# Table of Contents

## Chapter 1
**Introduction**

1.1 Purpose of the document ........................................... 9  
1.2 Athena and GAUDI .................................................. 9  
1.2.1 Document organization ......................................... 10  
1.3 Conventions ......................................................... 10  
1.3.1 Units ................................................................... 10  
1.3.2 Coding Conventions .............................................. 10  
1.3.3 Naming Conventions .............................................. 10  
1.3.4 Conventions of this document ............................... 10  
1.4 Release Notes ......................................................... 11  
1.5 Reporting Problems .................................................. 11  
1.6 User Feedback ........................................................ 11

## Chapter 2
**The framework architecture** ............................................. 13  
2.1 Overview ................................................................... 13  
2.2 Why architecture? ...................................................... 13  
2.3 Data versus code ...................................................... 14  
2.4 Main components ..................................................... 15

## Chapter 3
**Release notes** ............................................................ 19  
3.1 Overview ................................................................... 19  
3.2 New Functionality ..................................................... 19  
3.3 Changes that are not backwards compatible .................. 19  
3.4 Changed dependencies on external software ................ 19  
3.5 Bugs Fixed .............................................................. 19  
3.6 Known Bugs ............................................................ 20

## Chapter 4
**Establishing a development environment** ......................... 21  
4.1 Overview ................................................................... 21  
4.2 Establishing a login environment ............................... 21  
4.2.1 Commands to establish a bourne-shell or varient login environment ........................................... 21  
4.2.2 Commands to establish a c-shell or varient login environment ........................................... 22  
4.3 Using SRT to checkout ATLAS software packages ....... 22

## Chapter 5
Writing algorithms ................................................................. 23
  5.1 Overview ...................................................................... 23
  5.2 Algorithm base class ......................................................... 23
  5.3 Derived algorithm classes ................................................... 25
    5.3.1 Creation (and algorithm factories) ......................... 26
    5.3.2 Declaring properties ................................................. 26
    5.3.3 Implementing IAlgorithm ........................................ 27
  5.4 Nesting algorithms .......................................................... 29
  5.5 Algorithm sequences, branches and filters ......................... 30
    5.5.1 Filtering example ................................................... 31

Chapter 6
Scripting ................................................................................. 35
  6.1 Overview ...................................................................... 35
  6.2 Python scripting service ..................................................... 35
  6.3 Python overview .............................................................. 35
  6.4 How to enable Python scripting .......................................... 36
    6.4.1 Using a Python script for configuration and control .... 36
    6.4.2 Using a job options text file for configuration with a Python interactive shell 36
  6.5 Prototype functionality ....................................................... 37
  6.6 Property manipulation ....................................................... 38
  6.7 Synchronization between Python and Athena ...................... 39
  6.8 Controlling job execution .................................................. 40

Chapter 7
Accessing data ......................................................................... 43
  7.1 Overview ...................................................................... 43
  7.2 Using the data stores ......................................................... 43
  7.3 Using data objects ........................................................... 45
  7.4 Object containers ............................................................. 47
  7.5 Using object containers ...................................................... 48
  7.6 Data access checklist ......................................................... 50
  7.7 Defining new data types ..................................................... 50
  7.8 The SmartDataPtr/SmartDataLocator utilities ..................... 51
    7.8.1 Using SmartDataPtr/SmartDataLocator objects ........ 51
  7.9 Smart references and Smart reference vectors ...................... 52
  7.10 Saving data to a persistent store ........................................ 53

Chapter 8
StoreGate - the event data access model ..................................... 55
  8.1 Overview ...................................................................... 55
  8.2 The StoreGate design ......................................................... 55
    8.2.1 System Features and Entities ..................................... 56
Table of Contents

8.2.2 Characteristics of the Transient Data Store ........................................ 57
8.3 The StoreGate Prototype ................................................................. 58
8.4 Creating a DataObject. ....................................................................... 59
  8.4.1 Creating a single DataObject ......................................................... 59
  8.4.2 Creating a Collection of ContainedObjects ................................. 60
  8.4.3 Creating a ContainedObject ......................................................... 61
8.5 Recording a DataObject. ................................................................. 62
  8.5.1 Recording DataObjects without keys ........................................... 62
  8.5.2 Recording DataObjects with keys ............................................... 63
8.6 Retrieving a DataObject ................................................................. 63
  8.6.1 Retrieving DataObjects without a key ......................................... 64
  8.6.2 Retrieving a keyed DataObject .................................................... 65
  8.6.3 Retrieving all DataObjects of a given type ............................... 66
8.7 Store Access Policy ...................................................................... 67

Chapter 9
Data dictionary ...................................................................................... 69
  9.1 Overview ....................................................................................... 69
     9.1.1 Definition of Terms ................................................................. 69
     9.1.2 Roles of a Data Dictionary (DD) ............................................... 70
     9.1.3 Implementation Strategy ......................................................... 71
     9.1.4 Data Dictionary Language ....................................................... 71
     9.1.5 Code Generation ................................................................. 72
     9.1.6 Data Dictionary Design ......................................................... 72
     9.1.7 Time Line & Milestones ......................................................... 73
     9.1.8 Bibliography ......................................................................... 73

Chapter 10
Detector Description ........................................................................... 75
  10.1 Overview ..................................................................................... 75

Chapter 11
Histogram facilities ............................................................................. 77
  11.1 Overview ..................................................................................... 77
  11.2 The Histogram service ............................................................... 78
  11.3 Using histograms and the histogram service ............................. 79
  11.4 Persistent storage of histograms ................................................ 80
     11.4.1 HBOOK persistency ............................................................. 80
     11.4.2 ROOT persistency ............................................................... 80

Chapter 12
N-tuple and Event Collection facilities ........................................... 83
  12.1 Overview ..................................................................................... 83
## 12.2 N-tuples and the N-tuple Service

- **12.2.1 Access to the N-tuple Service from an Algorithm.**
- **12.2.2 Using the N-tuple Service.**

## 12.3 Event Collections

- **12.3.1 Writing Event Collections.**
- **12.3.2 Reading Events using Event Collections.**

## 12.4 Interactive Analysis using N-tuples

- **12.4.1 HBOOK.**
- **12.4.2 ROOT.**

---

### Chapter 13

**Framework services**

- **13.1 Overview.**
- **13.2 Requesting and accessing services.**
- **13.3 The Job Options Service.**
  - **13.3.1 Algorithm, Tool and Service Properties.**
  - **13.3.2 Job options file format.**
  - **13.3.3 Example.**
- **13.4 The Standard Message Service.**
  - **13.4.1 The MsgStream utility.**
- **13.5 The Particle Properties Service.**
  - **13.5.1 Initialising and Accessing the Service.**
  - **13.5.2 Service Properties.**
  - **13.5.3 Service Interface.**
  - **13.5.4 Examples.**
- **13.6 The Chrono & Stat service.**
  - **13.6.1 Code profiling.**
  - **13.6.2 Statistical monitoring.**
  - **13.6.3 Chrono and Stat helper classes.**
  - **13.6.4 Performance considerations.**
- **13.7 The Auditor Service.**
  - **13.7.1 Enabling the Auditor Service and specifying the enabled Auditors.**
  - **13.7.2 Overriding the default Algorithm monitoring.**
  - **13.7.3 Implementing new Auditors.**
- **13.8 The Random Numbers Service.**
- **13.9 The Incident Service.**
  - **13.9.1 Known Incidents.**
- **13.10 Developing new services.**
  - **13.10.1 The Service base class.**
  - **13.10.2 Implementation details.**

### Chapter 14

**Tools and ToolSvc**

- **14.1 Overview.**
14.2 Tools and Services ................................................................. 123
  14.2.1 “Private” and “Shared” Tools ........................................ 124
  14.2.2 The Tool classes ........................................................ 124
14.3 The ToolSvc ................................................................. 129
  14.3.1 Retrieval of tools via the IToolSvc interface .................... 130
14.4 GaudiTools ................................................................. 131
  14.4.1 Associators ............................................................. 131

Chapter 15
Converters ................................................................. 137
  15.1 Overview .................................................................. 137
  15.2 Persistency converters ................................................. 137
  15.3 Collaborators in the conversion process .............................. 138
  15.4 The conversion process ............................................... 140
  15.5 Converter implementation - general considerations .............. 142
  15.6 Storing Data using the ROOT I/O Engine ......................... 143
  15.7 The Conversion from Transient Objects to ROOT Objects ........ 144
    15.7.1 Non Identifiable Objects ......................................... 145
    15.8 Storing Data using other I/O Engines ......................... 146

