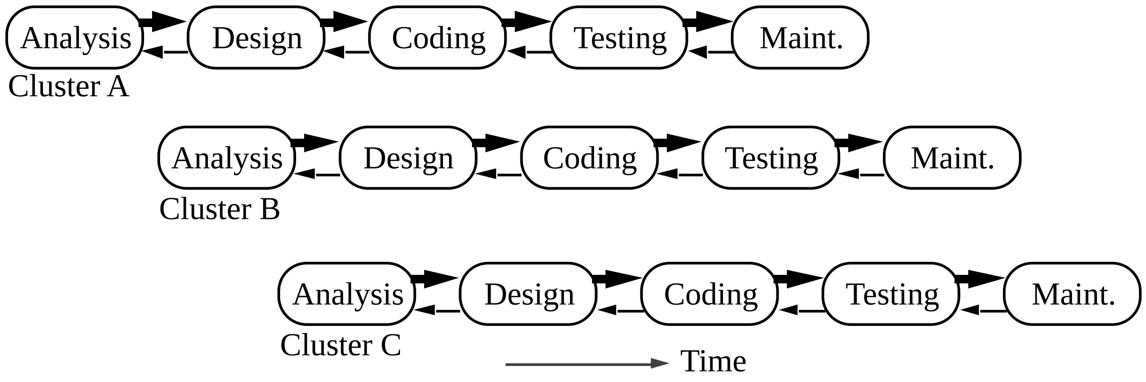

Figure 1: Iterative Class Cluster Development Cycles

Another difference between the OOP development cycle and the conventional cycle is that, once every class cluster is essentially complete, an attempt should be made to se whether any new abstractions can be identified from commonalities across the clusters. This identification can modify class hierarchies and produce further simplification for possible reuse.

1.5 History of OOP

Object oriented programming came into being in 1967 with the design of the Simula programming language [4] which was an extension of the Algol 60 programming language. The two major analysis and design techniques, Structured Analysis & Structured Design (SASD) and object oriented analysis and design (OOAD) can trace their roots back to about the same date when both the class and structured programming were introduced as concepts [5].

SASD and OOAD diverged almost immediately thereafter, with structured programming becoming very popular from the late 1960s, through the 1970s and early 1980s and finding its way into many large scale commercial software projects. For much of this period OOP remained in research labs and educational institutions and there were few languages that supported the concepts. Smalltalk was created during the 1970s [6], C++ [7] and Objective-C [8] in the early 1980s and Eiffel in 1985 [2]. Many other OO languages were created during this period but most have not survived in widespread use. Object oriented languages may be classified as either hybrid, building upon an already existing procedural language, or pure. Smalltalk and Eiffel are pure OO languages whilst Simula and C++ are hybrids.

Over the same period several OOA and OOD methodologies and notations have been created. A recent bibliography discussed 21 such notations [9]. Many share similarities, not surprising given their common support of the underlying object model. Perhaps the most popular are Booch [1] and Object Modelling Technique (OMT) [10].

1.6 OOP within HEP

The history of OOP within HEP goes back to a project using Simula in 1978 [11]. Other relatively early projects that I’m aware of are the Pions project at CERN [12] and the REASON project [13] at SLAC.

Despite these pioneering efforts, most projects involving OOP within HEP are still relatively small scale, and have tended to focus on components of on-line systems or graphics systems. However the Gismo [14] and GEANT4 [15] simulation frameworks and the work of several CERN R&D projects targeted at the LHC experiments are significant developments in the off-line environment. The BaBar collaboration at SLAC [16] is also intending to base its on-line and off-line code development on the OO methodology, using C++ as the implementation language. However, they recognise the large body of existing expertise in Fortran and C and are not excluding some continued development in those languages, albeit using object oriented analysis and design.

It is therefore clear that OOP is gradually becoming accepted within the mainstream of HEP software development. Crucial to this are not only the potential advantages of this technology, but also the continued ability to use the large body of legacy software, mainly written in Fortran. Unfortunately such inter-language interfaces are not well defined and this is an area of some concern. One possible solution to this problem is the Interface Definition Language (IDL) defined by the Object Management Group (OMG) as part of the Common Object Request Broker Architecture (CORBA) for distributed object applications [17]. This defines object interfaces in a programming language independent manner and also defines bindings to several programming languages including C, C++ and Smalltalk. A prototype binding to Fortran 90 is underway [18].

