The MINOS Off-line Software User's Manual
Version 0.20

November 14, 2000

Contents

Chapter 1

Modules, Algorithms and Candidates

1.1 Introduction

This chapter will tell you how event reconstruction works in our framework and how you write reconstruction code.

At its heart of the reconstruction is simple: input data-like objects are processed to create new output data-like objects which can subsequently be used as input for further processing. In terms of this framework, the processing is performed by Algorithm objects and the outputs are Candidate objects. The framework structures a job into a set of modules each of which is responsible for passing input to a Algorithm and returning the results to the framework. The framework's primary responsibilities are to:-

- Allow the job user to determine what modules get used and in what sequence.
- Allow the job user to configure the Algorithms that get used. In general Algorithms will have sets of constants that control their operation. Defaults set are drawn from the database but the framework allows the job user to fine tune them.
- Prevent memory leaks by maintaining control over all the major objects in the reconstruction process.

- Record a full processing history so that subsequently it would be possible to determine how each Candidate was built and each Algorithm configured.

Your primary responsibility, as an Algorithm developer, is simply to write the single member function that performs reconstruction. Typically individual member functions are procedural so, although set in an OO framework, the task is not so very different from the equivalent job in a Fortran framework.

In this chapter we will focus on reconstruction. However the machinery applies equally well to those phases of analysis that involve the creation of further objects. The remainder of this chapter is divided into the following sections:-

Concepts The concepts that underpin Modules, Handles, CandContexts, MOM, Algorithms and Candidates.

Naming Convention The naming convention used for Modules, Handles, Algorithms and Candidates.

A Simple Example: DigitListModule A walk through a simple reconstruction process.

Working with Modules How to design a Module.

Working with Candidates How to design, create and use Candidates.

Working with Algorithms How to design, create and use Algorithms.

1.2 Concepts

There is a UML diagram in Cand section of the The MINOS Class User Guide tha draws together most of the classes introduced in this section.

1.2.1 Module

A Module is the basic building block of a job. Eventually there will be modules for I/O, calibration, reconstruction, analysis etc. As part of job configuration, the job user tells the framework what Module objects it must create and use.

All modules inherit from JobCModule which is essentially an abstract interface, leaving all the real work to its subclasses. Modules are informed of state changes such as job begin or file end. They have member functions that are called to set them to work. For example, reconstruction modules are called via their Reco method to perform reconstruction on the current event.

1.2.2 Handle

A handle gives you controlled access to an object. If you are familiar with Fortran memory managers such as BOS or ZEBRA, then a handle is a bit like a link. Just as with links you can have multiple handles to the same object. In our framework we use handles for Algorithm and Candidate objects. The primary reasons why we use them are:-

Avoiding Memory Leaks If you directly create any object then it is your responsibility to destroy it when no longer needed, or find another object that is prepared to take on that responsibility! That's why local (stack-based) objects, i.e. objects declared locally within a function are so good; as soon as you leave the function they get destroyed. However this option won't work for event reconstruction! It is not possible to determine, at creation time, how long an individual reconstruction object will live; for later reconstruction may need them even if the creator no longer does. The solution is to use what are called smart pointers. The idea is that any number of smart pointers can point to an object but they must be the only means of access. Once all smart pointers to an object are deleted the object itself can safely be deleted, as it has become inaccessible. What prevents smart pointers from being a source of memory leaks themselves is that they are lightweight and ideally suited to be local objects. However they can also be passed back out of function by value so that its rather like a baton in a relay race; so long as it is passed between functions the object remains alive, but drop the last baton and the object dies.

Some of our handles are smart pointers and from the explanation given above, it follows that they should normally be local objects.

Object Protection Reconstruction objects are initially modifiable but subsequently they become frozen as subsequent reconstruction becomes dependent on them. In our framework the way we prevent objects from being modified is as follows:-

1. First we move all its get and set methods into a companion handle which is made a friend. Now, even if you have an object, you cannot change it, you must have its handle to do that.
2. The handle can determine if a modifying access is allowed and if not either forbid it or create a clone copy of the object which the user can safely modify.

1.2.3 CandContext

A CandContext object is a lightweight object, typically created on the stack that identifies which part of the current event is being used in the role of input. CandContexts are created by Modules and passed to Algorithms so that they know what to operate on.

1.2.4 MOM

MOM is the Minos Object Mapper (O.K. so the name is a bit contrived). Its function is to provide access to all the major components, or fragments, of an event. All I/O works at the fragment level so MOM is really an IO stream manager. Currently MOM is still in prototype design and actually owns all its fragments. A single object reconstruction consists of the following stages:-

1. Ask MOM for an input fragment.
2. Create a CandContext that points to this fragment.
3. Create a new reconstruction object specifying this CandContext and an Algorithm.
4. Hand the resulting object back to MOM.

1.2.5 Algorithm

All Algorithm objects inherit from AlgBase. The role of an Algorithm is to initialise or modify a Candidate i.e. a reconstruction object. Your job is to create a subclass and write the overloaded member function RunAlg which performs this operation.

Conceptually Algorithms are stateless objects although in reality what this means is that during its lifetime changes in its state do not influence the way its RunAlg works. In fact Algorithms have several types of state:-

- Configuration or extrinsic state. This is the set of all constants that control the way it operates. It is called extrinsic because it is not held in the Algorithm object at all but in a companion AlgConfig object. In this way the same Algorithm can be made to operate in multiple ways by associating it with a different AlgConfigs. Currently AlgConfig is just a dummy placeholder but eventually AlgConfig objects will get default state from the database with additional configuration from the job user.

- Implementation or intrinsic state. For example a neural net Algorithm may hold a object structure produced as a result of training. This state does determine how it operates but doesn't change with time.

- Performance state. For example you might want to keep statistics on how well the Algorithm is performing. Strictly speaking this is a second form of intrinsic state but it does not influence operation in any way. Currently the framework has no support for this type of state.

Algorithms are accessed via AlgHandles. This is both to avoid a memory leaks (see section 1.2.2 on handles) and to allow an Algorithm and an AlgConfig to be packaged up together into a single operational unit. The user asks AlgFactory for a specific Algorithm and is handed back an AlgHandle.

1.2.6 Candidate

All reconstruction objects are Candidates and inherit from CandBase.

The name is meant to convey the concept that such objects could in principle be tentative and that alternatives are possible. For example there may be several ways to assign hits to a Track and a Shower object if they overlap. The Candidate package has support machinery to allow multiple reconstructions from the same data, using different configured Algorithms and includes the hooks for equality and overlap testing. Of course this facility will not be used at every stage of reconstruction, but stands ready for use where appropriate. Other built-in support includes:-

- Recording of reconstruction history. Currently dummy.
- Memory leak and object state protection by the use of CandHandles. See the above discussion 1.2.2 of how Candidate objects are protected by devolving their get and set methods to their companion CandHandles.
- I/O. Currently just that provided by ROOT. The ability to distribute Candidate objects over multiple TTrees has yet to be addressed.

All Candidates have a list of daughter Candidates, although strictly speaking the list is of CandHandles to the daughters. The term "daughter" is somewhat misleading as it implies dependence. Sometimes this implication is correct, for example a CandDigitList *owns* a list of CandDigits; if the CandDigitList is destroyed, so are its CandDigits. However later we will want objects, say CandPhysicsEvent that are built from pre-existing CandTrack and CandShower objects but will not own them. Its is better to think of Candidates as being of two types:-

Simple The daughter list is empty. The Candidate's state is to be found in its other data members.

Compound The daughter list is not empty. The Candidate's state is to be found only in its daughters. Of course the daughter list might be empty on occasions, but that is still the only state.

Candidates are created by an associated factory. The user passes in an AlgHandle and a CandContext and receives a CandHandle.

1.3 Naming Convention

In order to create an Xxx object the following classes have to be developed.

CandXxx The reconstruction object.

CandXxxHandle Its handle.

AlgXxx The Algorithm that creates it.

AlgXxxConfig The configuration for AlgXxx.

XxxModule The Module that gets called to create and which puts AlgXxx to work to create the Xxx object.

Often, Xxx is a actually a list of other Candidates, for example CandDigitList is simple a list of CandDigits. In such cases i.e where:-

```
Xxx = YyyList
```

there are a further set of Yyy classes except for YyyModule; the framework only knows about Yyy or lists of Yyy, not both.

1.4 A Simple Example: DigitListModule

1.4.1 Introduction

We shall see all of the above concepts brought into play by studying a simple reconstruction process: the creation of a list of CandDigits from the corresponding RawDigits.

1.4.2 DigitListModule code

At the framework level, what is being created is a CandDigitList and so, by the naming convention above, the Module that is called is DigitListModule.

Examining the source for this we see that most member functions are dummy. for example:-

```
 void DigitListModule::BeginJob()
{
  MSG("JobC", Msg::kInfo) << "DigitListModule::BeginJobn";
}
```

and

```
JobC::Filter_t DigitListModule::AnaNonPhysics(const MomNavigator *mom)
{
  MSG("JobC", Msg::kInfo) << "DigitListModule::AnaNonPhysicsn";
  return JobC::kNoDecision;
}
```

all the real work is done in:-

```
JobC::Filter_t DigitListModule::Reco(MomNavigator *mom);
```

It is passed mom and has to return a JobC::Filter_t. Eventually this return will be used as a way of filtering further processing. For now it is dummy. Stripping out all the comments and printout, the code is remarkably short:-

```
   RawRecord *rr = (RawRecord *) mom->GetFragment("RawRecord");
   AlgHandle adlh = AlgFactory::GetAlgDigitList();
   CandContext cx(this, mom);
   cx.SetDataIn(rr);
   CandDigitListHandle cdlh = CandDigitList::MakeCandidate(adlh, cx);
   CandDigitListHandle *onheap = new CandDigitListHandle(cdlh);
   mom->AdoptFragment(onheap);
   return JobC::kNoDecision;
```

We will now pick this apart, statement by statement.

```
   RawRecord *rr = (RawRecord *) mom->GetFragment("RawRecord");
```

First mom is asked to find the fragment called "RawRecord". It returns a TObject pointer so has to be cast up to a RawRecord pointer.

```
   AlgHandle adlh = AlgFactory::GetAlgDigitList();
```

Next a stack based AlgHandle is filled by asking the AlgFactory for a handle to a AlgDigitList. Note the very typical passing by value when using handles, they are so lightweight that is easier and safer not to use pointers.

Although you don't really need to know, there is nothing very mysterious going on here, if you look in AlgFactory. you will see that GetAlgDigitList is a static method (i.e. is called without an object):-

```
  AlgHandle AlgFactory::GetAlgDigitList()
{
   if (!fsAlgInstanceDigitList)
     fsAlgInstanceDigitList = new AlgDigitList();
   if (!fsAlgConfigDigitList)
     fsAlgConfigDigitList = new AlgConfigDigitList();
   return AlgHandle(*fsAlgInstanceDigitList, *fsAlgConfigDigitList);
}
```

which is simply creating static an AlgDigitList and an AlgConfigDigitList if needed and then creating an AlgHandle from them.