Chapter 16
Visualization ............................................................... 147
  16.1 Overview ................................................................. 147

Chapter 17
Physical design issues ..................................................... 149
  17.1 Overview .................................................................. 149
  17.2 Accessing the kernel GAUDI framework .............................. 149
    17.2.1 The GaudiInterface Package .................................... 149
  17.3 Framework libraries ................................................... 150
    17.3.1 Component libraries .............................................. 150
    17.3.2 Linker Libraries .................................................. 153
    17.3.3 Dual purpose libraries and library strategy .......... 154
    17.3.4 Building the different types of libraries .......... 154
    17.3.5 Linking FORTRAN code ..................................... 154

Chapter 18
Framework packages, interfaces and libraries ......................... 157
  18.1 Overview .................................................................. 157
  18.2 Gaudi Package Structure .............................................. 157
    18.2.1 Gaudi Package Layout ........................................... 158
    18.2.2 Packaging Guidelines .......................................... 159
  18.3 Interfaces in Gaudi ...................................................... 160
18.3.1 Interface ID ........................................ 161
18.3.2 Query Interface .................................... 162
18.4 Libraries in Gaudi ..................................... 162
18.4.1 Component libraries ................................ 162
18.4.2 Linker libraries .................................... 165
18.4.3 Library strategy and dual purpose libraries. ... 166
18.4.4 Building and linking with the libraries .......... 166
18.4.5 Linking FORTRAN code ......................... 168

Chapter 19
Analysis utilities ....................................... 169
19.1 Overview.............................................. 169
19.2 CLHEP ............................................... 169
19.3 HTL .................................................. 169
19.4 NAG C ............................................... 170
19.5 ROOT ............................................... 170

Appendix A
References ............................................... 171

Appendix D
Installation guide ...................................... 187
19.6 Overview.............................................. 187
19.6.1 Acknowledgements................................. 187
19.6.2 External Packages ................................. 188
19.6.3 Installing CLHEP ................................. 188
19.6.4 Installing HTL ................................... 188
19.6.5 Installing CERNLIB .............................. 188
19.6.6 Installing ROOT ................................. 189
19.6.7 Installing Qt ...................................... 189
19.6.8 Installing CMT ................................. 189
19.6.9 Installing GAUDI ............................... 189
19.6.10 Installing the ATLAS release .................. 189
Chapter 1
Introduction

1.1 Purpose of the document

This document is intended for developers of the Athena control framework. Athena is based upon the GAUDI architecture that was originally developed by LHCb, but which is now a joint development project. This document, together with other information about Athena, is available online at:

http://web1.cern.ch/Atlas/GROUPS/SOFTWARE/00/architecture

This version of the Athena corresponds to Athena release 2.0.0. This is based upon ATLAS GAUDI version 0.7.2, which itself is based upon GAUDI version 7 with some patches.

1.2 Athena and GAUDI

As mentioned above Athena is a control framework that represents a concrete implementation of an underlying architecture. The architecture describes the abstractions or components and how they interact with each other. The architecture underlying Athena is the GAUDI architecture originally developed by LHCb. This architecture has been extended through collaboration with ATLAS, and an experiment neutral or kernel implementation, also called GAUDI, has been created. Athena is then the sum of this kernel framework, together with ATLAS-specific enhancements. The latter include the event data model and event generator framework.

The collaboration between LHCb and ATLAS is in the process of being extended to allow other experiments to also contribute new architectural concepts and concrete implementations to the kernel GAUDI framework. It is expected that implementation developed originally for a particular experiment will be adopted as being generic and will be migrated into the kernel. This has already happened with,
for example, the concepts of auditors, the sequencer and the ROOT histogram and ntuple persistency service.

For the remainder of this document the name Athena is used to refer to the framework and the name GAUDI is used to refer to the architecture upon which this framework is based.

1.2.1 Document organization

The document is organized as follows:

1.3 Conventions

1.3.1 Units

This section is blank for now.

1.3.2 Coding Conventions

This section is blank for now.

1.3.3 Naming Conventions

This section is blank for now.

1.3.4 Conventions of this document

Angle brackets are used in two contexts. To avoid confusion we outline the difference with an example.

The definition of a templated class uses angle brackets. These are required by the C++ syntax, so in the instantiation of a templated class the angle brackets are retained:

```
AlgFactory<UserDefinedAlgorithm> s_factory;
```
This is to be contrasted with the use of angle brackets to denote “replacement” such as in the specification of the string:

```
"<concreteAlgorithmType>/<algorithmName>"
```

which implies that the string should look like:

```
"EmptyAlgorithm/Empty"
```

Hopefully what is intended will be clear from the context.

### 1.4 Release Notes

Although this document is kept as up to date as possible, Athena users should refer to the release notes that accompany each ATLAS software release for any information that is specific to that release. The release notes are kept in the `offline/Control/ReleaseNotes.txt` file.

### 1.5 Reporting Problems

Eventually ATLAS will use the Remedy bug reporting system for reporting and tracking of problems. Until this is available, users should report problems to the ATLAS Architecture mailing list at `atlas-sw-architecture@atlas-lb.cern.ch`.

### 1.6 User Feedback

Feedback on this User Guide, or any other aspects of the documentation for Athena, should also be sent to the ATLAS Architecture mailing list.
Chapter 2
The framework architecture

2.1 Overview

In this chapter we outline some of the main features of the Gaudi architecture. A (more) complete view of the architecture, along with a discussion of the main design choices and the reasons for these choices may be found in reference [1].

2.2 Why architecture?

The basic “requirement” of the physicists is a set of programs for doing event simulation, reconstruction, visualisation, etc. and a set of tools which facilitate the writing of analysis programs. Additionally a physicist wants something that is easy to use and (though he or she may claim otherwise) is extremely flexible. The purpose of the Gaudi application framework is to provide software which fulfils these requirements, but which additionally addresses a larger set of requirements, including the use of some of the software online.

If the software is to be easy to use it must require a limited amount of learning on the part of the user. In particular, once learned there should be no need to re-learn just because technology has moved on (you do not need to re-take your licence every time you buy a new car). Thus one of the principal design goals was to insulate users (physicist developers and physicist analysists) from irrelevant details such as what software libraries we use for data I/O, or for graphics. We have done this by developing an architecture. An architecture consists of the specification of a number of components and their interactions with each other. A component is a “block” of software which has a well specified interface and functionality. An interface is a collection of methods along with a statement of what each method actually does, i.e. its functionality.
We may summarise the main benefits we gain from this approach:

**Flexibility**  This approach gives flexibility because components may be plugged together in different ways to perform different tasks.

**Simplicity**  Software for using, for example, an object data base is in general fairly complex and time consuming to learn. Most of the detail is of little interest to someone who just wants to read data or store results. A “data access” component would have an interface which provided to the user only the required functionality. Additionally the interface would be the same independently of the underlying storage technology.

**Robustness**  As stated above a component can hide the underlying technology. As well as offering simplicity, this has the additional advantage that the underlying technology may be changed without the user even needing to know.

It is intended that almost all software written by physicists, whether for event generation, reconstruction or analysis, will be in the form of specialisations of a few specific components. Here, specialisation means taking a standard component and adding to its functionality while keeping the interface the same. Within the application framework this is done by deriving new classes from one of the base classes:

- DataObject
- Algorithm
- Converter

In this chapter we will briefly consider the first two of these components and in particular the subject of the “separation” of data and algorithms. They will be covered in more depth in chapters 5 and 7. The third base class, Converter, exists more for technical necessity than anything else and will be discussed in Chapter 15. Following this we give a brief outline of the main components that a physicist developer will come into contact with.

### 2.3 Data versus code

Broadly speaking, tasks such as physics analysis and event reconstruction consist of the manipulation of mathematical or physical quantities: points, vectors, matrices, hits, momenta, etc., by algorithms which are generally specified in terms of equations and natural language. The mapping of this type of task into a programming language such as FORTRAN is very natural, since there is a very clear distinction between “data” and “code”. Data consists of variables such as:

```plaintext
integer n
real p(3)
```

and code which may consist of a simple statement or a set of statements collected together into a function or procedure:

```plaintext
real function innerProduct(p1, p2)
real p1(3), p2(3)
```
innerProduct = p1(1)*p2(1) + p1(2)*p2(2) + p1(3)*p2(3)
end

Thus the physical and mathematical quantities map to data and the algorithms map to a collection of functions.