1.7 Evolution or Revolution?

Given its common ancestry with structured programming, it is clear that OOP is evolutionary and not revolutionary in origin. Does that mean that the impact and potential results are only evolutionary and not revolutionary? The jury is still out on that issue. Reusability is a prime goal of OOP and this should bring about significant gains in productivity amortised across several projects. However, other than user interface and data structure libraries there have been relatively few successful class libraries in other problem domains. This might change with the work of the OMG and the distributed object services [19] component of CORBA. The real revolution will perhaps come when the focus of most people's attention within the HEP computing community is on the complete software development cycle rather than the details of programming language issues. Analysis and design are hard and the OO approach considerably improves the intuitive decomposition of the problem and the flexibility of the resulting solution.

2 Case Study 1: On-line Run Control

The role of the data acquisition system in a high energy physics experiment is to take digitized event data from the front-end electronics, gather together the many fragments for a single trigger to form an event, perhaps reject events in a processor farm and then record those that survive on some form of archival storage.

A conceptual data acquisition system is shown in Figure 2a. It takes the form of a pipeline having several stages. The trigger indicates the occurrence of a potentially interesting physics interaction. Event fragments are digitized in the multiple front-end modules (FEM), processed in the read-out controllers (ROC), typically by having pedestals subtracted and channels with values below some threshold being suppressed. The Event Builder (EVB) creates complete events from the multiple fragments and the processor farm typically acts as a software trigger processor, performing a physics selection filter on complete events, accepted events being written to archival storage, typically magnetic tape. Events or event fragments might be buffered at some or all stages in the pipeline.

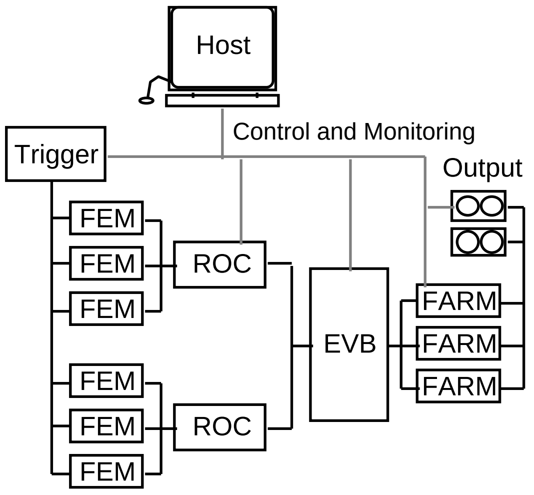

Figure 2a: Conceptual Data Acquisition System

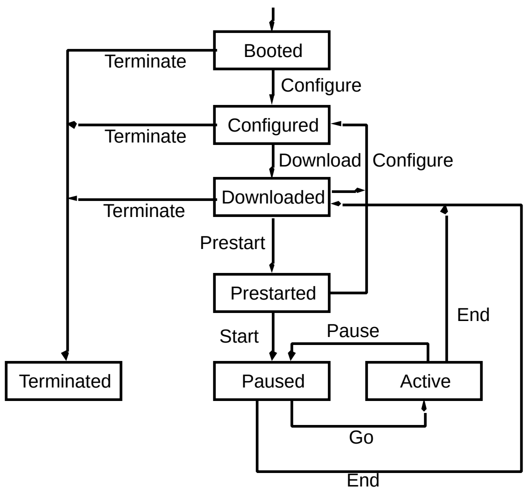

Figure 2b: Simplified State Transition Diagram

The role of the Run Control application is to control the elements of this pipeline during the various stages of data acquisition - starting a data taking run, perhaps pausing and resuming it and finally ending it. It plays no part in the actual flow of event data from the detector through the data pipeline, but communicates with the pipeline elements via a (conceptually) separate control and monitoring network. Most elements in the pipeline will be programmable, operating under control of a network aware real-time operating system, and require both code and data to be downloaded to them, depending typically on the type of data taking run that is to be performed and perhaps the results of a prior calibration process. Because of the buffering at the various stages on the pipeline, care must be taken to ensure that all such buffers are correctly flushed when pausing or ending a run.