Returning to DigitListModule::Reco,

```
   CandContext cx(this, mom);
   cx.SetDataIn(rr);
```

The first statements creates a stack based CandContext which is passed this, a pointer to the DigitListModule, so that the reconstruction history can record who the creating module and the mom were. The second statement programs the CandContext to point to the RawRecord.

```
   CandDigitListHandle cdlh = CandDigitList::MakeCandidate(adlh, cx);
```

This is where the real action takes place! MakeCandidate is a static method of CandDigitList and it creates a CandDigitList object using the Algorithm handle and the CandContext. It returns a CandDigitListHandle by value. During execution of this statement the AlgDigitList is passed the newly created CandDigitList and the CandContext and uses the input data it has selected to fill out the CandDigitList.

```
   CandDigitListHandle *onheap = new CandDigitListHandle(cdlh);
```

This bit of fancy footwork is needed because we are about to give the new CandDigitList to mom who adopts objects by address. We cannot pass the address of cdlh, as this is on the stack and will shortly disappear. Instead we make a copy of the stack CandDigitListHandle on the heap so that it can be passed to mom.

```
   mom->AdoptFragment(onheap);
```

Now the CandDigitListHandle and associated CandDigitList are passed to mom. The word Adopt is important, its part of our Coding Convention (see NAM-7), and means that mom now owns the CandDigitListHandle and has to worry about its ultimate destruction.

```
return JobC::kNoDecision;
```

Finally the function signals back to the caller. Currently this is dummy hence the kNoDecision value.

Its worth taking one last look at the code. Although it indirectly createda heap based CandDigitList and possibly a AlgDigitList, the code itself only created stack objects for the most part. The one exception was that last CandDigitList which it made mom's problem!

1.4.3 CandDigitList code

Now its time to take a closer look at the all important MakeCandidate:-

```
CandDigitListHandle cdlh = CandDigitList::MakeCandidate(adlh, cx);
```

Looking at CandDigitList member function MakeCandidate:-

```
 CandDigitListHandle CandDigitList::MakeCandidate
                              ( AlgHandle &ah,
                                CandContext &cx)
{
  CandDigitListHandle cdlh;
  new CandDigitList(ah, cdlh, cx);
  return cdlh;
}
```

It creates a stack based CandDigitListHandle and then creates a heap based CandDigitList which is passed the AlgHandle, the local CandDigitListHandle and the CandContext. Passing the CandDigitListHandle to the CandDigitList is the way it gets connected. Finally the CandDigitListHandle gets returned by value.

Now we have reached the constructor for CandDigitList which contains:-

```
 CandDigitList::CandDigitList(AlgHandle &ah, CandHandle &ch,
                                                     CandContext &cx) :
  CandBase(ah)
{
```

```
 // Run Algorithm to construct Candidate
   {                                       // Start of scope.
     CandDigitListHandle cdlh(this);       // cdlh will go out of scope
     ch = cdlh;                            // after setting ch.
   }                                       // End of scope.
   ah.RunAlg(ch, cx);
}
```

The first few statements ensure that the CandBase class is initialised with the AlgHandle and that ch, the CandHandle that has been passed in by reference, gets connected to the new CandDigitList. The crucial line is:-

```
   ah.RunAlg(ch, cx);
```

for this is where the AlgDigitList::RunAlg gets called.

1.4.4 AlgDigitList code

For that we need to look at AlgDigitList. and in particular:-

```
 void AlgDigitList::RunAlg(AlgConfig &ac, CandHandle &ch,CandContext &cx)
{

// Iterate over DigitDataBlock or CandDigitList daughters.
   if (cx.GetDataIn()->IsA()->InheritsFrom("RawRecord")) {

// Iterate over input RawDigit's to fill input CandDigitList.

     AlgHandle ah = AlgFactory::GetAlgDigit();
     CandContext cxx(this, cx.GetMom());

// Iterate over RawRecord's RawDataBlockList to find DigitDataBlock.
     RawRecord *rr = (RawRecord *) (cx.GetDataIn());
     TIter rdbit = rr->RawDataBlockIter();
     TObject *tob;
     while ((tob = rdbit())) {
       if (tob->IsA()->InheritsFrom("DigitDataBlock")) {
         DigitDataBlock *rdb = (DigitDataBlock *) tob;
         TIter rdit = rdb->GetDatumIter();
         RawDigit *rd;
         while ((rd = (RawDigit *) rdit())) {
           cxx.SetDataIn(rd);
           CandHandle cdh = CandDigit::MakeCandidate(ah, cxx);
           ch.AddDaughterLink(cdh);
         }
       }
```

```
    }
  }
  else if (cx.GetDataIn()->IsA()->InheritsFrom("TCollection")) {
    TCollection *cdl = (TCollection *) cx.GetDataIn();
    TIter cdit(cdl);
    CandDigitHandle *cdh;
    while ((cdh = (CandDigitHandle *) cdit())) {
      ch.AddDaughterLink(*cdh);
    }
  }
}
```

The outer if statement:-

```
  if (cx.GetDataIn()->IsA()->InheritsFrom("RawRecord")) {

  ...


  else if (cx.GetDataIn()->IsA()->InheritsFrom("TCollection")) {

  ...

  }
```

demonstrates how a single Algorithm can be made to work on different sorts of input data. The CandContext is sent its GetDataIn message and responds by returning a TObject pointer to it. Then using standard ROOT services the TObject is asked for its TClass which in turn is asked if it InheritsFrom from "RawRecord" or "TCollection" Asking if it InheritsFrom allows the code to work even if the data are subclasses of the required classes. Now, in our case we are interested in the "RawRecord" branch as we know that back in DigitListModule::Reco the CandContext was programmed to point to a RawRecord. Taking this branch to bits:-

```
    AlgHandle ah = AlgFactory::GetAlgDigit();
```

This code should look familiar, its a replay of what happened in DigitListModule::Reco only now we want an AlgHandle for a AlgDigit rather than an AlgDigitList.

```
    CandContext cxx(this, cx.GetMom());
```

This too is familiar. Here a local CandContext object is created that will be used to select an individual RawDigit for AlgDigit to operate on.

Ignoring the details of the inner loops, the remainder of the code is as follows:-

```
// Iterate over RawRecord's RawDataBlockList to find DigitDataBlock.
     RawRecord *rr = (RawRecord *) (cx.GetDataIn());
     TIter rdbit = rr->RawDataBlockIter();
     TObject *tob;
     while ((tob = rdbit())) {
      ...
     }
   }
```

First the RawRecord is retrieved using the supplied CandContext. (i.e. then CandContext associated with the CandDigitList). The RawRecord is asked to provide an iterator over all its RawDataBlocks and this iterator is loop on until exhausted.

Taking the next loop:-

```
     while ((tob = rdbit())) {
     if (tob->IsA()->InheritsFrom("DigitDataBlock")) {
         DigitDataBlock *rdb = (DigitDataBlock *) tob;
         TIter rdit = rdb->GetDatumIter();
         RawDigit *rd;
         while ((rd = (RawDigit *) rdit())) {
          ...
         }
     }
```

If a particular RawDataBlock happens to be, or inherits from, a DigitDataBlock then this block is processed which involves iterating over its individual RawDigits.

The innermost loop starts:-

```
         while ((rd = (RawDigit *) rdit())) {
           cxx.SetDataIn(rd);
           CandHandle cdh = CandDigit::MakeCandidate(ah, cxx);
```

For each RawDigit the local CandDigit is programmed to point to it and the CandDigit factory method MakeCandidate is used to make a CandDigit from the CandContext and the AlgDigit. The factory returns a CandHandle to the CandDigit. Again this should look familiar from DigitListModule::Reco.

```
ch.AddDaughterLink(cdh);
```

In DigitListModule::Reco the CandHandle was given to mom, but in this case the CandDigit is handed, via its handle, to the CandDigitList's daughter list. Note that it is doing this not by sending messages directly to the CandDigitList, but to its handle.

1.4.5 AlgDigit code

All that remains to be analysed is:-

```
CandHandle cdh = CandDigit::MakeCandidate(ah, cxx);
```

Much of this is a replay of CandDigitList::MakeCandidate. A CandDigitHandle and CandDigit get created and connected and during construction of CandDigit, AlgDigit member function RunAlg get called to fill it.

This method is for the most part very straight forward:-

```
 void AlgDigit::RunAlg(AlgConfig &ac, CandHandle &ch, CandContext &cx)
{
   CandDigitHandle &cdh = (CandDigitHandle &) ch;
   RawDigit *digit = (RawDigit *) cx.GetDataIn();

// Use CandDigitHandle methods to set CandDigit member variables.
   cdh.SetChannelId(digit->GetChannel());
   cdh.SetCharge((Float_t) digit->GetADC()); // Set to 100. * p.e. count
   cdh.SetTime((Float_t) digit->GetTime());  // Set to 1. ns time units?

// For a PlexHandle we need a context
   MomNavigator *mom = (MomNavigator *) cx.GetMom();

   if (mom) {
     RawRecord *rawrec = (RawRecord *) mom->GetFragment("RawRecord");
     VldContext vldc = rawrec->GetRawHeader()->GetVldContext();

// Get a PlexHandle for PlexStripEndId to RawChannelId conversions
     PlexHandle ph(&vldc);

     PlexSEIdAltL altlist = ph.ChannelToStripEnds(digit->GetChannel());
     cdh.SetPlexSEIdAltL(altlist);
   }
}
```

This first part simply consists of transfering data from the RawDigit to the CandDigit. Again notice how modification goes via the CandDigitHandle.

The second part involves the creation of a complete list of possible strip ends as determined by the multiplexing for this digit. Here we are getting into a level of detail that is not appropriate for this chapter; the only point to note is that again CandDigit is being modified via its handle.

1.4.6 Summary

To summarise, we have seen how:-

- XxxModule::Reco is used to specify an input (CandContext) and a configured Algorithm (AlgHandle to a AlgXxx) and use them to build an output (CandXxxHandle to a CandXxx) that is then stored in MOM.
- The constructor of the CandXxx uses the AlgXxx::RunAlg to initialise it.
- AlgXxx:RunAlg modifies a new, or existing CandXxx via its CandXxxHandle.

Although this may all look a little daunting, design of Candidate classes will mostly be left to the core group and XxxModules can almost be created starting with an existing one and making a couple of global replaces, which just leaves AlgXxx and in particular AlgXxx:RunAlg, and that last really is the job of the algorithm developer!

George, I need an example of a AlgXxx::RunAlg modifying an existing CandXxx i.e. being call outside the constructor.

1.5 Working with Modules

Modules are informed of states changes such as the begin and end of the job, file and run by calls to the appropraie member function, for example:-

```
void JobCModule::BeginRun();
```

Modules also have member functions Get and Put for I/O and Ana and Reco for analysis and reconstruction processing e.g.:-

To be continued.

1.6 Working with Candidates

To be supplied.

1.6.1 Designing Candidates

To be supplied.

1.6.2 Creating Candidates

To be supplied.

1.6.3 Using Candidates

To be supplied.

1.7 Working with Algorithms

To be supplied.

1.7.1 Designing Algorithms

To be supplied.

1.7.2 Creating Algorithms

To be supplied.