A priori, we see no reason why moving to a language which supports the idea of objects, such as C++, should change the way we think of doing physics analysis. Thus the idea of having essentially mathematical objects such as vectors, points etc. and these being distinct from the more complex beasts which manipulate them, e.g. fitting algorithms etc. is still valid. This is the reason why the Gaudi application framework makes a clear distinction between “data” objects and “algorithm” objects.

Anything which has as its origin a concept such as hit, point, vector, trajectory, i.e. a clear “quantity-like” entity should be implemented by deriving a class from the `DataObject` base class.

On the other hand anything which is essentially a “procedure”, i.e. a set of rules for performing transformations on more data-like objects, or for creating new data-like objects should be designed as a class derived from the `Algorithm` base class.

Further more you should not have objects derived from `DataObject` performing long complex algorithmic procedures. The intention is that these objects are “small”.

Tracks which fit themselves are of course possible: you could have a constructor which took a list of hits as a parameter; but they are silly. Every track object would now have to contain all of the parameters used to perform the track fit, making it far from a simple object. Track-fitting is an algorithmic procedure; a track is probably best represented by a point and a vector, or perhaps a set of points and vectors. They are different.

### 2.4 Main components

The principle functionality of an algorithm is to take input data, manipulate it and produce new output data. Figure 2.1 shows how a concrete algorithm object interacts with the rest of the application framework to achieve this.

The figure shows the four main services that algorithm objects use:

- The event data store
- The detector data store
- The histogram service
- The message service

The particle property service is an example of additional services that are available to an algorithm. The job options service (see Chapter 13) is used by the `Algorithm` base class, but is not usually explicitly seen by a concrete algorithm.
Each of these services is provided by a component and the use of these components is via an interface. The interface used by algorithm objects is shown in the figure, e.g. for both the event data and detector data stores it is the `IDataProviderSvc` interface. In general a component implements more than one interface. For example the event data store implements another interface: `IDataManager` which is used by the application manager to clear the store before a new event is read in.

An algorithm’s access to data, whether the data is coming from or going to a persistent store or whether it is coming from or going to another algorithm is always via one of the data store components. The `IDataProviderSvc` interface allows algorithms to access data in the store and to add new data to the store. It is discussed further in Chapter 7 where we consider the data store components in more detail.

The histogram service is another type of data store intended for the storage of histograms and other “statistical” objects, i.e. data objects with a lifetime of longer than a single event. Access is via the `IHistogramSvc` which is an extension to the `IDataProviderSvc` interface, and is discussed in Chapter 11. The n-tuple service is similar, with access via the `INtupleSvc` extension to the `IDataProviderSvc` interface, as discussed in Chapter 12.
In general an algorithm will be configurable: It will require certain parameters, such as cut-offs, upper limits on the number of iterations, convergence criteria, etc., to be initialised before the algorithm may be executed. These parameters may be specified at run time via the job options mechanism. This is done by the job options service. Though it is not explicitly shown in the figure this component makes use of the IProperty interface which is implemented by the Algorithm base class.

During its execution an algorithm may wish to make reports on its progress or on errors that occur. All communication with the outside world should go through the message service component via the IMessageSvc interface. Use of this interface is discussed in Chapter 13.

As mentioned above, by virtue of its derivation from the Algorithm base class, any concrete algorithm class implements the IAlgorithm and IProperty interfaces, except for the three methods initialize(), execute(), and finalize() which must be explicitly implemented by the concrete algorithm. IAlgorithm is used by the application manager to control top-level algorithms. IProperty is usually used only by the job options service.

The figure also shows that a concrete algorithm may make use of additional objects internally to aid it in its function. These private objects do not need to inherit from any particular base class so long as they are only used internally. These objects are under the complete control of the algorithm object itself and so care is required to avoid memory leaks etc.

We have used the terms “interface” and “implements” quite freely above. Let us be more explicit about what we mean. We use the term interface to describe a pure virtual C++ class, i.e. a class with no data members, and no implementation of the methods that it declares. For example:

```cpp
class PureAbstractClass {
    virtual method1() = 0;
    virtual method2() = 0;
}
```

is a pure abstract class or interface. We say that a class implements such an interface if it is derived from it, for example:

```cpp
class ConcreteComponent: public PureAbstractClass {
    method1() { }
    method2() { }
}
```

A component which implements more than one interface does so via multiple inheritance, however, since the interfaces are pure abstract classes the usual problems associated with multiple inheritance do not occur. These interfaces are identified by a unique number which is available via a global constant of the form: IID_InterfaceType, such as for example IID_IDataProviderSvc. Using these it is possible to enquire what interfaces a particular component implements (as shown for example through the use queryInterface() in the finalize() method of the SimpleAnalysis example).
Within the framework every component, e.g. services and algorithms, has two qualities:

- A concrete component class, e.g. TrackFinderAlgorithm or MessageSvc.
- Its name, e.g. “KalmanFitAlgorithm” or “stdMessageService”.
Chapter 3
Release notes

3.1 Overview

These release notes identify changes since the previous release, focusing on new functionality, changes that are not backwards compatible, changes in external dependencies, and a brief summary of bugs that have been fixed, or are known to be outstanding.

3.2 New Functionality

3.3 Changes that are not backwards compatible

3.4 Changed dependencies on external software

1. ATLAS release 2.0.0 depends upon ATLAS GAUDI release 0.7.2.

3.5 Bugs Fixed

In general these should be referenced by the appropriate Remedy number, but this is not currently available.
3.6 Known Bugs

None.
Chapter 4

Establishing a development environment

4.1 Overview

This Chapter describes how to establish an environment to allow a developer to modify or create Athena-based applications. The details of this will depend upon the particular site, on whether the user is using a CERN computer, and whether AFS is available. Consult your local system administrator for details of how to login and to create a minimal environment. What is described here is the appropriate setup procedures for a CERN user on a CERN machine.

4.2 Establishing a login environment

4.2.1 Commands to establish a bourne-shell or varient login environment

The commands in Listing 4.1 establish a minimal login environment using the bourne shell or varients (sh, bash, zsh, etc.) and should be entered into the .profile or .bash_profile (?) file.

Listing 4.1  Bourne shell and varients commands to establish an ATLAS login environment

```
export ATLAS_ROOT=/afs/cern.ch/atlas
export CVSVROOT=:kserver:atlas-sw.cern.ch:/atlascvs
if [ "$PATH" != "" ]; then
  export PATH=${PATH}:${ATLAS_ROOT}/software/bin
else
  export PATH=${ATLAS_ROOT}/software/bin
fi
source `art setup -s sh`
```
4.2.2 Commands to establish a c-shell or varient login environment

The commands in Listing 4.2 establish a minimal login environment using the c-shell or varients (csh, tcsh, etc.) and should be entered into the .login file.

Listing 4.2 C shell and varients commands to establish an ATLAS login environment

```
setenv ATLAS_ROOT /afs/cern.ch/atlas
setenv CVSROOT :kserver:atlas-sw.cern.ch:/atlascvs
if ( $?PATH ) then
  setenv PATH ${PATH}:${ATLAS_ROOT}/software/bin
else
  setenv PATH ${ATLAS_ROOT}/software/bin
endif
source `srt setup -s csh`
```

4.3 Using SRT to checkout ATLAS software packages

ATLAS software is organized as a set of hierarchical packages, each package corresponding to a logical grouping of (typically) C++ classes. These packages are kept in a centralized code repository, managed by CVS [Ref]. Self-contained snapshots of the package hierarchy are created at frequent intervals, and executables and libraries are created from them. These snapshots are termed releases, and in many cases users can execute applications directly from a release of their choice. Each release is identified by a three-component identifier of the form ii.jj.kk (e.g. 1.3.2).
Chapter 5
Writing algorithms

5.1 Overview

As mentioned previously the framework makes use of the inheritance mechanism for specialising the Algorithm component. In other words, a concrete algorithm class must inherit from ("be derived from" in C++ parlance, "extend" in Java) the Algorithm base class.