The following discussion is based loosely on the CODA Run Control application [20] for the CEBAF data acquisition system which was implemented in the Eiffel programming language [2].

2.1 The Abstractions

2.1.1 Finite State Machines

The main abstraction is that of a Finite State Machine (FSM). A FSM can belong in one of several states, responding to external stimuli by performing an action and transitioning to another state. For our example, all elements of the pipeline react to the same set of external stimuli, obeying the same State Transition Diagram as shown in Figure 2b. This describes the various phases of a data taking run and the transitions that take place. Some of the main transitions are:

- Configure. This transition determines the conditions under which the forthcoming data taking run is to be performed. It might involve selecting one of several run types (*e.g.* physics, cosmics, calibration) that will modify the details of subsequent transitions.
- Download. This transition involves loading the distributed elements with the information that is appropriate for the forthcoming run. This might be implemented as actually downloading all the code and data, or might involve just informing the elements of the configuration details, letting them upload themselves over the network.
- Prestart. This primes the system for a new run without performing a new download operation, perhaps zeroing out scalers etc.
- Pause & Go. These initiate and halt data taking temporarily during a run.
- End. This terminates a run, perhaps closing the file on the output device.

2.1.2 Pipeline, Pipeline Stages and Pipeline Elements

The conceptual data acquisition system is based on a pipeline having multiple stages, each stage having possibly multiple elements. Each element is accessible from the host computer via the control and monitoring network and this access forms the basic granularity of the system. Typically the FEMs are not intelligent enough to be directly attached to the network so the ROCs are treated as agents for communication to them. If this simplified model is inadequate, the FEMs are treated as the second pipeline stage, the ROCs as the third. The event builder, processor farm and output device form the remainder of the pipeline stages.

A Pipeline Element is a software object that acts a proxy for the corresponding hardware element in the pipeline. The details of the communication protocol can be deferred to a discussion of the implementation classes. It will not be discussed further here.

The pipeline, the pipeline stages and pipeline elements all act as FSMs, obeying the same state diagram. This view raises the issue as to what state the pipeline or a pipeline stage is considered to be in while it is waiting for all its elements to complete their transitions. In the formal FSM model each transition is considered to be instantaneous, but in our model, it might take a finite amount of time for an element to complete a transition. This can be addressed by introducing additional “transitioning” states such as “going”, “pausing”, “downloading” etc.

Finally one must address the issue of how to deal with a situation where a transition is initiated but for some reason one of the elements fails to complete it successfully. The approach taken for this example does not conform to the formal FSM approach, but assumes that the pipeline as a whole will also fail to make the transition and so remain in its original state, albeit indicating that an error has occurred. All elements apart from the element that failed will however have completed their transition to the new state. Once operator or automatic intervention has corrected the problem with the failing element, the pipeline can be re-transitioned. This will act as a no-operation on all but the failing element since they are all in the desired state; only that element will be re-transitioned to re-synchronize the pipeline.

2.1.3 Sequencing

One aspect of the problem that is not immediately obvious is that the sequencing by which the various pipeline elements are transitioned. In general the elements at the same pipeline stage are independent of each other and so may be transitioned in parallel, but the different pipeline stages will need to be transitioned sequentially. Consider the situation when issuing a “Go” request to activate data taking. It is essential that the request is first issued to the tail of the pipeline, then to the penultimate stage and so on until the hardware trigger at the head of the pipeline is enabled. Conversely, when pausing a run, the pause request must first be issued to the hardware trigger, then the system must wait for event fragments to be flushed out from the next stage before the subsequent stages can be paused in sequence.

A simple sequencer might just be hardcoded to loop over the pipeline stages in the desired direction for each transition, but a more complex one might allow more flexibility.

2.1.4 Transitioners

At any state in the State Diagram, only a few stimuli and transitions to new states are valid. For example, it is not possible to transition from the Booted state directly to the Prestarted state. These restrictions are handled by the concept of transitioners that act on a target object, taking it from an initial state to a final state. The target of a transitioner could be the complete pipeline, a pipeline stage or an individual pipeline element. Transitioners understand the intermediate transitioning states and apply timeouts in order that the pipeline should not stall if a single element fails.