1.7.3 Using Algorithms

To be supplied.

Chapter 2

Message Service

2.1 Introduction

The message service serves two main functions. The first is to provide a convenient way to output formatted numerical data as the C++ "<<" operator is lacking in this regard. The second is to provide a utility to filter and direct text output such as debugging, warning, and error messages. The MINOS C++ coding conventions state that users are to use the message service rather than `cout` and `cerr` directly so that these messages are processed in a uniform way.

2.2 Formatting Numerical Data

The message service provides a simple class to allow users to specify the width and precision of numerical output in much the same way as provided by the C routine `printf`. For example:

```
#include "MessageService/MsgService.h"
#include "MessageService/MsgFormat.h"

int main(void)
{
  int    i = 12345;
  double x = 123.456;
  MsgFormat ifmt("%9d");
```

```
  MsgFormat ffmt("%9.3f");

  // Don't worry about the MSG thing - its covered in the
  // next section. Just think of it as a tricky "cout"
  MSG("XXX",Msg::kInfo) << "[" ifmt(i) << "]" << endl;
  MSG("XXX",Msg::kInfo) << "[" ffmt(x) << "]" << endl;
}
```

will print the number 12345 using a width of 9 characters and 123.456 with a width of 9 characters with a precision of 3 places beyond the decimal point. Output looks like this:

```
[____12345]
[__123.456]
```

where the underscores mark blank spaces. The formating options available are summarized in table 2.1.

-	Left justify in field
+	Always show sign
O	Pad left with zeros
o	Octal format
x,X	Hexadecimal format
f	Fixed format
e,E	Scientific format
g,G	General format

Table 2.1: Format flags for use with class `MsgFormat`.

2.3 Text Messages

2.3.1 Basics

The MessageService also allows users to send text messages to output streams. Each message is associated with a given package (specified by the prefix chosen for the package) and a priority level. The simplest use looks like this:

```
#include "MessageService/MsgService.h"
.
.
.
// The CVSID line is required so that the message service can tag
// error messages with the version number attached to this file
CVSID("$Id: message_service.tex,v 1.1 2000/06/28 22:36:55 messier Exp $");
.
.
.
{
  MSG("Pkg", Msg::kVerbose) << "This is a VERBOSE message\n";
  MSG("Pkg", Msg::kDebug)   << "This is a DEBUG message\n";
  MSG("Pkg", Msg::kInfo)    << "This is an INFO message\n";
  MSG("Pkg", Msg::kWarning) << "This is a WARNING message\n";
  MSG("Pkg", Msg::kError)   << "This is an ERROR message\n";
  MSG("Pkg", Msg::kFatal)   << "This is a FATAL message\n";
}
```

which sends text messages from package `Pkg` with various priorities ranging from lowest to highest. The threshold for printing can be set package by package. For example, if you wanted to see all the `Debug` messages but not all the `Verbose` messages from the package `Pkg` you would set the threshold for the package `Pkg` to `Debug`. The default threshold is `Info`. Eventually the `JobControl` package will provide a simple interface for doing this without recompiling, but for now it must be done manually with a line of code like:

```
MsgService::Instance()->GetStream("Pkg")->SetLogLevel(Msg::kDebug);
```

Note that messages that are below print threshold are not formatted. This means that unprinted messages cause only a negligible performance hit. On my machine (Pentium 350 MHz) the performance hit from 100000 unprinted messages was a total 0.15 seconds.

2.3.2 Advanced Features

The following outlines some of the more advanced features of the `MessageService`. Note that the code fragments shown will eventually be replaced by commands available from the `JobControl` package.

	Priority	Prefix	Date	File	CVS Version	Line #
Verbose	x	x				
Debug	x	x				
Info						
Warning	x	x				
Error	x	x	x	x	x	x
Fatal	x	x	x	x	x	x

Table 2.2: Default information attached to each message.

- **Sending Messages to Multiple Output Streams** The message service allows messages to be send to multiple output streams. For example, if you wanted messages from package `Pkg` to go to both `cout` and a file `PkgInfo.txt` you would use:

```
MsgService::Instance()->
  GetStream("Pkg")->AttachOStream(Msg::kInfo,"PkgInfo.txt");
```

- **Merging of Two Output Streams** The output of two streams can be merged like this:

```
MsgService::Instance()->
  GetStream("PkgA")->AttachOStream(Msg::kInfo,"merge.txt");
MsgService::Instance()->
  GetStream("PkgB")->AttachOStream(Msg::kInfo,"merge.txt");
```

  In this case `Info` messages from both `PkgA` and `PkgB` will be merged in the file `merge.txt`.

- **Concatenation of Two Output Streams** The output of two streams can be concatenated automatically like this:

```
MsgService::Instance()->
  AddCatStream("cat.txt","PkgA",Msg::kInfo);
MsgService::Instance()->
  AddCatStream("cat.txt","PkgB",Msg::kInfo);
```

In this case `Info` messages sent from package `PkgA` and `PkgB` will be concatenated together in the file `cat.txt`.

- **Changing Format** By default, the message service will attach additional information to each message printed. The defaults are listed in table 2.2. To modify the defaults follow these examples:

```
MsgService::Instance->
  GetStream("PkgA")->AddFormat(Msg::kInfo,Msg::kName);
MsgService::Instance->
  GetStream("PkgA")->RemoveFormat(Msg::kInfo,Msg::kName);
MsgService::Instance->
  GetStream("PkgA")->SetFormat(Msg::kInfo,Msg::kName+Msg::kFile);
```

- **Statistics** To view the number of messages printed by package and file use:

```
MsgService::Instance()->PrintStatistics();
```

Chapter 3

Navigation

3.1 Introduction

Moving between related objects of an object structure is called *Navigation*. This chapter describes a set of tools that have been developed to help navigate MINOS object structures. It is divided into the following sections:-

- Section 3.2 provides a full explanation on how the tools are used.
- Section 3.3 explains the need for a set of general purpose navigation tools.
- Section 3.4 outlines the functionality of the tools and sketches the classes that implement them.

Each section has been designed to be read in isolation; so if you are only interested in using the tools without the whys and the wherefores, you can just concentrate on section 3.2 and ignore the others.

3.2 Using the Tools

3.2.1 Introduction

This section has been designed to be read without looking at the preceeding sections of this chapter so some level of repetition is unavoidable. However it concentrates exclusively on explaining how to use the tools without discussing design issues.

The purpose of the tools is to aid navigation of the event and detector object structures. The navigation objects are transient, that is to say they are created as necessary and are not stored permanently. The idea is that designers of the event and detector object structures provide code to create the navigation objects on demand. The end user seems a uniform interface via these navigation objects.

The heart of navigation is the iterator. Its objective is to give access to a set of objects. In the Fortran days you might store a set of items in an array and then process them sequentially using an index. In the OO world the index is replaced by an iterator which can be asked repeatedly to return the next object until there is none left. The advantage is that the you no longer care where or how the objects are stored, all you need is a method to retrieve them.

The navigation tools have two types of iterators:-

- Those that operate on a single set of objects.
- Those that operate on a set of objects that are participate in a relationship with a second set of objects.

The sets are homogeneous i.e. all the members of the set are of the same class. or at least are derived from a common class. In both cases the iterator accesses objects from a single set and provides sequential and random access, sorting and selection services. The only difference between the two types is that the second type has an additional selection facility: to select the sub-set of objects associated with a single object from the other set of the relationship.

3.2.2 Creating the Class Code

Most of the navigation code is generic, that is to say it can support operations on any sets of objects. It does this by accessing all objects via void pointers.

Such pointers are very powerful, they can point at any byte in memory, and so are very dangerous as well! So you never deal directly with these pointers but always use objects that pass the right type of pointer. This means that for any set of objects you wish to navigate there has to be an associated iterator class that returns this type of object. The class is named XxxItr where Xxx denotes the particular class name. So if you have a class called MyClass the tools require the creation of a class called MyClassItr. This code can be created using a C++ macro. Assuming that your class code has the normal .h header and .cxx implementation files, then all you need do is:-

1. In MyClass.h add:-

   ```
   #include "Navigation/XxxItr.h"
   XXXITRDEF(MyClass)
   ```

2. In MyClass.cxx add:-

   ```
   XXXITRIMP(MyClass)
   ```

3. In the LinkDef.h associated with your code add:-

   ```
   #pragma link C++ class MyClassItr;
   #pragma link C++ class MyClassKeyFunc;
   ```

   The purpose of MyClassKeyFunc will be explained when we deal with sorting in section 3.2.5.

Note: the header file XxxItr.h really is typed like that i.e. Xxx is not a placeholder but is to be typed literally.

3.2.3 Creating Iterators for Supported Classes

The main section on creating tools objects will be deferred until section 3.2.10. It may seem strange that the section on using the tool objects precedes the main section that describes their instantiation but, as has been explained above, the tools assist a user to navigate event and detector object structures and that instantiation of the tool objects will be the responsibility of the designers of these object structures. So, for the most part, you will need to refer to the documentation on these classes for information. However the tools know how to build some defaults sets so, to create an iterator over a set, you can always try:-

```
#include "Xxx.h"
XxxItr myItr;
```

where Xxx is the name of the class of objects in the set (e.g. MyClass). To create an iterator of class Xxx in association with a second class Yyy try:-

```
#include "Xxx.h"
XxxItr myItr("Yyy");
```

If the tools don't know where the default set of Xxx objects is, then a null iterator is made. This can be checked by asking its size:-

```
if ( myItr.Size() == 0 ) {
  cout << "Iterator over an empty set" << endl;
}
```

If you are not lucky then you will have to refer to documentation elsewhere on creating iterators or build them yourself as described in section 3.2.10

3.2.4 Sequential Access

Sequential access is the mostly commonly used functionality of an OO iterator. Indeed it is all that some types of iterator can do. Iterators are very popular and several different programming styles have developed. As far as possible the navigation tools support the main ones to help you become familiar with the different styles that you will undoubtedly encounter looking at external code.

Iterating without operator overloading

The first programming style is the most natural for anyone just starting with OO: no iterator operations involve operating overloading.

Suppose you have an XxxItr called myItr and the class Xxx has a Print() member function, then this will print out every one:-

```
while ( myItr.IsValid() ) {
  Xxx* myObj = myItr.Ptr();
  myObj->Print();
  myItr.Next();
}
```

The loop operates while the iterator's member function IsValid() returns true i.e. until there is no Xxx objects to return. The member function Ptr() returns a pointer to the current Xxx which can then be asked to Print() before the iterator is moved forward by asking it to perform its Next() function.