In this chapter we first look at the base class itself. We then discuss what is involved in creating concrete algorithms: specifically how to declare properties, what to put into the methods of the IAlgorithm interface, the use of private objects and how to nest algorithms. Finally we look at how to set up sequences of algorithms and how to control processing through the use of branches and filters.

5.2 Algorithm base class

Since a concrete algorithm object is-an Algorithm object it may use all of the public methods of the Algorithm base class. The base class has no protected methods or data members, and no public data members, so in fact, these are the only methods that are available. Most of these methods are in fact provided solely to make the implementation of derived algorithms easier. The base class has two main responsibilities: the initialization of certain internal pointers and the management of the properties of derived algorithm classes.

A part of the Algorithm base class definition is shown in Listing 5.1. Include directives, forward declarations and private member variables have all been suppressed. It declares a constructor and destructor; the three key methods of the IAlgorithm interface; several accessors to services that a concrete algorithm will almost certainly require; a method to create a sub algorithm, the two methods of the IProperty interface; and a whole series of methods for declaring properties.
Listing 5.1 The definition of the Algorithm base class.

```cpp
1: class Algorithm : virtual public IAlgorithm,
2: virtual public IProperty {
3: public:
4: // Constructor and destructor
5: Algorithm( const std::string& name, ISvcLocator *svcloc );
6: virtual ~Algorithm();
7:
8: StatusCode sysInitialize();
9: StatusCode initialize();
10: StatusCode sysExecute();
11: StatusCode execute();
12: StatusCode sysFinalize();
13: StatusCode finalize();
14: const std::string& name() const;
15:
16: virtual bool isExecuted() const;
17: virtual StatusCode setExecuted( bool state );
18: virtual StatusCode resetExecuted();
19: virtual bool isEnabled() const;
20: virtual bool filterPassed() const;
21: virtual StatusCode setFilterPassed( bool state );
22:
23: template<class Svc>
24: StatusCode service( const std::string& svcName, Svc*& theSvc );
25: IMessageSvc* msgSvc();
26: void setOutputLevel( int level );
27: IDataProviderSvc* eventSvc();
28: IConversionSvc* eventCnvSvc();
29: IDataProviderSvc* detSvc();
30: IConversionSvc* detCnvSvc();
31: IHistogramSvc* histoSvc();
32: INtupleSvc* ntupleSvc();
33: IChronoStatSvc* chronoSvc();
34: IRndmGenSvc* randSvc();
35: ISvcLocator* serviceLocator();
36:
37: StatusCode createSubAlgorithm( const std::string& type,
38: const std::string& name, Algorithm*& pSubAlg );
39: std::vector<Algorithm*>* subAlgorithms() const;
40:
41: virtual StatusCode setProperty(const Property& p);
42: virtual StatusCode getProperty(Property* p) const;
43: const Property& getProperty( const std::string& name) const;
44: const std::vector<Property*>& getProperties() const;
45: StatusCode setProperties();
46:
47: StatusCode declareProperty(const std::string& name, int& reference);
48: StatusCode declareProperty(const std::string& name, float& reference);
49: StatusCode declareProperty(const std::string& name, double& reference);
50: StatusCode declareProperty(const std::string& name, bool& reference);
51: StatusCode declareProperty(const std::string& name,
52: std::string& reference);
53: // Vectors of properties not shown
54: private:
55: // Data members not shown
56: Algorithm(const Algorithm& a); // NO COPY ALLOWED
57: Algorithm& operator=(const Algorithm& rhs); // NO ASSIGNMENT ALLOWED;
```
Constructor and Destructor  The base class has a single constructor which takes two arguments: The first is the name that will identify the algorithm object being instantiated and the second is a pointer to one of the interfaces implemented by the application manager: ISvcLocator. This interface may be used to request special services that an algorithm may wish to use, but which are not available via the standard accessor methods (below).

The IAlgorithm interface  Principally this consists of the three pure virtual methods that must be implemented by a derived algorithm: initialize(), execute() and finalize(). These are where the algorithm does its useful work and discussed in more detail in section 5.3. Other methods of the interface are the accessor name() which returns the algorithm’s identifying name, and sysInitialize(), sysFinalize(), sysExecute() which are used internally by the framework. The latter three methods are not virtual and may not be overridden.

Service accessor methods  Lines 21 to 35 declare accessor methods which return pointers to key service interfaces. These methods are available for use only after the Algorithm base class has been initialized, i.e. they may not be used from within a concrete algorithm constructor, but may be used from within the initialize() method (see 5.3.3). The services and interface types to which they point are self explanatory (see also Chapter 2). Services may be located by name using the templated service() function in lines 21 and 22 or by using the serviceLocator() accessor method on line 35, as described in section 13.2 of Chapter 13. Line 26 declares a facility to modify the message output level from within the code (the message service is described in detail in section 13.4 of Chapter 13).

Creation of sub algorithms  The methods on lines 37 to 39 are intended to be used by a derived class to manage sub-algorithms, as discussed in section 5.4.

Declaration and setting of properties  As mentioned above, one of the responsibilities of the base class is the management of properties. The methods in lines 41 to 45 are used by the framework to set properties as defined in the job options file. The declareProperty methods (lines 47 to 53) are intended to be used by a derived class to declare its properties. This is discussed in more detail in section 5.3.2, and in Chapter 13.

Filtering  The methods in lines 14 to 19 are used by sequencers and filters to access the state of the algorithm, as discussed in section 5.5.

5.3 Derived algorithm classes

In order for an algorithm object to do anything useful it must be specialised, i.e. it must extend (inherit from, be derived from) the Algorithm base class. In general it will be necessary to implement the methods of the IAlgorithm interface, and declare the algorithm’s properties to the property management machinery of the Algorithm base class. Additionally there is one non-obvious technical matter to cover, namely algorithm factories.
5.3.1 Creation (and algorithm factories)

As mentioned before, a concrete algorithm class must specify a single constructor with the same parameter signature as the constructor of the base class.

In addition to this a concrete algorithm factory must be provided. This is a technical matter which permits the application manager to create new algorithm objects without having to include all of the concrete algorithm header files. From the point of view of an algorithm developer it implies adding two lines into the implementation file, of the form:

```cpp
static const AlgFactory<ConcreteAlgorithm> s_factory;
const IAlgFactory& ConcreteAlgorithmFactory = s_factory;
```

where “ConcreteAlgorithm” should be replaced by the name of the derived algorithm class (see for example lines 10 and 11 in Listing 5.2 below).

5.3.2 Declaring properties

In general a concrete algorithm class will have several data members which are used in the execution of the algorithm proper. These data members should of course be initialized in the constructor, but if this was the only mechanism available to set their value it would be necessary to recompile the code every time you wanted to run with different settings. In order to avoid this, the framework provides a mechanism for setting the values of member variables at run time.

The mechanism comes in two parts: the declaration of properties and the setting of their values. As an example consider the class TriggerDecision in Listing 5.2 which has a number of variables whose value we would like to set at run time.

The default values for the variables are set within the constructor (within an initialiser list) as per normal. To declare them as properties it suffices to call the declareProperty() method. This method is overloaded to take an std::string as the first parameter and a variety of different types for the second parameter. The first parameter is the name by which this member variable shall be referred to, and the second parameter is a reference to the member variable itself.

In the example we associate the name “PassAllMode” to the member variable m_passAllMode, and the name “MuonCandidateCut” to m_muonCandidateCut. The first is of type boolean and the second an integer. If the job options service (described in Chapter 13) finds an option in the job options file belonging to this algorithm and whose name matches one of the names associated with a member variable, then that member variable will be set to the value specified in the job options file.
In order to implement IAlgorithm you must implement its three pure virtual methods initialize(), execute() and finalize(). For a top level algorithm, i.e. one controlled directly by the application manager, the methods are invoked as is described in section 4.6. This dictates what it is useful to put into each of the methods.

Initialization In a standard job the application manager will initialize all top level algorithms exactly once before reading any event data. It does this by invoking the sysInitialize() method of each top-level algorithm in turn, in which the framework takes care of setting up internal references to standard services and to set the algorithm properties by calling the method setProperties(). This causes the job options service to make repeated calls to the setProperty() method of the algorithm, which actually assigns values to the member variables. Finally, sysInitialize() calls the initialize() method, which can be used to do such things as creating histograms, or creating sub-algorithms if required (see section 5.4). If an algorithm fails to initialize it should return StatusCode::FAILURE. This will cause the job to terminate.