One possibility would be to allow sequences of allowed transitions. Thus one might be allowed to transition directly from the Configured to the Active state by passing through the Downloaded and Prestarted states. This would allow flexibility in the user interface and perhaps speed error recovery operations.

2.1.5 Data Taking Run

This concept just embodies the overall behaviour of a data taking run. It is a slightly broader concept than the pipeline since it has additional information associated with it. For example, it holds the knowledge of the current run number and the run type. It has a containment association

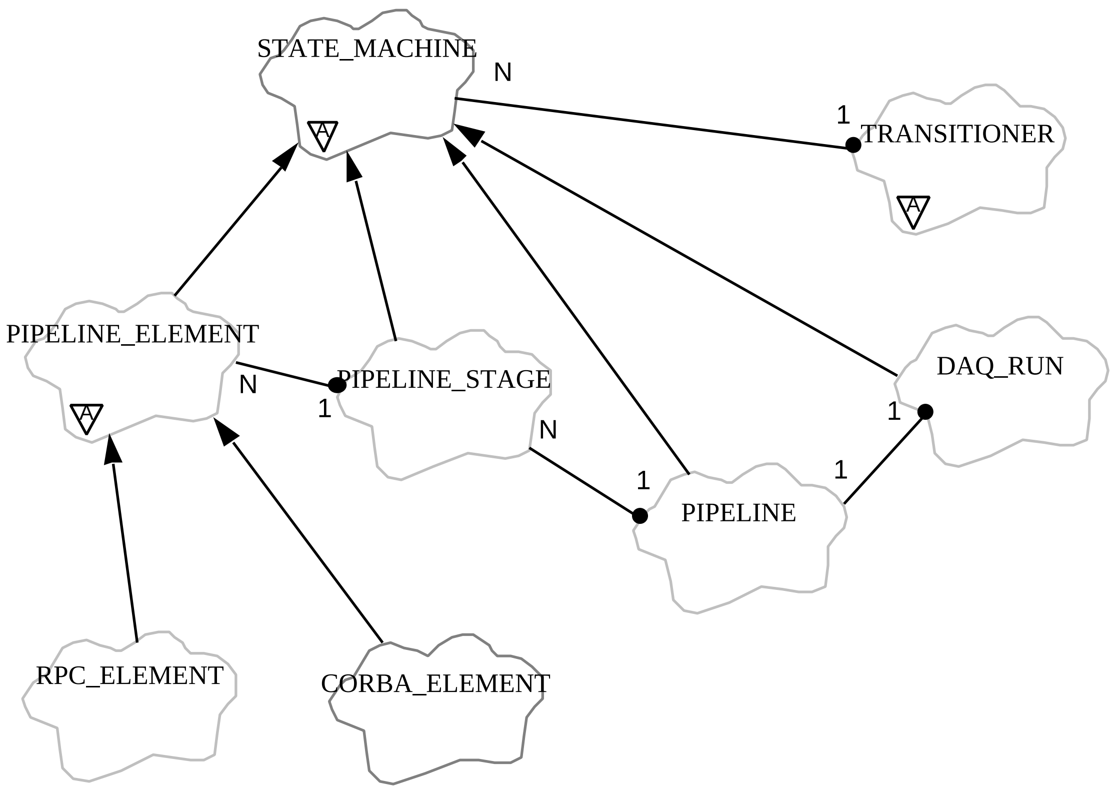

Figure 3: The class diagram

with the pipeline. The data taking run is the abstraction that the operator really wishes to interact with when controlling the experiment.

2.2 The Classes

The class diagram is shown in Figure 3. This uses the Booch notation [1] where the arrowed lines denote an inheritance relationship (the arrow pointing towards the ancestor) and the lines with a solid ball at one end denote containment relationships, the cardinality of which is expressed by the numbers appearing alongside.