The iterator will return a zero if asked to return a pointer when IsValid() returns false and as 0 is the value of false you can, if you like to cut down lines of code, rewrite the above as:-

```
while ( Xxx* myObj = myItr.Ptr() ) {
  myObj->Print();
  myItr.Next();
}
```

For people who like to be minimalist (which may not be conducive to code transparency), this also works:-

```
myItr.Prev();
while ( Xxx* myObj = myItr.NextPtr() ) myObj->Print();
```

Prev() moves the iterator back (even if it was at the first object) and NextPtr() moves forward and returns the pointer to the object. At any time you can move back to the first object in the set with:-

```
myItr.ResetFirst();
```

As has already been hinted at, you can iterate over the set in reverse order if you want to:-

```
myItr.ResetLast()
while ( myItr.IsValid() ) {
  Xxx* myObj = myItr.Ptr();
  myObj->Print();
  myItr.Prev();
}
```

There is also a PrevPtr() if you need it. If you really know that the iterator is pointing to a valid position you can get it to return the Xxx object itself rather than a pointer, by using the Obj() member function:-

```
while ( myItr.IsValid() ) {
  myItr.Obj().Print();
  myItr.Next();
}
```

The above shows an example of chaining that can confuse you if you are new to C++: myItr.Obj() sends myItr its Obj() message, to which it responds by returning the current object hence the expression myItr.Obj() is equivalent to the current Xxx object which can then be sent its Print() message. We could have achieved the same thing with:-

```
myItr.Ptr()->Print();
```

For completeness there is also NextObj() and PrevObj() but these must be used with extreme caution: having moved, if there is no remaining objects the program will probably crash; its far better to ask for the pointer and then test to see if it is valid i.e. not equal to zero.

Functors Iterators

Functors are objects that have the function operator () overloaded. This means that the object can be called rather like a function. This style is favoured by ROOT for its iterators. The Functor style is also popular for small "single-minded" objects. The member functions of the XxxItr class have been chosen to ensure that any segment of code using ROOT iterators would continue to work if replaced by these navigation iterators. To loop over all objects in the above example with this style you would code:-

```
while ( Xxx* myObj = myItr() ) myObj->Print();
```

myItr() isn't a function call but a call to myItr operator() member function. This does two things:-

1. Returns a pointer to the current object.
2. Moves the iterator on in the current direction (initially forward).

You can flip the sign on the current direction at any time with:-

```
myItr.SetDirection(+1);    //Move forward
myItr.SetDirection(-1);    //Move backward
```

To be helpful, if you ask the iterator to ResetFirst() it will set the default direction is set forward and if you ResetLast() it will set the default direction is backwards. There is also:-

```
myItr.Reset();
```

which moves it back to the beginning relative to the current direction i.e. it is equivalent to ResetFirst() if the default direction is forwards otherwise it is equivalent to ResetLast()

STL like Iterators

STL is the Standard Template Library which comes complete with a set of containers (objects that hold other objects) and iterators that operate on these containers. STL is part if the standard C++ so the STL style effectively becomes the standard for iterators. STL views iterators as if they were pointers to an array of objects, consequently the operators ++ and `--` move the pointer forwards and backwards and the dereference operator * returns the object itself.

The tools only partially support this style. In particular:-

- Only the prefix ++ and `--` operators are supported (++myItr is O.K. but not myItr++).
- In STL you get the start and end limits of the iterator from the container; with the tools you have to use IsValid() to check.

This is how you would iterate:-

```
while ( myItr.IsValid() } {
  (*myItr).Print();
  ++myItr;
}
```

As before ResetFirst() and ResetLast() moves to the start and end of the set.

Its is possible, but probably not advisable, to mix all of the above 3 styles in a single loop!

3.2.5 Sorting

It is often useful to be able to process all the objects in a set in some predefined order. The tools support this and, because of the way they work allow you to sort the set without disturbing the underlying object structure. This is possible because they have their own private copy of the set stored in a NavSet object, and your XxxItr object connects to, and iterates over, this set. This also means you can create multiple iterators simultaneously and have each ordered in its own way.

The way ordering works is that you have to tell the tools how to assign a value to every member of the set and then they sort the set into ascending order based on the value. If you want to processes in descending order you can just process the set backwards of course. The type of value you assign must be a simple type: an integer, floating point, or character string and its value must be derived from public member functions of the objects in the set. To support the notion of a sortable value, the tools have a type of object called a NavKey with the following properties:-

- Can be constructed from the simple types: integer, floating point or character string.
- Can be compared to another NavKey.

Your first job is to write a little function which takes a pointer to your Xxx object and returns a NavKey. As NavKeys can be created directly from simple types, writing this function is simple. Suppose Xxx has the member function Time() that returns a Float_t, then you could create a function, say KeyFromTime:-

```
#include "Navigation/NavKey.h"
static NavKey KeyFromTime( Xxx* obj) { return obj->Time(); }
```

The expression `obj->Time()` is of type Float_t but the compiler wants a NavKey, however it knows how to create a NavKey from a Float_t so creates one. Remember in C and C++ all functions have to be defined at the file level, they cannot be embedded within other functions to hide them. The 'static' keyword at least keeps the name local to the file so the name KeyFromTime won't conflict with the same name defined in other files.

In a similar way, if Xxx has the member function Name() that returns a pointer to Char_t and PlaneNo() that returns Int_t you could write:-

```
static NavKey KeyFromName( Xxx* obj) { return obj->Name(); }
static NavKey KeyFromPlaneNo( Xxx* obj) { return obj->PlaneNo(); }
```

The function names are arbitrary, but it makes sense to be consistent and choose something logical. The above functions just returned values from a single member function but you can be more inventive of course. Suppose you wanted to sort by plane number and within plane number by time and further suppose you know that Time() are always positive and less than 100., then the function you need is:-

```
static NavKey KeyFromPlaneNoTime( Xxx* obj)
  { return obj-PlaneNo()*100. + obj->Time(); }
```

Having written the function, you next job is to get it applied. It might seem reasonable for XxxItr to have a member function that could take your function as an argument and apply it. That would work but is flawed for several reasons:-

- Its not the iterator you want to sort but the NavSet it is connected to. So it is better that you ask the XxxItr for its NavSet and then pass the function to it.
- NavSets are generic: they don't know about Xxx objects so cannot accept your function. Instead you ask your XxxItr for an XxxKeyFunc. This is a little wrapper object that will accept your function and it is this that can be passed to the NavSet. This has the advantage that XxxKeyFunc objects can be "programmed" in other ways besides the method described here. The code for XxxKeyFunc class is created by the same macros that create the XxxItr code; so once you have an XxxItr you also have its companion XxxKeyFunc

This is what the code looks like to sort Xxx by Time:-

```
#include "Navigation/NavKey.h"
#include "Navigation/NavSet.h"

...


// Create an XxxKeyFunc.  Any member function that has
// the word "Create" means that you, as the caller are going
// to be the owner of a new object and will need to look after it!
XxxKeyFunc* myKf = myItr.CreateKeyFunc();

// Program it with the sort function.
mkKf->SetFun(KeyFromTime);

// Ask the iterator for its NavSet and then pass the
// XxxKeyFunC to it. The name "Adopt" means that the NavSet
// now owns the XxxKeyFunc and can delete it anytime it
// pleases e.g. when asked to adopt another.
myItr.GetSet()->AdoptSortKeyFunc(myKf);
```

```
// As you no longer own the XxxKeyFunc the safest thing is
// to clear the pointer so that you are not tempted to
// use it again!
myKf = 0;
```

> *Important: When a NavSet is given a sort function it resets all its iterators by sending them their ResetFirst() message.*

You can change the sort function at any time and as often as you like, but bear in mind that the iterator gets reset each time you do. NavSet applies the ordering at once but if subsequently anyone changes the state of any objects in the set which would result in reordering, this will not be reflected in the order processed. If you know this may have happened you can refresh (see 3.2.9).

You can cancel sorting by passing a null pointer to AdoptSortKeyFunc:-

```
myItr.GetSet()->AdoptSortKeyFunc(0);
```

This won't unsort the set of course but will mean that the NavSet will not attempt to maintain order after operations such as mapping (see 3.2.6)

3.2.6 Selection

General Principles

You can apply several different forms of selection and the result is that you only see the reduced set until the selection is cancelled. The Reset* messages move the iterator to the first or last selected object and the sequential access only returns selected objects. However the Size() member function continues to return the full size of the set; use the SizeSelect() member function to see how many are currently selected. With no selection in force, SizeSelect() is equivalent to Size().

> *Important: When a selection is applied or cancelled on a NavSet the NavSet resets all its iterators by sending them their Reset-First() message.*

If more than one form of selection is in force, they form an AND - objects are selected only if all criteria are satisfied.

Selection by Slicing

Slicing means selecting only those objects whose key value lies within a specified range of values. Suppose you had sorted your Xxx objects by time and now only wanted to process those that lie within the time window 10. to 50. To do this you need to send NavSet its Slice message:-

```
myItr.GetSet()->Slice(10.,50.);
cout << "There are " << myItr.SizeSelect()
     << " selected objects."" << endl;
```

after which myItr will be reset to the start ready for you to process all the Xxx objects in the time range. If the lower and upper limits coincide you can supply a single argument. For example:-

```
myItr.GetSet()->Slice(30.);
```

selects objects whose time is exactly 30. Of course for floating point values, requiring an exact match like this may fail if rounding errors are involved.

Slicing will fail unless:-

- A sort is currently in force.
- The type of sort key (integer, float or character string) matches the type of both arguments of the Slice call.

As with sorting, the slice will not reflect subsequent changes to the objects in the set. Use the refresh message (3.2.9) if necessary.

To cancel slice selection call Slice without arguments:-

```
myItr.GetSet()->Slice();
```

As this is a change in selection, all iterators get reset.

Selection by User Function

Selection by user is very similar to sorting (3.2.5); the user defines a function that returns a NavKey, wraps it in an XxxKeyFunc, and passes it to the NavSet. In this case the function has just to return true or false (non-zero or zero), and the XxxKeyFunc is passed to the NavSet by its AdoptSelectKey-Func message.

Suppose you want to select Xxxs that have either a time less than 50. or a plane number greater that 100, then you could code a selection function like this:-

```
NavKey SelectPlaneNoTime( Xxx* obj)
   { return obj->PlaneNo() > 100. || obj->Time() < 50.; }
```

Having written the function, you apply it like this:-

```
// Create an XxxKeyFunc.  Any member function that has
// the word "Create" means that you, as the caller are going
// to be the owner of a new object and will need to look after it!
XxxKeyFunc* myKf = myItr.CreateKeyFunc();

// Program it with the sort function.
mkKf->SetFun(KeyFromTime);

// Ask the iterator for its NavSet and then pass the
// XxxKeyFunC to it. The name "Adopt" means that the NavSet
// now owns the XxxKeyFunc and can delete it anytime it
// pleases e.g. when asked to adopt another.
myItr.GetSet()->AdoptSelectKeyFunc(myKf);

// As you no longer own the XxxKeyFunc the safest thing is
// to clear the pointer so that you are not tempted to
// use it again!
myKf = 0;
```

Unlike other forms of selection, selection by user function is done “on the fly” i.e. it is applied as part of access. So changes to object state are reflected. This means that the SizeSelect() method is more expensive; NavSet has to loop over the set, applying the selection function to each in turn.

In the normal course of events the object set is not expected to change while navigation is in progress, but if it does while looping over them with an iterator, you may miss some or may retrieve some that subsequently should have been ignored. Navigation of a set that is changing is always a potential problem, see Refreshing Iterators (3.2.9) for a discussion.