Figure 5.1 shows an example trace of the initialization phase. Sub-algorithms are discussed in section 5.4.

Execution The guts of the algorithm class is in the execute() method. For top level algorithms this will be called once per event for each algorithm object in the order in which they were declared to the
application manager. For sub-algorithms (see section 5.4) the control flow may be as you like: you may call the `execute()` method once, many times or not at all.
Just because an algorithm derives from the Algorithm base class does not mean that it is limited to using or overriding only the methods defined by the base class. In general, your code will be much better structured (i.e. understandable, maintainable, etc.) if you do not, for example, implement the execute() method as a single block of 100 lines, but instead define your own utility methods and classes to better structure the code.

If an algorithm fails in some manner, e.g. a fit fails to converge, or its data is nonsense it should return from the execute() method with StatusCode::FAILURE. This will cause the application manager to stop processing events and end the job. This default behaviour can be modified by setting the <myAlgorithm>.ErrorMax job option to something greater than 1. In this case a message will be printed, but the job will continue as if there had been no error, and just increment an error count. The job will only stop if the error count reaches the ErrorMax limit set in the job option.

The framework (the Algorithm base class) calls the execute() method within a try/catch clause. This means that any exception not handled in the execution of an Algorithm will be caught at the level of sysExecute() implemented in the base class. The behaviour on these exceptions is identical to that described above for errors.

**Finalization** The finalize() method is called at the end of the job. It can be used to analyse statistics, fit histograms, or whatever you like. Similarly to initialization, the framework invokes a sysFinalize() method which in turn invokes the finalize() method of the algorithm and of any sub-algorithms.

Monitoring of the execution (e.g. cpu usage) of each Algorithm instance is performed by auditors under control of the Auditor service (described in Chapter 13). This monitoring can be turned on or off with the boolean properties AuditInitialize, AuditExecute, AuditFinalize.

The following is a list of things to do when implementing an algorithm.

- Derive your algorithm from the Algorithm base class.
- Provide the appropriate constructor and the three methods initialize(), execute() and finalize().
- Make sure you have implemented a factory by adding the magic two lines of code (see 5.3.1).

## 5.4 Nesting algorithms

The application manager is responsible for initializing, executing once per event, and finalizing the set of top level algorithms, i.e. the set of algorithms specified in the job options file. However such a simple linear structure is very limiting. You may wish to execute some algorithms only for specific types of event, or you may wish to “loop” over an algorithm’s execute method. Within the Athena application framework the way to have such control is via the nesting of algorithms or through algorithm sequences (described in section 5.5). A nested (or sub-) algorithm is one which is created by, and thus belongs to and is controlled by, another algorithm (its parent) as opposed to the application manager. In this section we discuss a number of points which are specific to sub-algorithms.
In the first place, the parent algorithm will need a member variable of type `Algorithm*` (see the code fragment below) in which to store a pointer to the sub-algorithm.

```
Algorithm* m_pSubAlgorithm; // Pointer to the sub algorithm
std::string type; // Must be a member variable of the parent class
std::string name; // Name to be given to subAlgorithm
StatusCode sc; // Status code returned by the call
sc = createSubAlgorithm(type, name, Algorithm* & m_pSubAlgorithm);
```

The sub-algorithm itself is created by invoking the `createSubAlgorithm()` method of the `Algorithm` base class. The parameters passed are the type of the algorithm, its name and a reference to the pointer which will be set to point to the newly created sub-algorithm. Note that the name passed into the `createSubAlgorithm()` method is the same name that should be used within the job options file for specifying algorithm properties.

The algorithm type (i.e. class name) string is used by the application manager to decide which factory should create the algorithm object.

The execution of the sub-algorithm is entirely the responsibility of the parent algorithm whereas the `initialize()` and `finalize()` methods are invoked automatically by the framework as shown in Figure 5.1. Similarly the properties of a sub-algorithm are also automatically set by the framework.

Note that the `createSubAlgorithm()` method returns a pointer to an `Algorithm` object, not an `IAlgorithm` interface. This means that you have access to the methods of both the `IAlgorithm` and `IProperty` interfaces, and consequently as well as being able to call `execute()` etc. you can also explicitly call the `setProperty(Property&)` method of the sub-algorithm, as is done in the following code fragment. For this reason with nested algorithms you are not restricted to calling `setProperty()` only at initialization. You may also change the properties of a sub-algorithm during the main event loop.

```
Algorithm *m_pSubAlgorithm;
sc = createSubAlgorithm(type, name, Algorithm* & m_pSubAlgorithm);
IntegerProperty p("Counter", 1024);
m_pSubAlgorithm->setProperty(p);
```

Note also that the vector of pointers to the sub-algorithms is available via the `subAlgorithms()` method.

## 5.5 Algorithm sequences, branches and filters

A physics application may wish to execute different algorithms depending on the physics signature of each event, which might be determined at run-time as a result of some reconstruction. This capability is supported in Athena through sequences, branches and filters. A `sequence` is a list of Algorithms. Each
Algorithm may make a *filter* decision, based on some characteristics of the event, which can either allow or bypass processing of the downstream algorithms in the sequence. The filter decision may also cause a *branch* whereby a different downstream sequence of Algorithms will be executed for events that pass the filter decision relative to those that fail it. Eventually the particular set of sequences, filters and branches might be used to determine which of multiple output destinations each event is written to (if at all). This capability is not yet implemented but is planned for a future release of Athena.

A Sequencer class is available in the GaudiAlg package which manages algorithm sequences using filtering and branching protocols which are implemented in the Algorithm class itself. The list of Algorithms in a Sequencer is specified through the Members property. Algorithms can call `setFilterPassed( true/false )` during their `execute()` function. Algorithms in the membership list downstream of one that sets this flag to `false` will not be executed, *unless* the StopOverride property of the Sequencer has been set, or the filtering algorithm itself is of type Sequencer and its BranchMembers property specifies a branch with downstream members. Please note that, if a sub-algorithm is of type Sequencer, the parent algorithm must call the `resetExecuted()` method of the sub-algorithm before calling the `execute()` method, otherwise the sequence will only be executed once in the lifetime of the job!

An algorithm *instance* is executed only once per event, even if it appears in multiple sequences. It may also be enabled or disabled, being enabled by default. This is controlled by the Enable property. Enabling and disabling of algorithm instances is a capability that is designed for use with the Python interactive scripting language support within Athena.

The filter passed or failed logic for a particular Algorithm instance in a sequence may be inverted by specifying the :invert optional flag in the Members list for the Sequencer in the job options file.

A Sequencer will report filter success if either of its main and branch member lists succeed. The two cases may be differentiated using the Sequencer `branchFilterPassed()` boolean function. If this is set true, then the branch filter was passed, otherwise both it and the main sequence indicated failure.

The following examples illustrate the use of sequences with filtering and branching.

### 5.5.1 Filtering example

Listing 5.3a and Listing 5.3b show extracts of the job options file and Python script of the AlgSequencer example: a Sequencer instance is created (line 2) with two members (line 5); each member is itself a Sequencer, implementing the sequences set up in lines 7 and 8, which consist of Prescaler, EventCounter and HelloWorld algorithms. The StopOverride property of the TopSequence is set to true, which causes both sequences to be executed, even if the first one indicates a filter failure.

The Prescaler and EventCounter classes are example algorithms distributed with the GaudiAlg package. The Prescaler class acts as a filter, passing the fraction of events specified by the PercentPass property (as a percentage). The EventCounter class just prints each event as it is encountered, and summarizes at the end of job how many events were seen. Thus at the end of job,
the Counter1 instance will report seeing 50% of the events, while the Counter2 instance will report seeing 10%.

Note the same instance of the HelloWorld class appears in both sequences. It will be executed in Sequence1 if Prescaler1 passes the event. It will be executed in Sequence2 if Prescaler2 passes the event only if Prescaler1 failed it.