The *STATE_MACHINE* class imposes the Finite State Machine on its descendant classes. Thus it requires that they respond to “Configure”, “Download”, “Pause” etc., without specifying an implementation. The *PIPELINE*, *PIPELINE_STAGE* and *PIPELINE_ELEMENT* classes are direct descendants of *STATE_MACHINE*. The *PIPELINE* class contains an ordered list of *PIPELINE_STAGE* objects and the *PIPELINE_STAGE* class contains a list of *PIPELINE_ELEMENT* objects. In this simple implementation the sequencing of transitions is performed by appropriate traversal of the list of *PIPELINE_STAGE* objects contained by the *PIPELINE* object. One difference between the implementations of the *PIPELINE* and *PIPELINE_STAGE* classes is that the *PIPELINE* class must transition its pipeline stages sequentially, whereas the *PIPELINE_STAGE* class transitions all its *PIPELINE_ELEMENTS* in parallel.

The *RPC_ELEMENT* and *CORBA_ELEMENT* classes are concrete classes that implement specific communication protocols between the proxy objects and the corresponding hardware.

The *DAQ_RUN* class adds the attributes of the run number and run type and contains a single instance of the *PIPELINE* class. It performs transition or status requests by delegation to the pipeline object.

The diagram is somewhat simplified in that not all the relationships between the *STATE_MACHINE* hierarchy and the *TRANSITIONER* class are displayed. Similarly, descendant classes of *TRANSITIONER* corresponding to the various transitions are not displayed.

2.3 Eiffel Language Support for OOP Concepts

This application is implemented in the Eiffel programming language. It's support for the various OOP concepts discussed in Section 1 is shown in Table 1.

Concept	Implementation	Code Example

Abstraction	Deferred Classes	`deferred class FS_MACHINE`
		`   ....`
		`   pause ( run: DAQ_RUN ) is`
		`      deferred`
		`   end;`

Encapsulation	Explicit export control
	Restricted export
		`class DAQ_RUN`
		`feature {NONE}`
		`   ....`
		`feature {DAQ_RUN}`
		`   ....`

Inheritance	Single & Multiple	`class RUN inherit`
		`   FS_MACHINE;`
		`feature`
		`   ....`
		`end`

Polymorphism	Unless *frozen*	`class RUN`
		`feature`
		`   frozen set_name( name: STRING ) is`
		`   ....`
		`   end;`

Genericity	Generic Classes
	Constrained Genericity
		`class LIST[T]`
		`class HASH_TABLE[T->HASHABLE]`

Lifecycle	Object Creation
	Garbage Collection
		`class RUN`
		`creation`
		`   make`
		
		`   !!run.make;`

Table I: Eiffel Language Support for OOP Concepts

3 Case Study 2: An Application Framework

The described application framework is suitable for general HEP reconstruction programs and is designed to provide flexibility without the need for recompilation. The implementation upon which this discussion is based is described in Reference [21].

Underlying concepts for the following discussion are those of data taking runs and events. A data taking run, identified by a run number, is a management unit of stability during data acquisition for an experiment. Such runs are characterized by having sets of adjustable parameters that are appropriate for data accumulated within the run, but which might differ from the values for adjacent runs. Thus any analysis code that wishes to operate on the data for a particular run must be given the opportunity to first access the appropriate parameters.

An event comprises the data for a single interaction or trigger of the experimental apparatus and is typically organized as a hierarchical set of data structures corresponding to the digitized output from the various detector subsystems.

A typical scenario for event reconstruction is the following: the events from one or more data taking runs are used as the input to an application that manipulates them, perhaps subsequently rejecting some of them as being uninteresting physics, or sorting them according to the underlying physics processes.

3.1 The Abstractions

3.1.1 Modules

The application framework is based on the concept of *modules*. A module is a fragment of executable code that has a well-defined interface and performs a well-defined service. The interface is imposed by requiring that each module inherit from an abstract parent class. Generally modules are totally independent of each other, operating purely on the basis of their own internal configuration, data taking run specific information and the input event data. A module might generate new information which might be added to the existing event information or might perform a filter function based on the event characteristics or might perform some statistical operation, integrating the results from multiple events.

Each module will provide an interface to the framework that includes a unique name and functions that will be called at the beginning and end of the job, at the beginning and end of each data taking run (*i.e.* when the run number changes) and a per event function.