To cancel selection by user function pass NavSet a null pointer:-

```
myItr.GetSet()->AdoptSelectKeyFunc(0);
```

Selection by Marking

To be supplied.

Selection by Mapping

Mapping is the process by which a relationship is navigated. Suppose you have two sets A and B that have a relationship, that is to say that each member in one set is associated with zero or more members in the other set. You can select on this association. For example, if you have an iterator over the A set, you can select those that are associated with an individual B set member by passing the NavSet a pointer to the B set member. If that member has no association, or if you pass a pointer to a B that isn't a member of the B set then, as you might expect, you get an empty A sub-set. This is also true unless your iterator was created for a set that is participating in a relationship.

To illustrate the idea suppose your Xxx set is in 1:n relationship with a second set of Yyy objects, that is to say for every Xxx there is 1 or more Yyy but each Yyy has only one Xxx. Further suppose both Xxx and Yyy have a Name() member function. This is how you could loop over all Xxx objects and list the Yyys it is associated with:-

```
//  Include the class headers (which should both use the XXXITRDEF to
//  declare the XxxItr and YyyItr classes).
#include "Xxx.h"
#include "Yyy.h"
#include "Navigation/NavSet.h"

//  Create iterators over both sets in the relationship.
//  This code will only work if this relationship is built in;
//  you will have to see other documentation for other relationships.
XxxItr itrXxx("Yyy");
```

```
YyyItr itrYyy("Xxx");

cout << "The Xxx set has " << itrXxx.Size() << " members" << endl;
cout << "The Yyy set has " << itrYyy.Size() << " members" << endl;

//  Outer loop over all the Xxx set.

while ( itrXxx.IsValid() {

  Xxx* myXxx = itrXxx.Ptr();

//  Subselect Yyy to the association with this Xxx.
  itrYyy.GetSet()->Map(myXxx);

  cout << "Xxx " << myXxx->Name() << " is associated "
       << itrYyy.SizeSelect() << " Yyy objects:-" << end;

// Inner loop over all Yyy associated with this Xxx.

  while ( itrYyy.IsValid() (

    Yyy* myYyy = itrYyy.Ptr();
    cout << "  Yyy " << myYyy->Name() << endl;
    itrYyy.Next();
  }

itrXxx.Next();

}
```

If you know the result of the mapping is a single member subset, it is a bit tedious to perform the map and then pick up the first (and only) set member so, to allow a shortcut, you can pass the Map request directly to your iterator and have it return a pointer to the first set member.

> *Important: Sending a Map message to an XxxItr still results it its NavSet being sent its Map message and, as with other forms of selection, results in all the NavSet's iterators being sent their ResetFirst() message.*

In our example each Yyy is associated with exactly one Xxx so you could write a loop over all Yyy set members and list their association like this:-

```
// Undo any selection by mapping in force on Yyy iterator.
// This will also reset the iterator to the start of its range.
itrYyy.Map(0);

while ( itrYyy.IsValid() ) {

  Yyy* myYyy = itrYyy.Ptr();
  Xxx* myXxx = itrXxx.Map(myYyy);
  cout << "Yyy " << myYyy->Name() << " is associated with  "
       << myXxx->Name() << end;
  itrYyy.Next();
}
```

Note that supplying a null pointer to NavSet's Map function:-

```
itrYyy.Map(0);
```

cancels the mapping selection and, like other changes in selection, resets the iterators.

3.2.7 Random Access

Although their primary function is to give sequential access to a set, iterators can also give random access using the:-

```
Bool_t Jump(Int_t index)
```

member function. If the index is valid i.e. is within range and the corresponding object is selected then the iterator moves to the new position and the function returns kTRUE. Otherwise the iterator remains unchanged (which might also be invalid) and kFALSE is returned. E.g.:-

```
if ( itr.Jump(6) ) cout << "Iterator moved to position 6" << endl;
else               cout << "Position 6 is invalid" << endl;
```

You can also save the current index using GetIndex and restore it later:-

```
Int_t savePos = itr.GetIndex();

...

// Restore position
itr.Jump(savePos);
```

In case you were wondering, there is a SetIndex(Int_t index) and it is equivalent to Jump(Int_t index)

Some points to note about jumping:-

- For a set without any selection at all the valid index range is 0..itr.Size() - 1 just like a C array.
- Sorting and selection by mapping both shuffle the set members; so attempting to use a previously saved index will probably access the wrong member!
- Other forms of selection can make a previously saved index invalid i.e. if its object is no longer selected.

For all of the points given above, jumping should be used with caution. It is probably best used without any form of selection and after any required sorting has been applied.

If you are confident that you know the jump will suceed you can use JumpPtr, which both jumps are retrieves the pointer (cf NextPtr):-

```
cout << "Object at position 6:" << itr.JumpPtr(6)->Print() << endl;
```

3.2.8 Cloning Iterators

There are times when it is useful to have several iterators in operation at once. Having created a single iterator, it is very simple to create a copy by cloning it with the copy constructor:-

```
XxxItr myItr2(myItr);
```

This copies the exact state, including the current position and direction of myItr into the new iterator myItr2. After this they can be operated on independently. Further iterators can be manufactured in the same way. To bring any two back into step just assign one to the other:-

```
// Move myItr to the current position and direction of myItr2.
myItr = myItr2;
```

> *Important: Cloned iterators all operate on the same set; changes in sorting and selection will affect them all in the same way. In particular, changes to sort order and selection will cause all iterators to be reset.*

If you want your iterators to be truly independent; each with its own sort and select then create iterators from scratch rather than by cloning them.

3.2.9 Refresing Iterators

The navigation tools are primarily designed to navigate stable object structures. The idea is that some Monte Carlo or reconstruction process creates the object structure and also provides a method to create an iterator to allow you to explore the results of the process. Attempting to navigate an object structure that is still being changed is a risky business! To require that an iterator do a complete revalidation of its view of a set (as held in the NavSet), each time you used it, would incur an unreasonable overhead. Instead it assumes that nothing has changed and, if it has, this could result in:-

- Missing members, if new objects are added to the set.
- Accessing non-existent members and probably crashing the program, if existing set members are deleted.
- Accessing out of sequence, if object state changes.
- Selecting the wrong members, if object state changes.

Some operations will reflect changes, due to the way that the system is implemented:-

- Selection by user function happens on the fly; the selection function is applied as the object is accessed so reflects the object's current state.
- Mapping rebuilds the subset (or full set in cancelling) from the Lattice and then reapplies any sort and selection so will reflect the current state of the object set.

but it would be unwise to rely on these "accidental" features.

If you really need to navigate a changing object structure because you are building it and want to use the tools, then you can, at any time, issue a refresh request to the NavSet:-

```
myItr.GetSet()->Refresh();
```

This is an expensive operation as it has to rebuild the set, resort, reselect and finally reset all iterators.

3.2.10 Creating the Navigation Objects

Creating Iterators

The navigation tools have two types of iterators:-

- Those that operate on a single set of objects. These iterators are built from TCollection objects, which is the base class for all ROOT containers:-

```
#include "TCollection.h"

TCollection* mySet;  // A ROOT container of Xxx objects.

// Create a iterator over the set.
XxxItr myItr(mySet);

// The iterator can also be created from a ROOT iterator
// using either a pointer to a TIter:-
TIter* myRootItr2 = new TIter(mySet);
XxxItr myItr2(myRootItr2);

// or the TIter itself:-
TIter  myRootItr3(mySet);
XxxItr myItr3(myRootItr3);
```

- Those that operate on a set of objects that are participate in a relationship with a second set of objects. These iterators are built from Lattice objects:-

```
#include "Lattice/Lattice.h"

Lattice* myXxxWithYyy;  // A Lattice holding a relationship:-
                        //   o A set of Xxx objects on the left
                        //   o A set of Yyy objects on the right

// Create an iterator over the Xxx set.
XxxItr itrXxx(myXxxWithYyy, Lattice::kLeft));

/ Create an iterator over the Yyy set.
YyyItr itrYyy(myXxxWithYyy, Lattice::kRight);
```

For information on the creation of Lattice object, see 3.2.10.

Creating Lattices

The concept, and much of the code for the Lattice package, written by Jon Thaler for the CLEO experiment. A Lattice is an object that expresses a relationship between two sets as shown by the following diagram:-

To be continued...

3.3 Background - The need for generic tools

Over the years HEP developed a very successful and simple architecture for Monte Carlo and Reconstruction programs. It consisted of dynamic public data structures describing the events and detector that were operated upon by a set of independent processors. As Fortran 77 does not support dynamic data structures, the HEP community developed memory managers for the purpose, the last of these being ZEBRA. These managers came complete with navigational tools to traverse links of the data structure. Powerful those these managers were they had two serious drawbacks:-

- No type safety: it was the user's responsibility to interpret the meaning, and even the type, of each word in a bank.
- No implementation hiding: any changes made by a bank's designer had to be reflected in changes to all user code that accessed the bank.

Consequently the design and use of banks were very strongly coupled which serious hampered program development. One of the driving forces behind OO methodology was the need to break this coupling between implementation and use. In an OO design the components (objects now instead of banks) control all access and coupling is limited to the public interface the object's support. However this responsibility includes navigation to related objects and this can be unsatisfactory for several reasons:-

- Missing facilities. Navigation is often to a set of objects and then it can be helpful to processes them ordered or selected by some property. Either the objects themselves have to provide this facility or the user has to add code, and neither seems a satisfactory for what is clearly a generic problem.
- Asymmetric navigation. If there is a relationship between objects of class A and class B it is resonable to require navigation in either direction. However this may be difficult or even impossible within the objects themselves. If the B objects are derived from A objects then the A objects may be unaware of the Bs.
- Non-uniform navigation. Relationships between objects that are always read and written together can be by means of standard C++ pointers

as ROOT I/O supports such pointer I/O. However for relationships between objects written to separate records, say between event hits and detector planes, pointers are not adequate and schemes that convert pointers into keys that remain valid under an output input cycle have to be employed. Then navigation is more complex and is dependent on the way the object structure is organised into I/O sub structures.

The purpose of the MINOS navigational tools is to address all of the above. The tools *do not* permanently record relationships; the objects retain that responsibility. Indeed they, or some other software element, has to take on new responsibility: the creation of transient (i.e. created as required during execution) navigation objects. However, once that is done the end users see uniform fully functional navigational interface which is independent of the actual implementation.

Much of the foundation, not to mention significant portions of code, come from the Lattice classes which implement arbitrary n:m relationships by Jon Thaler for the CLEO experiment.

3.4 Function and Form

3.4.1 Functionality

This section outlines the functionality of the tools.

Navigation involves two types of activity.

Navigation within a homogenous set While navigating within a homogeneous set (i.e. a set in which all objects are of the same type) the tools provide the following functionality:-

- Sorting according user supplied function of object state.
- Selection by:-
  - user supplied function.
  - value slice. For the current sorting function the user can supply upper and lower limits in the sorted value.
  - marking. The user can switch on/off a mask associated with each object in the set.

Navigation between pairs of homogenous sets The tools support 1:n navigation, and may eventually support full n:m navigation between pairs of sets.