**Listing 5.3a** Example job options using Sequencers demonstrating filtering

```plaintext
1: ApplicationMgr.DLLs += { "GaudiAlg" };  
2: ApplicationMgr.TopAlg = { "Sequencer/TopSequence" };  
3:  
4: // Setup the next level sequencers and their members  
5: TopSequence.Members = {"Sequencer/Sequence1", "Sequencer/Sequence2"};  
6: TopSequence.StopOverride = true;  
7: Sequence1.Members = {"Prescaler/Prescaler1", "HelloWorld",  
                         "EventCounter/Counter1"};  
8: Sequence2.Members = {"Prescaler/Prescaler2", "HelloWorld",  
                         "EventCounter/Counter2"};  
9:  
10: Prescaler1.PercentPass = 50.;  
11: Prescaler2.PercentPass = 10.;
```

**Listing 5.3b** Example Python script using Sequencers demonstrating filtering

```plaintext
1: theApp.DLLs = [ "GaudiAlg" ]  
2: theApp.TopAlg = [ "Sequencer/TopSequence" ]  
3: TopSequence = Algorithm( "TopSequence" )  
4:  
5: # Setup the next level sequencers and their members  
6: TopSequence.Members = [ "Sequencer/Sequence1", "Sequencer/Sequence2"]  
7: Sequence1 = Algorithm( "Sequence1" )  
8: Sequence2 = Algorithm( "Sequence2" )  
9: TopSequence.StopOverride = true  
10: Sequence1.Members = [ "Prescaler/Prescaler1", "HelloWorld",  
                         "EventCounter/Counter1" ]  
11: Prescaler1 = Algorithm( "Prescaler1" )  
12: Counter1 = Algorithm( "Counter1" )  
13: Sequence2.Members = [ "Prescaler/Prescaler2", "HelloWorld",  
                         "EventCounter/Counter2" ]  
14: Prescaler2 = Algorithm( "Prescaler2" )  
15: Counter2 = Algorithm( "Counter2" )  
16:  
17: Prescaler1.PercentPass = 50.  
```

**Sequence branching**

Listing 5.4a and Listing 5.4b illustrate the use of explicit branching. The BranchMembers property of the Sequencer specifies some algorithms to be executed if the algorithm that is the first member of the branch (which is common to both the main and branch membership lists) indicates a filter failure. In
this example the EventCounter instance Counter1 will report seeing 80% of the events, whereas Counter2 will report seeing 20%.

Listing 5.4a Example job options using Sequencers demonstrating branching

```
1: ApplicationMgr.DLLs += { "GaudiAlg" };  
2: ApplicationMgr.TopAlg = { "Sequencer" };  
3:  
4: // Setup the next level sequencers and their members  
5: Sequencer.Members = {"HelloWorld", "Prescaler",  
   "EventCounter/Counter1"};  
6: Sequencer.BranchMembers = {"Prescaler", "EventCounter/Counter2"};  
7:  
8: Prescaler.PercentPass = 80.;
```

Listing 5.4b Example Oython script using Sequencers demonstrating branching

```
1: theApp.DLLs = [ "GaudiAlg" ]  
2: theApp.TopAlg = [ "Sequencer" ]  
3: Sequencer = Algorithm( "Sequencer" )  
4:  
5: # Setup the next level sequencers and their members  
6: Sequencer.Members = [ "HelloWorld", "Prescaler",  
   "EventCounter/Counter1" ]  
7: HelloWorld = Algorithm( "HelloWorld" )  
8: Prescaler = Algorithm( "Prescaler" )  
9: Counter1 = Algorithm( "Counter1" )  
10: Sequencer.BranchMembers = [ "Prescaler", "EventCounter/Counter2"]  
11: Counter2 = Algorithm( "Counter2" )  
12:  
```

Listing 5.5a and Listing 5.5b illustrate the use of inverted logic. They achieve the same goal as the example in Listing 5.4a and Listing 5.4b through use of two sequences with the same instance of a Prescaler filter, but where the second sequence contains inverted logic for the single instance.

Listing 5.5a Example job options using Sequencers demonstrating inverted logic

```
1: ApplicationMgr.DLLs += { "GaudiAlg" };  
2: ApplicationMgr.TopAlg = { "Sequencer/Seq1", "Sequencer/Seq2" };  
3:  
4: // Setup the next level sequencers and their members  
5: Seq1.Members = {"HelloWorld", "Prescaler", "EventCounter/Counter1"};  
   "EventCounter/Counter2"};  
7:  
8: Prescaler.PercentPass = 80.;
```
Listing 5.5b  Example Python script using Sequencers demonstrating inverted logic

1: theApp.DLLs = [ "GaudiAlg" ]
2: theApp.TopAlg = [ "Sequencer/Seq1", "Sequencer/Seq2" ]
3: Seq1 = Algorithm( "Seq1" )
4: Seq2 = Algorithm( "Seq2" )
5:
6: # Setup the next level sequencers and their members
7: Seq1.Members = ["HelloWorld", "Prescaler", "EventCounter/Counter1"]
9: HelloWorld = Algorithm( "HelloWorld" )
10: Prescaler = Algorithm( "Prescaler" )
11: Counter1 = Algorithm( "Counter1" )
12: Counter2 = Algorithm( "Counter2" )
13:
Chapter 6
Scripting

6.1 Overview

Athena scripting support is available in prototype form. The functionality is likely to change rapidly, so users should check with the latest release notes for changes or new functionality that might not be documented here.

6.2 Python scripting service

In keeping with the design philosophy of Athena and the underlying GAUDI architecture, scripting is defined by an abstract scripting service interface, with the possibility of there being several different implementations. A prototype implementation is available based upon the Python[4] scripting language. The Python scripting language will not be described in detail here, but only a brief overview will be presented.

6.3 Python overview

This section is in preparation.
6.4 How to enable Python scripting

Two different mechanisms are available for enabling Python scripting.

1. Replace the job options text file by a Python script that is specified on the command line.

2. Use a job options text file which hands control over to the Python shell once the initial configuration has been established.

6.4.1 Using a Python script for configuration and control

The necessity for using a job options text file for configuration can be avoided by specifying a Python script as a command line argument as shown in Listing 6.1.

Listing 6.1 Using a Python script for job configuration

```
athena MyPythonScript.py [1]
```

Notes:

1. The file extension .py is used to identify the job options file as a Python script. All other extensions are assumed to be job options text files.

This approach may be used in two modes. The first uses such a script to establish the configuration, but results in the job being left at the Python shell prompt. This supports interactive sessions. The second specifies a complete configuration and control sequence and thus supports a batch style of processing. The particular mode is controlled by the presence or absence of Athena-specific Python commands described in Section 6.8.

6.4.2 Using a job options text file for configuration with a Python interactive shell

Python scripting is enabled when using a job options text file for job configuration by adding the lines shown in Listing 6.2 to the job options file.

Listing 6.2 Job Options text file entries to enable Python scripting

```
ApplicationMgr.DLLs += { "SIPython" }; [1]
```

Notes:
1. This entry specifies the component library that implements Python scripting. Care should be taken to use the “+=" syntax in order not to overwrite other component libraries that might be specified elsewhere.

2. This entry specifies the Python scripting implementation of the abstract Scripting service. As with the previous line, care should be taken to use the “+=" syntax in order not to override other services that might be specified elsewhere.

Once the initial configuration has been established by the job options text file, control will be handed over to the Python shell.

It is possible to specify a specific job options configuration file at the command line as shown in Listing 6.3.

**Listing 6.3** Specifying a job options file for application execution

```bash
 athena [job options file] [1]
```

Notes:

1. The job options text file command line argument is optional. The file `jobOptions.txt` is assumed by default.

2. The file extension `.py` is used to identify the job options file as a Python script. All other extensions are assumed to be job options text files. The use of a Python script for configuration and control is described in Section 6.4.1.

### 6.5 Prototype functionality

The functionality of the prototype is limited to the following capabilities. This list will be added to as new capabilities are added:

1. The ability to read and store basic Properties for framework components (Algorithms, Services, Auditors) and the main ApplicationMgr that controls the application. Basic properties are basic type data members (int, float, etc.) or SimpleProperties of the components that are declared as Properties via the declareProperty() function.