Several types of specialized modules are supported within the framework. These include the following:

- Input modules, which act as the source of data. One example might read data from a data file or files, another might access events from the event server in the on-line environment and another might select events on the basis of a collection of objects stored in an object-oriented database (OODBMS). In all of these cases the remainder of the executable modules should be unaffected by the origin of the data. Only one Input Module can be active at any one time.
- Output Modules. These act as the sink of data. One will output event data to several possible output data files, thus supporting the concept of simultaneous output streams. Another will make events available to the on-line event server and another might store updated events in an OODBMS.
- Filter Modules. A filter module can terminate or re-direct the subsequent processing of an event based on its filter criteria and the characteristics of the event. A simple filter module can signify success or failure, terminating processing of the event in the latter case. A more complex filter module can act as a switcher, causing subsequent processing to be redirected to one of several other modules.

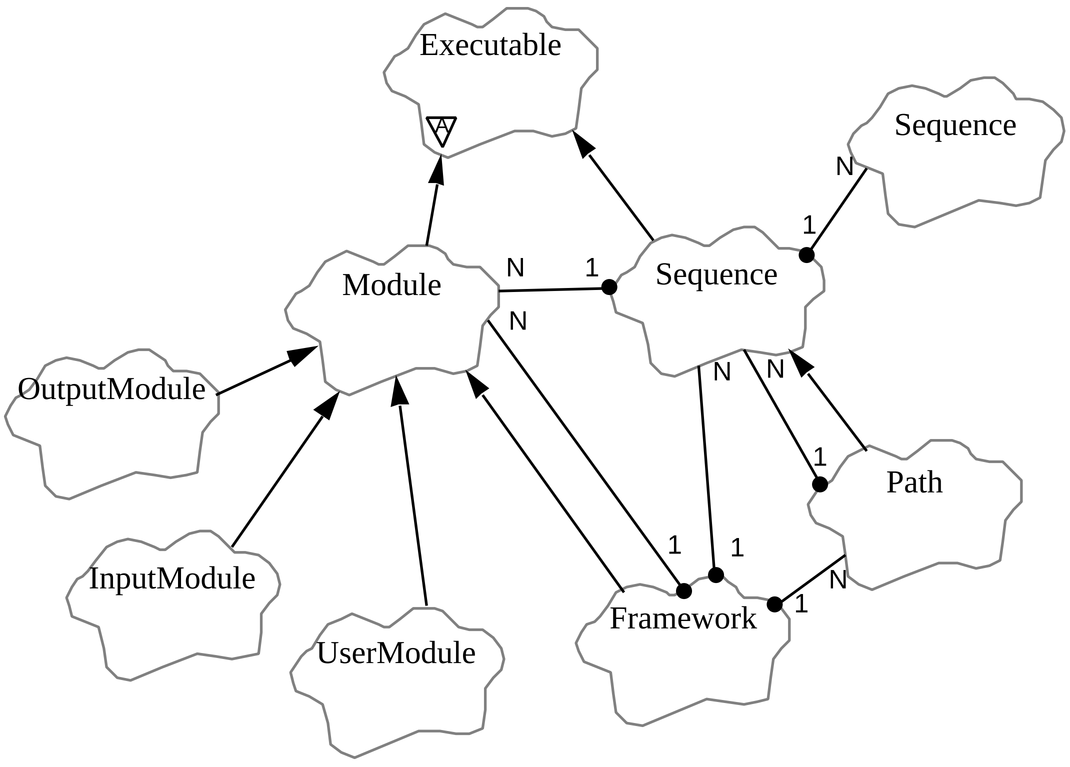

Figure 4: The framework class diagram

3.1.2 Sequences and Paths

Multiple modules can be combined into a *sequence* having a unique name. A sequence may also include other sequences to provide an arbitrary nesting depth.

A *path* is a list of modules and sequences that begins at the input module and terminates at the output module. The processing for a path may be prematurely terminated by the action of a filter module. Multiple paths are supported, corresponding perhaps to different physics processes.

3.2 The Classes

The framework class diagram is shown in Figure 4. The *Sequence* class appears twice because a sequence can itself contain another sequence.

3.2.1 Executable

The basic underlying abstraction is that of an *Executable*, an entity that provides the following services:

- A beginning of job service. This is requested once per job.
- A beginning of run service. This is requested whenever the run number changes.
- A per-event service. This is requested for each event that enters the processing chain.
- An end of run service. This is the complement of the beginning of run service.
- An end of job service. This is requested at the end of each job.