The tools place no restrictions on the type of object set being navigated and are non-invasive i.e. no changes need to be made to the object's class. This confers an additional advantage as it implies multiple navigational objects operating concurrently. For example, this would mean that the same set could be simultaneously sorted with several different sort criteria.

Finally the tools are type safe, that is to say, although they work for any type of object the user always deals with objects of the correct type and does not have to perform a potentially dangerous cast of a pointer to the right type.

3.4.2 Form

See List of Classes of the Navigation PACKRAT For a description of the classes that implement the tools with the aid of a few simple UML (Unified Modelling Language) diagrams.

Chapter 4

Plexus

> **plex·us** (plĕk'sus) *n. pl.* **-us** or **-us·es.** *Anat.* A structure in the form of a network, esp. of nerves, blood vessels, or lymphatics: *the solar plexus.* [< Lat., p.p. of *plectere,* to plait.]

4.1 Introduction

The `Plex` package provides services for managing the logical connection between elements of the detector above the physical level. Thus it is not concerned with the placement of physical scintillator strips and steel sheets in 3-space, but instead is responsible for providing the navigation from strips to fibers to pixel spots to pixels (anodes) to photomultiplier tube and so forth. It allows reverse navigation as well (e.g. one-to-many as well as many-to-one).

A connection map representing one of the detectors is valid for a particular time and may evolve as components are added or re-configured (e.g. to correct a mis-cabling). Thus instantiating a plexus must involve constructing it for a particular detector at a particular time in its existence – a plexus will be constructed that has a time range that covers the requested time. The underlying mapping information will come from the database.

Diagram 4.1 is a schematic sketch of some of the detector components and how they relate. Figure 4.2 is an illustrative (i.e., non-authoritative) representation of the same concepts as seen from a UML[1] perspective. The logical id and a time stamp can then be used to query the database concerning properties of the components (e.g. serial #, production batch, measurements

[1] Unified Modelling Language, *c.f.* `http://www.rational.com/uml/index.jtmpl`

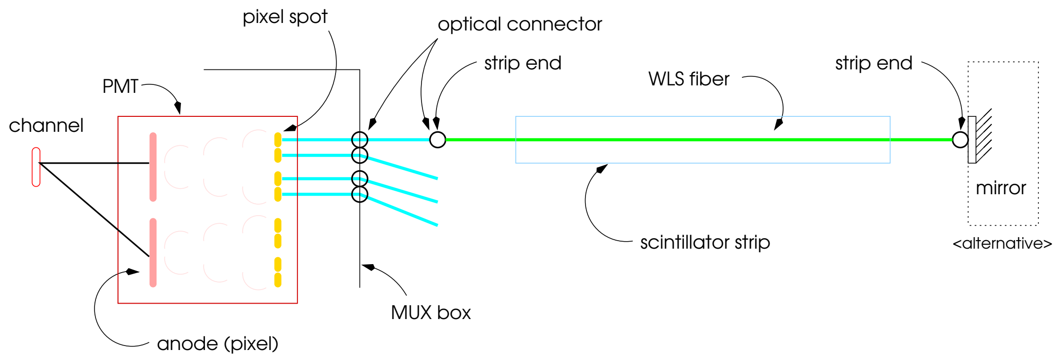

Figure 4.1:

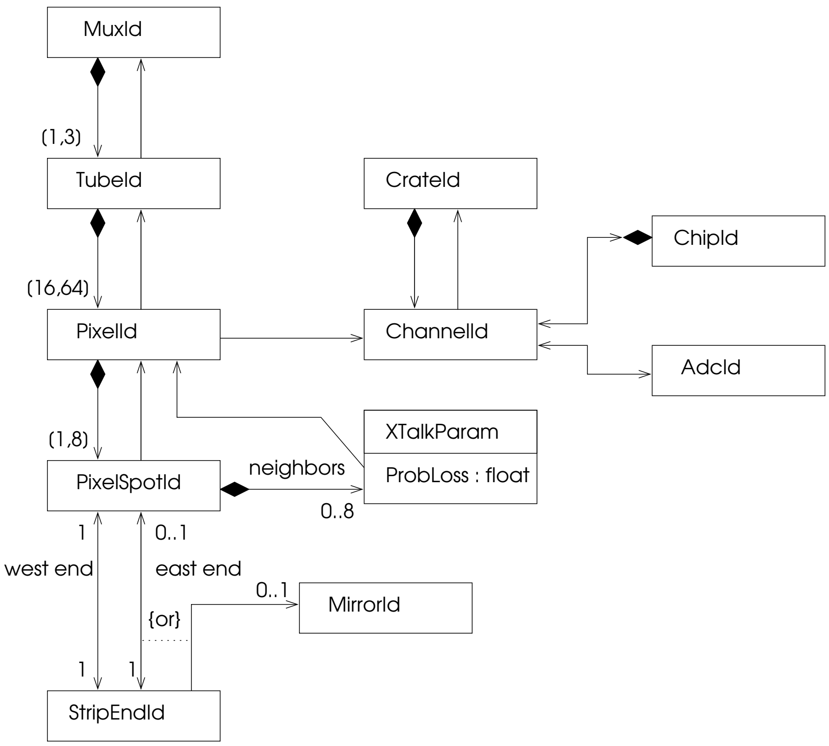

Figure 4.2:

done at the component level).

4.2 Creating a plexus

The user does not work directly with a `Plexus` object, but instead works through a `PlexHandle`. These `PlexHandle`s are lightweight proxy objects that do little work of any substance. During the creation of a `PlexHandle` a *loan pool* of real `Plexus` objects searched for an appropriate object; if no match exists one will be created. The `PlexHandle` then forwards member function calls to the correct `Plexus`. This allows the `PlexHandle` to share the larger underlying collection of information and it lowers the overhead of the creation and deletion of `PlexHandle`s. This approach also allows the `PlexHandles` to be created on the stack and thus avoids a common source of memory leaks.

```
#ifdef KLUDGE
#include "RerootExodus/RerootExodus.h"
#endif
#include "RawData/RawRecord.h"
#include "Plex/PlexHandle.h"

#ifdef KLUDGE
  // create a VldContext object identifying the configuration
  // ...this kludge sneaks a peek at the REROOT/MINFAST objects
  VldContext vldc = RerootExodus::BuildVldContext();

#else
  // somehow acquire a raw record, perhaps:
  RawRecord *rawrec = (RawRecord*) (mom->GetFragment("RawRecord"));

  // create a VldContext object identifying the configuration
  VldContext vldc = rawrec->GetRawHeader()->GetVldContext();
#endif

  // create a PlexHandle (on the stack)
  PlexHandle ph(vldc)

  [....]
  // work with PlexHandle and it gets automagically destroyed
  // when the code block ends, but the next invocation is
  // likely to acquire the same underlying Plexus without
  // significant overhead.
```

4.3 PlexStripEndId

The most frequently encountered identifier is a `PlexStripEndId`. This identifies a logical scintillator *strip-end*; it encapsulates identifiers for the physical detector, plane, strip-in-plane it refers to. It also, when meaningful, identifies which end of the strip in the case of two-sided readout. The components can be extracted via `Get` methods. The most common comparisons to other `PlexStripEndId`s are available via methods in the class:

```
#include "MessageService/MsgService.h"
#include "Plex/PlexStripEndId.h"

  PlexStripEndId seid = [...]
  PlexStripEndId another = [...]

  // decompose PlexStripEndId into components
  DetectorType::Detector_t detector = seid.GetDetector();
  Int_t                    plane    = seid.GetPlane();
  Int_t                    strip    = seid.GetStrip();
  CompassDir::CompassDir_t end      = seid.GetEnd();

  // seid.Print() might yield "|      Far|   2|123|   East| 0|"
  // for the far detector, plane 2, strip 123, east end.

  // output exactly one message
  if ( ! seid.IsSameStrip(another) )
     MSG("XXX",Msg::kInfo) << " not the same (detector,plane,strip)" << endl;

  if ( seid.IsSameStripEnd(another) )
     MSG("XXX",Msg::kInfo) << " same (detector,plane,strip,end)" << endl;

  if ( seid.IsOppositeStripEnd(another) )
     MSG("XXX",Msg::kInfo) << " same (detector,plane,strip) && different (end)" << endl;
  }
```

4.4 PlexSEIdAltL

Initially the `PlexStripEndId` to assign to a readout digitization is not unique because of optical or electrical summing (mistakenly referred to as "multiplexing"). The `PlexSEIdAltL` is a list of `PlexStripEndId` alternatives. Starting with a `RawChannelId` the plexus can construct a list and hand it back to the user for refinement. To facilitate the refinement the `PlexSEIdAltL` manages the list with a weight associated with each alternative. Mechanisms for iterating over the list, setting the associated weights, removing entries and extracting the the `PlexStripEndIds` and weights are provided. The interpretation of the *weight* value is not specified – it could be $-\chi^2$, log-likelihood, etc. Any algorithm must be consistent in its usage; the only expectation is that large values are "better". There are a number of broad general categories of methods for the `PlexSEIdAltL` class: characterization, iteration, weight modification, addition, deletion.

An unexceptional list will normally have a consistent plane number and end enumeration and so `GetPlane()` and `GetEnd()` are generally well defined operations. It is conceivable that a miscabling could mix signals from different planes so that the mapping from `RawChannelId` to a list of `PlexStripEndIds` can't give a unique answer, but currently that isn't handled other than to note, as a printed warning, the exception. The `GetSize()` method returns the number of entries in the list. Iteration and retrieval of values is illustrated in the following code fragment:

```
#include "MessageService/MsgService.h"
#include "Plex/PlexSEIdAltL.h"

   PlexSEIdAltL alt = [...]

   // iterate forward, retrieve each value in turn
   alt.SetFirst();
   while ( alt.IsValid() ) {
      PlexStripEndId seid = alt.GetCurrentSEId()
      Float_t        wgt  = alt.GetCurrentWeight();
      alt.Next();
   }

   // iterate backwards
   alt.SetLast();
   while ( alt.IsValid() ) {
      [...]
      alt.Previous();
   }

   // retrieve values for the entry with the largest weight
   // the iterator cursor is left unchanged
   PlexStripEndId best_seid = alt.GetBestSEId()
   Float_t        best_wgt  = alt.GetBestWeight()
```

The ordering of entries initially follows the order in which they were entered into the list. The list can be sorted by weight value but this must be via an explicit method call so as not to confuse iteration.

One can set all the weights to zero using `ClearWeights()`. The individual weight for the "current" position of the iterator cursor can be adjusted via `SetCurrentWeight()` or `AddToCurrentWeight()`.

There is one method, `AddStripEndId()`, for adding to the list of strip-end, weight pairs but this is the purview of the plexus itself and isn't a normal operation for the user. The user of the list though may wish to delete selected entries and a variety of such facilities are provided:

- `DropZeroWeights()` removes all entries with a weight exactly equal to zero.

- `KeepTopWeights(n,keeporder)` keeps at least the top `n` entries with the largest weights. The resulting list may have more than `n` entries if it would otherwise have to decide between two entries with identical weights. If the boolean flag `keeporder` is true than the final list retains the initial ordering, otherwise it is sorted in descending order so that the first is the entry with the largest weight.