2. The ability to retrieve and store individual elements of array properties.

3. The ability to specify a new set of top level Algorithms.

4. The ability to add new services and component libraries and access their capabilities

5. The ability to specify a new set of members or branch members for Sequencer algorithms.

6. The ability to specify a new set of output streams.

7. The ability to specify a new set of "AcceptAlgs", "RequireAlgs", or "VetoAlgs" properties for output streams.
6.6 Property manipulation

An illustration of the use of the scripting language to display and set component properties is shown in Listing 6.4:

Listing 6.4 Property manipulation from the Python interactive shell

```
>>> Algorithm.names
('TopSequence', 'Sequencer1', 'Sequence2')

>>> Service.names
('MessageSvc', 'JobOptionsSvc', 'EventDataSvc', 'EventPersistencySvc',
'DetectorDataSvc', 'DetectorPersistencySvc', 'HistogramDataSvc',
'NTupleSvc', 'IncidentSvc', 'ToolSvc', 'HistogramPersistencySvc',
'ParticlePropertySvc', 'ChronoStatSvc', 'RndmGenSvc', 'AuditorSvc',
'ScriptingSvc', 'RndmGenSvc.Engine')

>>> TopSequence.properties
{'ErrorCount': 0, 'OutputLevel': 0, 'BranchMembers': [],
'AuditExecute': 1, 'AuditInitialize': 0, 'Members':
['Sequencer/Sequence1', 'Sequencer/Sequence2'], 'StopOverride': 1,
'Enable': 1, 'AuditFinalize': 0, 'ErrorMax': 1}

>>> TopSequence.OutputLevel
'OutputLevel': 0

>>> TopSequence.OutputLevel=1

>>> TopSequence.Members=['Sequencer/NewSeq1', 'Sequencer/NewSeq1']

>>> TopSequence.properties
{'ErrorCount': 0, 'OutputLevel': 1, 'BranchMembers': [],
'AuditExecute': 1, 'AuditInitialize': 0, 'Members':
['Sequencer/NewSeq1', 'Sequencer/NewSeq1'], 'StopOverride': 1,
'Enable': 1, 'AuditFinalize': 0, 'ErrorMax': 1}

>>> theApp.properties
{'JobOptionsType': 'FILE', 'EvtMax': 100, 'DetDbLocation': 'empty',
'Dlls': ['HbookCnv', 'SI_Python'], 'DetDbRootName': 'empty',
'JobOptionsPath': 'jobOptions.txt', 'OutStream': [],
'HistogramPersistency': 'HBOOK', 'EvtSel': 'NONE', 'ExtSvc':
['PythonScriptingSvc/ScriptingSvc'], 'DetStorageType': 0, 'TopAlg':
['Sequencer/TopSequence']}

>>> 
```

Notes:

1. The ">>>" is the Python shell prompt.
2. The set of existing Algorithms is given by the Algorithm.names command.
3. The set of existing Services is given by the Service.names command.
4. The values of the properties for an Algorithm or Service may be displayed using the `<name>.properties` command, where `<name>` is the name of the desired Algorithm or Service.

5. The value of a single Property may be displayed (or used in a Python expression) using the `<name>.<property>` syntax, where `<name>` is the name of the desired Algorithm or Service, and `<property>` is the name of the desired Property.

6. Single valued properties (e.g. `IntegerProperty`) may be set using an assignment statement. Boolean properties use integer values of 0 (or FALSE) and 1 (or TRUE). Strings are enclosed in single-quotes or double-quotes.

7. Multi-valued properties (e.g. `StringArrayProperty`) are set using " [...] " as the array delimiters.

8. The `theApp` object corresponds to the ApplicationMgr and may be used to access its properties.

### 6.7 Synchronization between Python and Athena

It is possible to create new Algorithms or Services as a result of a scripting command. Examples of this are shown in Listing 6.5:

**Listing 6.5** Examples of Python commands that create new Algorithms or Services

```python
>>> theApp.ExtSvc = [ "ANewService" ]
>>> theApp.TopAlg = [ "TopSequencer/Sequencer" ]
```

If the specified Algorithm or Service already exists then its properties can immediately be accessed. However, in the prototype the properties of newly created objects cannot be accessed until an equivalent Python object is also created. This restriction will be removed in a future release.

This synchronization mechanism for creation of Python Algorithms and Services is illustrated in Listing 6.6:

**Listing 6.6** Examples of Python commands that create new Algorithms or Services

```python
>>> theApp.ExtSvc = [ "ANewService" ]
>>> ANewService = Service( "ANewService" ) [1]
>>> theApp.TopAlg = [ "TopSequencer/Sequencer" ]
>>> TopSequencer = Algorithm( "TopSequencer" ) [2]
>>> TopSequencer.properties
```

Notes:
1. This creates a new Python object of type Sequencer, having the same name as the newly created Athena Sequencer.

2. This creates a new Python object of type Algorithm, having the same name as the newly created Athena Algorithm.

The Python commands that might require a subsequent synchronization are shown in Listing 6.7:

**Listing 6.7  Examples of Python commands that might create new Algorithms or Services**

```python
theApp.ExtSvc = [...]  
theApp.TopAlg = [...]  
Sequencer.Members = [...]  
Sequencer.BranchMembers = [...]  
OutStream.AcceptAlgs = [...]  
OutStream.RequireAlgs = [...]  
OutStream.VetoAlgs = [...]  
```

### 6.8 Controlling job execution

This is very limited in the prototype, and will be replaced in a future release by the ability to call functions on the Python objects corresponding to the ApplicationMgr (theApp), Algorithms, and Services.

In the prototype, control is returned from the Python shell to the Athena environment by the command in Listing 6.8:

**Listing 6.8  Python command to resume Athena execution**

```python
>>> theApp.Go  
```

Notes:

1. This is a temporary command that will be replaced in a future release by a more flexible ability to access more functions of the ApplicationMgr.

This will cause the currently configured event loop to be executed, after which control will be returned to the Python shell.

Typing Ctrl-D (holding down the Ctrl key while striking the D key) at the Python shell prompt will cause an orderly termination of the job. Alternatively, the command shown in Listing 6.9 will also cause an orderly application termination.

**Listing 6.9  Python command to terminate Athena execution**

```python
>>> theApp.Exit  
```
This command, used in conjunction with the theApp.Go command, can be used to execute a Python script in batch rather than interactive mode. This provides equivalent functionality to a job options text file, but using the Python syntax. An example of such a batch Python script is shown in Listing 6.10:

**Listing 6.10** Python batch script

```python
>>> theApp.TopAlg = [ "HelloWorld" ]
[other configuration commands]
>>> theApp.Go
>>> theApp.Exit
```
Chapter 7
Accessing data

7.1 Overview

The data stores are a key component in the application framework. All data which comes from persistent storage, or which is transferred between algorithms, or which is to be made persistent must reside within a data store. In this chapter we use a trivial event data model to look at how to access data within the stores, and also at the \texttt{DataObject} base class and some container classes related to it.

We also cover how to define your own data types and the steps necessary to save newly created objects to disk files. The writing of the converters necessary for the latter is covered in Chapter 15.

7.2 Using the data stores

There are four data stores currently implemented within the

\textbf{Listing 7.1} Example job options using Sequencers demonstrating inverted logic

\begin{verbatim}
1: ApplicationMgr.DLLs += { "GaudiAlg" };  
2: ApplicationMgr.TopAlg = { "Sequencer/Seq1", "Sequencer/Seq2" };  
3:  
4: // Setup the next level sequencers and their members  
5: Seq1.Members = {"HelloWorld", "Prescaler", "EventCounter/Counter1"};  
     "EventCounter/Counter2"};  
7:  
8: Prescaler.PercentPass = 80.;
\end{verbatim}
framework: the event data store, the detector data store, the histogram store and the n-tuple store. Event
data is the subject of this chapter. The other data stores are described in chapters 10, 11 and 12 respectively. The stores themselves are no more than logical constructs with the actual access to the data being via the corresponding services. Both the event data service and the detector data service implement the same `IDataProviderSvc` interface, which can be used by algorithms to retrieve and store data. The histogram and n-tuple services implement extended versions of this interface (`IHistogramSvc`, `INTupleSvc`) which offer methods for creating and manipulating histograms and n-tuples, in addition to the data access methods provided by the other two stores.

Only objects of a type derived from the `DataObject` base class may be placed directly within a data store. Within the store the objects are arranged in a tree structure, just like a Unix file system. As an example consider Figure 7.1 which shows the trivial transient event data model of the `RootIO` example. An object is identified by its position in the tree expressed as a string such as: “/Event”, or “/Event/MyTracks”. In principle the structure of the tree, i.e. the set of all valid paths, may be deduced at run time by making repeated queries to the event data service, but this is unlikely to be useful in general since the structure will be largely fixed.

![Figure 7.1](unnamed) The structure the event data model of the RootIO example.

All interactions with the data stores should be via the `IDataProviderSvc` interface. The key methods for this interface are shown in Listing 7.2.