Note that these services are purely abstract, the main goal of the *Executable* class being to impose the interface upon its children, forcing them to provide implementations.

3.2.2 Module

The interface for the *Module* class is identical to that of *Executable*, but here it makes sense for there to be some default action for each of the services. This default implementation is a null operation (*i.e.* no action), but this is different from the situation with the *Executable* class, where there was no implementation specified. Each instance of the *Module* class may be enabled or disabled. This allows an application to be created containing a large set of modules, only some of which are of interest at any one time.

User supplied modules will inherit from *Module*, overriding the default implementations as they require, providing specific implementations for them as appropriate.

The *InputModule, OutputModule* and *FilterModule* classes are specialization of *Module*

3.2.3 Sequence

The *Sequence* class exhibits the identical interface to that of *Executable*, but the implementation of each service involves delegation to the members of the sequence. Thus the concrete implementation of each service will involve looping over all members of the sequence, delegating the request to them. In the case where the member is itself a sequence, it will again delegate the request to its members. This is an example of polymorphism, where several classes exhibit the same interface but have different implementations.

3.2.4 Paths

A *Path* is a specialized sequence and has the identical interface, and in the simplified model, the same implementation. However, it is treated as a descendant class because it has specific attributes in the full implementation that are not discussed here.

3.2.5 Framework

What is not perhaps immediately obvious is that the framework itself exhibits the same interface as *Executable*. It must provide services for the beginning of the job, beginning of run etc. It is only the implementation that is specific. In particular:

- The beginning of job service is delayed such that the beginning of job service for each module is accessed on the first occasion that the module is enabled.
- The beginning of run service is implemented as a loop over all enabled modules, accessing their beginning of run service.
- The end of run service is implemented as a similar loop to the beginning of run service.
- The end of job is implemented as a loop over all modules that have been enabled at some point in this execution, accessing their end of job service.
- The per-event service is implemented as an access to the event service for the enabled input module, followed by a loop over each path, accessing the event service for each sequence or module in the appropriate sequence. If a path is terminated through the action of a filter module, processing proceeds to the next path. Finally, the event service for the enabled output modules is accessed.

3.2.6 Lists and Hash Tables

Common to many of the application-specific abstractions is the concept of a list. A list is an object that chains other objects together, allowing a client to ask for each object in sequence. It has a head and a tail and mechanisms for inserting or removing elements.

Many possible implementation of a list are possible. The most space efficient are those based on arrays, but these are less efficient for adding or removing items within the list itself. Other implementations are singly (or doubly) linked lists, where node objects are defined that contain a pointer to the next (and previous) node as well as containing the item itself.

The main point here is that the application code should deal with an abstract list interface without worrying about the details of the implementation other than when instantiating the list. A subsequent decision to change the requirements resulting in a change in the list implementation (perhaps to allow for more efficient backwards traversal) then has very localized consequences; only the actual instantiation code has to be modified.

Another requirement is that the names of modules, sequences and paths be unique. Thus each new module must have its name checked against all the already existing ones. Whilst a list could be used for this purpose, a hash-table is a more efficient method of performing this, especially in this environment where wild-carding of names is not allowed. A hash-table uses a simple hash key, in this case derived from the name, and stores items indexed by this key, resolving collisions where two names hash to the same key. Access is then very efficient, most items being located following a single hashing operation. Contrast this to the situation where a list of names is maintained, every item of which has to be checked for a name clash.

The important point here is that the interface to an abstract hash table should be defined. Several different concrete classes that provide different implementations of this interface might then be provided. Initially an implementation based on a list class might be adequate. Eventually tests might show that the performance of this was no longer acceptable and another, more efficient concrete class having the same interface might be implemented. In this case the only change to the user code would be to instantiate the appropriate concrete class. No other changes would be necessary.

4 C++ Language Support for OOP Concepts

This framework is implemented in the C++ programming language. It's support for the various OOP concepts discussed in Section 1 is shown in Table 2.