- `DropCurrent()` removes the strip-end, weight pair pointed to by the iterator cursor. The cursor index remains unchanged but the gap is eliminated, effectively moving the cursor on to the next entry. Thus removal via this method should be done in a reverse loop:

```
// iterate *backwards*, remove various values
// testing each and skipping none
alt.SetLast();
while ( alt.IsValid() ) {
   [...]
   if (bad_choice) alt.DropCurrent();
   alt.Previous();
}
```

Chapter 5

Leak Checker

5.1 Introduction

OO programs are dynamic; the objects they consist of are created and destroyed throughout the course of program execution. Objects require memory to hold their data members. The memory for objects created at execution time comes from the stack or the heap. When objects are destroyed it is essential that the memory that they occupied is recovered. This is automatic for stack based objects, but is left to the application code for heap based objects. Failing to destroy objects correctly by using the delete operator results in memory clogged with dead objects, a phenomenon referred to as a *memory leak*

The LeakChecker package is a very simple leak checker facility developed as part of the framework to provide the first level of defense against memory leaks. It can be used to allow basic leak checking to be built into package validation.

> *LeakChecker is not a substitute for a full function leak checker which should still be applied to complete programs.*

To use the checker, probes (C++ macro calls) are added to the code at each object creation and destruction. By default these macros generate nothing and so can safely be a permanent addition to the code. Activation is controlled by a preprocessor option which then generates calls to record object creation and destruction with a LeaLeakChecker object. This object can be used by validation code to query the number of objects created and the number currently active. The information can be supplied globally or for a specific class.

5.2 Inserting Leak Checker probes into code

The best place to insert the probes are into the class constructors and destructors, but this is not possible for external code such as ROOT. In this case the probes have to be added where the new and delete operators are used on them.

5.2.1 Adding probes for MINOS classes

The leak checker macros are defined in:-

```
#include "LeakChecker/Lea.h"
```

The macro LEA_CTOR should be inserted into every constructor and LEA_DTOR into the destructor. These macros deduce the class name from the file name so file names must follow the standard naming convention i.e. the class name followed by .cxx or .h, See below if this is not the case e.g. defining multiple classes in a single file.

> *It is essential that the compiler not be allowed to generate default and copy constructors as calls to these will not be recorded.*

The code for a dummy class supplying both default and copy constructors would look like this:-

```
#include "LeakChecker/Lea.h"

class MyClass {
public:
  MyClass() { LEA_CTOR; }
  MyClass(const MyClass& that) { LEA_CTOR; *this = that }
 ~MyClass() { LEA_DTOR;  }
...

}
```

If you don't want a copy constructor then the trick is simply to declare, but not define, a private one e.g:-

```
private:
  MyClass(const MyClass& that);
}
```

being private nobody else can call it so you don't have to define it.

If your class is not defined in a file with the standard naming convention then you will have to use extended forms LEA_CTOR_NM and LEA_DTOR_NM (where NM = named). These macros take two arguments: the name of the class and the address of the object. Using these macros the above example becomes:-

```
#include "LeakChecker/Lea.h"

class MyClass {
public:
  MyClass() { LEA_CTOR_NM("MyClass",this); }
  MyClass(const MyClass& that)
            { LEA_CTOR_NM("MyClass",this); *this = that }
 ~MyClass() { LEA_DTOR_NM("MyClass",this);  }
...

}
```

5.2.2 Adding probes for foreign classes

The extended macro forms LEA_CTOR_NM and LEA_DTOR_NM can be used with each new and delete operation to record construction and destruction of objects for classes where the modification of the source is not an option. Of course you should not do this for MINOS class objects that have probes built into their constructors and destructors, although if you do, this only means double counting; it won't mask a leak or spuriously indicate one.

Here is sample code recording object creation and destruction:-

```
#include "LeakChecker/Lea.h"

  TheirClass* myObj = new TheirClass(...);
  LEA_CTOR_NM("TheirClass",myObj);

  ...

  delete myObj;
  LEA_DTOR_NM("TheirClass",myObj);
```

```
  myObj = 0;
```

This records heap objects. In principle the same technique could also be used for stack based objects:-

```
#include "LeakChecker/Lea.h"

  {
    TheirClass myObj(...);
    LEA_CTOR_NM("TheirClass",&myObj);

    ...

    LEA_DTOR_NM("TheirClass",&myObj);
  }
```

The DTOR call has to be placed at the point where the object goes out of scope. Not only would this be easy to overlook but will fail if control does not fall through to the end of the compound statement. Further, stack based objects are not a source of leaks and all that is lost by not recording them is that they will not be included in the total creation statistics. For this reason you are advised:-

> *Do not use the macros to record creation and destruction of stack objects.*

5.3 Activating the probes

As explained in the introduction, by default the macro probes are disabled; producing no code at all. Consequently they can be permanent additions to the source. To activate them the code has to be compiled with the preprocessor option -DLEAK_CHECK. This option can be selected when using the standard Makefiles by defining the environmental variable ENV_CXXFLAGS. The following shows how to activate the probes in the MyPackage package:-

```
cd MyPackage
setenv ENV_CXXFLAGS  -DLEAK_CHECKER
gmake clean lib
unsetenv ENV_CXXFLAGS
```

if not using csh, substitute the equivalent commands for setenv and unsetenv.

In this way leak checking can be activated package by package, or even class by class. Note however, probes used for foreign classes can give spurious results if not all new and delete probes are activated. If an activated package gives objects to another unactivated package to delete this will appear as a memory leak. Conversely the inverse will appear as a negative leak. These problems can be avoided by using the leak checker as part of a validation suite as described in the next section.

5.4 Accessing Leak Checker statistics

Activating the probes as described above causes statistics to be collected, class by class on object creation and destruction. To access this information you require the services of the LeaLeakChecker object which can be accessed using it static Instance() method:-

```
#include "LeakChecker/LeaLeakChecker.h"

...

LeaLeakChecker* lea = LeaLeakChecker::Instance();
```

LeaLeakChecker has methods to count the total number of objects created and the number currently active (= created - destroyed):-

```
 UInt_t GetNumCreated(const Char_t* name = 0) const;
 UInt_t GetNumActive(const Char_t* name = 0) const;
```

they take as their argument the name of the required class. If omitted, the sum for all classes is returned. You can reset the counts for any class using:-

```
 void Reset(const Char_t* name = 0);
```

again, omitting the argument means all classes.

A simple summary of the current state of the leak checker system can be sent to the MessageService as follows:-

```
 MSG("Nav",Msg::kInfo) << lea;
```

The leak checker is primarily a first line of defense, to be incorporated into stand-alone package validation. Suppose you have an XxxPackage which has a validation suite contained in the XxxValidate class and executed using the method:-

```
Bool_t RunAllTests();
```

A test function to run it and also perform leak checking could be as follows:-

```
Bool_t TestXxx() {


//  Perform full validation with leak checking and
//  return kTRUE if successful.

#include "LeakChecker/LeaLeakChecker.h"
#include "MessageService/MsgService.h"
#include "XxxPackage/XxxValidate.h"

  Bool_t  ok = kTRUE;

//  Clear leak checker.
  LeaLeakChecker* lea = LeaLeakChecker::Instance();
  lea.Reset();

//  Run validation suite within an inner scope so that
//  all stack based objects are destroyed before checking
//  for leaks.

  {
    XxxValidate v;
    ok = v.RunAllTests();
  }

//  See if leak checker is enabled by checking total created.

  if ( lea->GetNumCreated() == 0 ) {
    MSG("Xxx",Msg::kInfo) << "Warning: Leak checking disabled!!"
  << endl;
  }

//  If it has make sure that there are no objects left.
```

```
  else if ( lea->GetNumActive() ) {
    MSG("Xxx",Msg::kInfo) << "Error, leaks detected! " << lea;
    ok = kFALSE;
  }

  return ok;
}
```

5.5 A final word of warning

The leak checker describe here is not a substitute for a full function leak checker such as Purify for a number of reasons:-

- It relies on the user to manually insert probes in all the right places.
- For MINOS classes the user must ensure the compiler is not left to create default and copy constructors - although its probably best not to rely on these in any case.
- Monitoring foreign class code is clumsy, every new and delete has to be tracked down. Unless all probe points are activated together the leak checker results will be unreliable.
- Although leaks are detected there are no addition facilities to locate the source of the leak. It would be possible to add some basic facilities here if there is the demand.

Despite all of the above, the system should prove useful for any package that can be tested stand-alone. Adding the probes is not particularly onerous and then the package developer can use the system to check for leaks throughout package development on her/his own machine. In this way some of the basic leaks can be eliminated before moving onto integrated system testing using a more powerful checker.

Chapter 6

Job Control

The job control package provides the interface to the data for reconstruction and analysis modules. The package defines the basic unit for reconstruction and analysis, the "JobCModule" and provides several standard modules which allow users to organize the various modules into reconstruction/analysis jobs.

6.1 Job Modules

A job module is the basic unit for interfacing to the data. Reconstruction and analysis algorithms will be called from job modules. Physicist-Programmers will be writing and plugging their code into job modules.

6.1.1 How to Make a Job Module

This section will outline how to make a job module. This section describes how to create a `JobCModule` using the `JobControl/JobCDemoModuleA` as an example. The header file for `JobCDemoModuleA` is shown in Figure 6.1. A sketch of the implementation for this module is shown in Figure 6.2.

User modules must inherit from the base class `JobCModule` as shown in line 019 in the figure. The user module can optionally provide implementations for one or more of the following methods:

- `BeginJob()` - called at the start of a job
- `EndJob()` - called at the end of a job

```
//////////////////////////////////////////////////////////////////////////
// $Id: job_control.tex,v 1.1 2000/08/24 21:24:54 messier Exp $
//
// A dummy module used for testing
//
// messier@huhepl.harvard.edu
//////////////////////////////////////////////////////////////////////////
#ifndef JOBCDEMOMODULEA_H
#define JOBCDEMOMODULEA_H

#ifndef JOBCMODULE_H
#include "JobCModule.h"
#endif

class MomNavigator;
class JobCommand;

//......................................................................

class JobCDemoModuleA : public JobCModule
{
public:
  JobCDemoModuleA();
  ~JobCDemoModuleA();

  void BeginJob();
  void EndJob();
  void BeginFile();
  void EndFile();
  void BeginRun();
  void EndRun();

  void Get(MomNavigator *mom);

  JobC::Filter_t Ana(const MomNavigator *mom);
  JobC::Filter_t Reco(MomNavigator *mom);
  JobC::Filter_t AnaNonPhysics(const MomNavigator *mom);
  JobC::Filter_t RecoNonPhysics(MomNavigator *mom);

  void HandleCommand(JobCommand *command);
  void Help();
  void Report();
  void Reset();
private:
};

#endif // JOBCDEMOMODULEA_H
//////////////////////////////////////////////////////////////////////////
```

Figure 6.1: Header file for a dummy job module.