Concept	Implementation	Code Example

Abstraction	Pure Virtual Classes	`class Executable {`
		`   ....`
		`   virtual event( ) = 0;`
		`}`

Encapsulation	Public, Protected & Private Members	`class Executable {`
		`public:`
		`   ....`
		`protected:`
		`   ....`
		`}`

Table II: C++ Language Support for OOP Concepts

Concept	Implementation	Code Example

Inheritance	Single & Multiple	`class Module : public Executable {`
		`   ....`
		`}`

Polymorphism	Virtual Functions	`class Module : public Executable {`
		`   ....`
		`   virtual void event( );`
		`}`

Genericity	Templated Classes	`template<class Item>`
		`class List {`
		`}`

Lifecycle	New & Delete
	Constructor & Destructor
	No Garbage Collection
		`Path* aPath = new Path( “MyPath” );`
		`   ....`
		`   delete aPath;`

Table II: C++ Language Support for OOP Concepts

5 Conclusions

Many of the abstractions in these case studies are intuitively obvious, this being one of the main attractions of the object oriented approach. Most of them were identified during the analysis phase of software development, but some were only identified later during the design phase. In particular, inheritance relationships are mainly identified during the design process. Inheritance is a powerful tool in both enforcement of a common interface and in providing a hierarchical specialisation of concepts. Polymorphism provides a complementary mechanism for simplification. These example demonstrate the support for OOP concepts by two different programming languages, one of which (C++) is a hybrid, being based on an existing procedural language, the other of which (Eiffel) is a pure object oriented language.

6 References

[1] G. Booch, *Object Oriented Analysis and Design with Applications*, Benjamin/Cummings, 1994.

[2] B. Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.

[3] R. Martin, *Designing Object-Oriented C++ Applications using the Booch Method,* Prentice Hall, 1995.

[4] O.-J. Dahl and K. Nygaard, *SIMULA - An Algol-based Simulation Language*, Communication of the ACM, vol 9, no. 9, pp. 671-678, 1966.

[5] O.-J. Dahl, E. Dijkstra and C. Hoare, *Structured Programming,* Academic Press, 1972.

[6] A. Goldberg and A. Kay, *Smalltalk-72 Instruction Manual,* Tech. Report SSL-76-6, Xerox Palo Alto Research Center. 1976.

[7] B. Stroustrup, *Data Abstraction in C,* AT&T Bell Laboratories Tech. Journal, vol. 63, no. 8, Part 2, pp. 1701-1732, Oct. 1984.

[8] B. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.

[9] A. Hutt, *Object Analysis and Design: Description of Methods,* John Wiley, 1994.

[10] J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[11] C. Day, *Exclusive Electroproduction of Phi and Lambda(1520),* Thesis, Cornell Univ.,

1978.

[12] J. Bettels, D. R. Myers, *The Pions Graphics System*, CERN-DD/86/6, Mar. 1986.

[13] W. Attwood et al., *The REASON Project*, SLAC-PUB-5242, Apr 1990.

[14] W. Attwood et al., *GISMO: An Object Oriented Program for High-Energy Physics Event Simulation and Reconstruction*, Int. J. Mod. Phys. C3, pp. 459-478, 1992.

[15] A. Dell'Acqua et al., *GEANT4: An Object Oriented toolkit for simulation in HEP*, CERN/DRDC/94-29 Aug. 1994.

[16] BaBar Collaboration (D. Boutigny et al.), *BaBar Technical Design Report,* SLAC-R-95-457, Mar 1995.

[17] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 1.1*, OMG TC Document 91.12.1, Dec. 1991.

[18] D. Quarrie*, Fidle: An IDL to FORTRAN 90 and Eiffel Compiler,* submitted to Computing in High Energy Physics (CHEP) Conference, Sep. 1995.

[19] Object Management Group, *Object Services Architecture*, OMG TC Document 92.8.4, Oct. 1992.

[20] D. Quarrie et al.*, An Object Oriented Run Control Environment for the CEBAF Data Acquisition System*, Proceedings of the Computing in High Energy Physics (CHEP) Conference, 1992

[21] F. Porter and D. Quarrie, *An Analysis Framework and Data Model Prototype for the BaBar Experiment*, submitted to Computing in High Energy Physics (CHEP) Conference, Sep. 1995