- `BeginFile()` - called when a new file begins
- `EndFile()` - called when the current file ends
- `BeginRun()` - called when the data run number changes
- `EndRun()` - called when the data run ends.
- `Get()` - called to add data to an event from an I/O stream
- `Put()` - called to write the data to an I/O stream
- `Ana()` - called to perform read-only analysis of data
- `Reco()` - called to perform reconstruction of data
- `AnaNonPhysics()` - called for non-physics data records
- `RecoNonPhysics()` - called for non-physics data records
- `HandleCommand()` - provides a user interface to the module
- `Help()` - prints a help message
- `Report()` - prints a report of the module status
- `Reset()` - set the state of the module back to its "begin run" status.

Note that currently not all of these methods are supported. For example, the `Begin/EndFile` and `Begin/EndRun` methods will not get called in the current framework, nor will `AnaNonPhysics`, `RecoNonPhysics` methods get called. However, `BeginJob`, `EndJob`, `Ana`, `Reco` *are* all currently supported and several examples exist in the framework.

The various methods that look at data records are allowed to make pass/fail decisions on the event by returning values of type `JobC::Filter_t` which has the values `kPassed`, `kFailed`, or `kNoDecision`.

The `HandleCommand` method is passed a pointer to an object of type `JobCommand`. Commands are of the form `/c1/c2/c3/... o1 o2 o3 ...` where `c1, c2, c3, ...` specify the command path (through menus and sub-menus) and `o1, o2, o3...` are the options associated with the command. Module can navigate the command path and options using the methods `JobCommand::PopCmd()`, `JobCommand::PopOpt()`, `JobCommand::PopIntOpt()`, `JobCommand::PopFloatOpt()`. For example,

```
UserModule::HandleCommand(JobCommand *cmd)
{
  string c1 = cmd->PopCmd();
  if (c1 == "Set") {
    string opt = cmd->PopOpt();
    if      (opt == "Foo") fFoo = cmd->PopIntOpt();
    else if (opt == "Bar") fBar = cmd->PopFloatOpt();
    else { ... print warning... }
  }
  else if (c1 == "Draw") {
    string opt = cmd->PopOpt();
    if      (opt == "Top")  this->DrawTopView();
    else if (opt == "Side") this->DrawSideView();
    else { ... print warning... }
  }
}
```

handles the commands:

```
/UserModule/Set Foo 12.0
/UserModule/Set Bar 2
/UserModule/Draw Top
/UserModule/Draw Side
```

The implementation file has the responsibility of declaring the existence of the job module to the framework. This is done by the lines 009 and lines 019-021 in Figure 6.2. These lines use the macro `JOBMODULE` from the include file `JobCModuleRegistry.h` to declare the existence of the job module. The three arguments of the macro are 1) The exact class name of the module, 2) a "common" name for the module which users will use to refer to the module and 3) a short description of what the module does.

6.2 User Interface

6.2.1 Basics

The job controller provides an interface to ROOT. This looks like:

```
root [0] JobCROOT *jc = JobCROOT::Instance();
root [1] (*jc)("Help")
```

```
////////////////////////////////////////////////////////////////////////
// $Id: job_control.tex,v 1.1 2000/08/24 21:24:54 messier Exp $
//
// Empty implementation of a job module
//
// messier@huhepl.harvard.edu
////////////////////////////////////////////////////////////////////////
#include "JobCDemoModuleA.h"
#include "MessageService/MsgService.h"
#include "JobCModuleRegistry.h" // For JOBMODULE macro

//......................................................................

CVSID("$Id: job_control.tex,v 1.1 2000/08/24 21:24:54 messier Exp $");

// Declare this module to JobControl. Arguments are:
//  (1) The class name
//  (2) The human-readable name
//  (3) A short, human-readable description of what the module does
JOBMODULE(JobCDemoModuleA,
          "DemoModuleA",
          "A dummy module used for testing the JobControl package");

//......................................................................

...
```

Figure 6.2: Implementation file for a dummy job module.

In general job control commands are of the format:

`/[path]/[module]/[command] [option1] [option2] ...`

where

- `path` is a name of a job path
- `module` is a name of a job module
- `command` is the name of a command to send to the module
- `option` is a possible option for the command

Commands are in general case sensitive. For help with a specific job module try: `(*jc)("Help [module1] [module2] ... ")`.

In addition to `Help` the job controller provides the command `Echo On/Off` to turn echoing of commands sent to the job controller on or off. For batch processing this would allow the configuration commands to be echoed to a log file.

6.2.2 Message Service Configuration

`Msg` is a module for configuring the message service. It can be used to set log levels, direct output, set formats etc. for the various package streams. For example, to lower the print threshold for the stream `Trk` use: `/Msg/SetLevel Trk Debug`

Possible commands are:

- `/Msg/SetLevel <stream> <level>` Set the log level for `stream` to `level` where `level` is Verbose,Debug,Info,Error,Warning,Fatal
- `/Msg/Attach <stream>::<loglevel> <ostream>` Attach the output stream `ostream` to the list of output streams messages sent to `stream` of type `loglevel` get send to. `ostream` can be `cout`, `cerr`, `clog`, or a file name. `loglevel` can be Verbose, Debug, Info, Warning, Error, Fatal, or All.
- `/Msg/Format <stream>::<loglevel> <Add/Remove> <format>` Add/Remove additional tags to messages. Possibilities are: Priority, Time, File, CVSId, and Line.

6.2.3 Job Paths

Path is a module for creating and holding the job paths and job modules used by your job. Possible commands are:

- `/Path/Add <path> <module>::<method>`
- `/Path/Add <path> <module>::<method> <position>` Add module's `<module>` `<method>` to the path `path` at the `<position>` (1,2,3...). `<position>` is optional.
- `/Path/Create <path>`
- `/Path/Create <path> <mod1>::<meth1> <mod2>::<meth2> ...` Create a path named `path` and add nodes specified by the module and method pairs `<mod>::<meth>` to it.
- `/Path/Delete <path>` Delete the path named `path`
- `/Path/Remove <module>`
- `/Path/Remove <module>::<method>` Remove the node `module::method` from a path If `method` is not specified all nodes using `module` are removed.

Once a path is created use these commands to perform operations on it:

- `/MyPath/Run` - run all records through the path
- `/MyPath/Run Nin/Npass/Nfail [n]` - run records through the path until `n` records have been input, pass all modules in the path, or fail one or more modules in the path.
- `/MyPath/Skip` - skip some records ahead
- `/MyPath/Report` - report on statistics for this path
- `/MyPath/Reset` - reset this path
- `/MyPath/Back` - back the path up some number of records

6.2.4 Sample Session

The following shows a sample session which creates a job path and runs one event.

```
\footnotesize
root [0] JobCROOT *jc = JobCROOT::Instance();
root [1] (*jc)("/Path/Create MINFAST "
         "RerootToRawDataModule::Get "
         "RerootToRawDataModule::Reco");

MINFAST (Nin=0 Npass=0 Nfail=0)
  1) RerootToRawDataModule::Get (Nin=0 Npass=0 Nfail=0) (t = 0 s )
  2) RerootToRawDataModule::Reco (Nin=0 Npass=0 Nfail=0) (t = 0 s )

root [2] (*jc)("/MINFAST/Run Nin 1");
root [3] (*jc)("/MINFAST/Run Nin 1");
Processed 1 records:
MINFAST (Nin=1 Npass=1 Nfail=0)
  1) RerootToRawDataModule::Get (Nin=1 Npass=1 Nfail=0) (t = 0.01 s )
  2) RerootToRawDataModule::Reco (Nin=1 Npass=1 Nfail=0) (t = 0.24 s )

root [4] (*jc)("/Path/Add MINFAST DemoModuleA::Reco");

MINFAST (Nin=1 Npass=1 Nfail=0)
  1) RerootToRawDataModule::Get (Nin=1 Npass=1 Nfail=0) (t = 0.01 s )
  2) RerootToRawDataModule::Reco (Nin=1 Npass=1 Nfail=0) (t = 0.24 s )
  3) DemoModuleA::Reco (Nin=0 Npass=0 Nfail=0) (t = 0 s )
```

For an example of use within the MINFAST package please see the `jc_init.C` and `jc_next.C` macros.

Chapter 7

Dependency

As the software system becomes increasingly complex managing the dependencies among and within packages will become increasingly important. The Dependency package is a set of tools to analyzing the dependency structure of the system.

7.1 How to make the list of dependencies?

To make the list of dependencies for the packages in the repository setup your MINFAST path as you normally would then type:

```
gmake dep
```

in the Dependency package area. This will produce a set of files:

- `dep.err` is the log of errors and warnings encountered. Most of these can be ignored since the script does not try to trace all the dependencies through packages external to our software framework. However, searching this file for "Warning" will show all the circular dependencies encountered in the system.
- `all.dep` this full list of package dependencies. See below for tips on interpreting the contents of this file.
- `package.dep` where `package` is the name of a specific package in the system. This file will contain the list of dependencies within a particular package.

The dependency report files have the general format:

```
level. package dep-level dep-package
```

For a package `package level` is the layer of this package in the hierarchy. Packages with high levels depend on many other packages. For example, packages at level 0 depend only on external packages, packages at level 1 depend on things at level 0 and below, etc. The package `package` depends on the package `dep-package` which is found at layer `dep-level` in the hierarchy. Packages flagged with `<n>` (`n` is a numeric label) are circularly dependent on other packages/classes flagged with the same numbers.

7.2 Why is this useful?

I have found this useful in several ways:

- Circular dependencies among packages prevents them from being tested and developed independently from one another and greatly increases the complexity of learning, maintaining, testing, and debugging the software system.
- Within a package, a circular dependency is often the results of a weak design. If two classes need functions located within each other perhaps that's telling you that those functions should be pushed up or down into a third class. Perhaps those two classes are so closely related that you really have one class and not two?
- By laying out the dependency tree this helps identify those classes which can be tested in as unit ("sub-package"). Testing code on many levels helps bugs and other problems to be located quickly. For example, suppose you have five classes that have the dependencies diagramed in Figure 7.1.

The classes in Figure 7.1 should be tested in the following combinations:

- A alone,
- B alone,
- C alone,

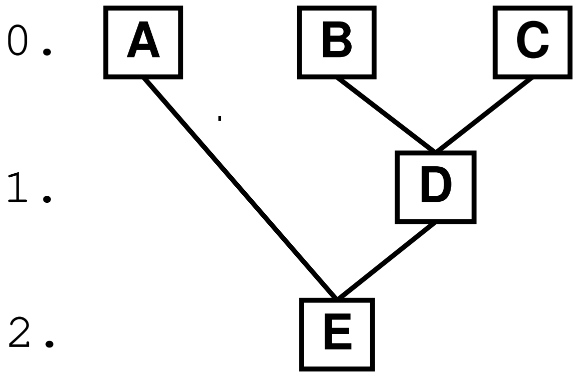

Figure 7.1: A sample set of dependencies. These five classes can be tested in units of A, B, C, DBC, EADBC.

- DBC together
- E-A-(DBC) together

If the classes are tested this way problems can be quickly located. Eg. if the test of E-C-D-A-B fails one can look at the tests of A and D-B-C for the problem. If the problem is not found there then one should look for the error in class E.