**Listing 7.2** Some of the key methods of the `IDataProviderSvc` interface.

```cpp
StatusCode findObject(const std::string& path, DataObject*& pObject);
StatusCode findObject(DataObject* node, const std::string& path,
                      DataObject*& pObject);
StatusCode retrieveObject(const std::string& path, DataObject*& pObject);
StatusCode retrieveObject(DataObject* node, const std::string& path,
                          DataObject*& pObject);

StatusCode registerObject(const std::string path, DataObject*& pObject);
StatusCode registerObject(DataObject *node, DataObject*& pObject);
```
The first four methods are for retrieving a pointer to an object that is already in the store. How the object got into the store, whether it has been read in from a persistent store or added to the store by an algorithm, is irrelevant.

The **find** and **retrieve** methods come in two versions: one version uses a full path name as an object identifier, the other takes a pointer to a previously retrieved object and the name of the object to look for below that node in the tree.

Additionally the **find** and **retrieve** methods differ in one important respect: the **find** method will look in the store to see if the object is present (i.e. in memory) and if it is not will return a null pointer. The **retrieve** method, however, will attempt to load the object from a persistent store (database or file) if it is not found in memory. Only if it is not found in the persistent data store will the method return a null pointer (and a bad status code of course).

### 7.3 Using data objects

Whatever the concrete type of the object you have retrieved from the store the pointer which you have is a pointer to a **DataObject**, so before you can do anything useful with that object you must cast it to the correct type, for example:

```c++
1: typedef ObjectVector<MyTrack> MyTrackVector;
2: DataObject *pObject;
3: 
4: StatusCode sc = eventSvc()->retrieveObject("/Event/MyTracks", pObject);
5: if( sc.isFailure() )
6:   return sc;
7: 
8: MyTrackVector *tv = 0;
9: try {
10:   tv = dynamic_cast<MyTrackVector *> (pObject);
11: } catch(...) {
12:     // Print out an error message and return
13:   } 
14: // tv may now be manipulated.
```

The **typedef** on line 1 is just to save typing: in what follows we will use the two syntaxes interchangeably. After the **dynamic_cast** on line 10 all of the methods of the **MyTrackVector** class become available. If the object which is returned from the store does not match the type to which you try to cast it, an exception will be thrown. If you do not catch this exception it will be caught by the algorithm base class, and the program will stop, probably with an obscure message. A more elegant way to retrieve the data involves the use of **Smart Pointers** - this is discussed in section 7.8

As mentioned earlier a certain amount of run-time investigation may be done into what data is available in the store. For example, suppose that we have various sets of testbeam data and each data set was taken with a different number of detectors. If the raw data is saved on a per-detector basis the number of
sets will vary. The code fragment in Listing 7.3 illustrates how an algorithm may loop over the data sets without knowing a priori how many there are.

**Listing 7.3** Code fragment for accessing an object from the store

```
1: std::string objectPath = "Event/RawData";
2: DataObject* pObject;
3: StatusCode sc;
4: 
5: sc = eventSvc()->retrieveObject(objectPath, pObject);
6: 
7: IdataDirectory *dir = pObject->directory();
8: IdataDirectory::DirIterator it;
9: for(it = dir->begin(); it != dir->end(); it++) {
10:     DataObject *pDo;
11:     sc = retrieveObject(pObject, (*it)->localPath(), pDo);
12:     // Do something with pDo
13: }
```

The last two methods shown in Listing 7.2 are for registering objects into the store. Suppose that an algorithm creates objects of type UDO from, say, objects of type MyTrack and wishes to place these into the store for use by other algorithms. Code to do this might look something like:

**Listing 7.4** Registering of objects into the event data store

```
1: UDO* pO; // Pointer to an object of type UDO (derived from DataObject)
2: StatusCode sc;
3: 
4: pO = new UDO;
5: sc = eventSvc()->registerObject("/Event/tmp","OK", pO);
6: 
7: // THE NEXT LINE IS AN ERROR, THE OBJECT NOW BELONGS TO THE STORE
8: delete pO;
9: 
10: UDO autopO;
11: // ERROR: AUTOMATIC OBJECTS MAY NOT BE REGISTERED
12: sc = eventSvc()->registerObject("/Event/tmp", "notOK", autopO);
```

Once an object is registered into the store, the algorithm which created it relinquishes ownership. In other words the object should not be deleted. This is also true for objects which are contained within other objects, such as those derived from or instantiated from the ObjectVector class (see the following section). Furthermore objects which are to be registered into the store must be created on the heap, i.e. they must be created with the `new` operator.
7.4 Object containers

As mentioned before, all objects which can be placed directly within one of the stores must be derived from the `DataObject` class. There is, however, another (indirect) way to store objects within a store. This is by putting a set of objects (themselves not derived from `DataObject` and thus not directly storable) into an object which is derived from `DataObject` and which may thus be registered into a store.

An object container base class is implemented within the framework and a number of templated object container classes may be implemented in the future. For the moment, two “concrete” container classes are implemented: `ObjectVector<T>` and `ObjectList<T>`. These classes are based upon the STL classes and provide mostly the same interface. Unlike the STL containers which are essentially designed to hold objects, the container classes within the framework contain only pointers to objects, thus avoiding a lot of memory to memory copying.

A further difference with the STL containers is that the type `T` cannot be anything you like. It must be a type derived from the `ContainedObject` base class, see Figure 7.1. In this way all “contained” objects have a pointer back to their containing object. This is required, in particular, by the converters for dealing with links between objects. A ramification of this is that container objects may not contain other container objects (without the use of multiple inheritance).

![Diagram](image.png)

**Figure 7.1** The relationship between the `DataObject`, `ObjectVector` and `ContainedObject` classes.

As mentioned above, objects which are contained within one of these container objects may not be located, or registered, individually within the store. Only the container object may be located via a call to `findObject()` or `retrieveObject()`. Thus with regard to interaction with the data stores a container object and the objects that it contains behave as a single object.
The intention is that “small” objects such as clusters, hits, tracks, etc. are derived from the ContainedObject base class and that in general algorithms will take object containers as their input data and produce new object containers of a different type as their output.

The reason behind this is essentially one of optimization. If all objects were treated on an equal footing, then there would be many more accesses to the persistent store to retrieve very small objects. By grouping objects together like this we are able to have fewer accesses, with each access retrieving bigger objects.

### 7.5 Using object containers

The code fragment below shows the creation of an object container. This container can contain pointers to objects of type MyTrack and only to objects of this type (including derived types). An object of the required type is created on the heap (i.e. via a call to new) and is added to the container with the standard STL call.

```cpp
ObjectVector<MyTrack> trackContainer;
MyTrack* h1 = new MyTrack;
trackContainer.push_back(h1);
```

After the call to push_back() the MyTrack object “belongs” to the container. If the container is registered into the store, the hits that it contains will go with it. Note in particular that if you delete the container you will also delete its contents, i.e. all of the objects pointed to by the pointers in the container.

Removing an object from a container may be done in two semantically different ways. The difference being whether on removal from a container the object is also deleted or not. Removal with deletion may be achieved in several ways (following previous code fragment):

```cpp
trackContainer.pop_back();
trackContainer.erase( end() );
delete h1;
```

The method pop_back() removes the last element in the container, whereas erase() maybe used to remove any other element via an iterator. In the code fragment above it is used to remove the last element also.

Deleting a contained object, the third option above, will automatically trigger its removal from the container. This is done by the destructor of the ContainedObject base class.

If you wish to remove an object from the container without destroying it (the second possible semantic) use the release() method:
Since the fate of a contained object is so closely tied to that of its container life would become more complex if objects could belong to more than one container. Suppose that an object belonged to two containers, one of which was deleted. Should the object be deleted and removed from the second container, or not deleted? To avoid such issues an object is allowed to belong to a single container only.

If you wish to move an object from one container to another, you must first remove it from one and then add to the other. However, the first operation is done implicitly for you when you try to add an object to a second container:

```cpp
container1.push_back(h1); // Add to first container
container2.push_back(h1); // Move to second container
    // Internally invokes release().
```

Since the object h1 has a link back to its container, the `push_back()` method is able to first follow this link and invoke the `release()` method to remove the object from the first container, before adding it into the second.

In general your first exposure to object containers is likely to be when retrieving data from the event data store. The sample code in Listing 7.5 shows how, once you have retrieved an object container from the store you may iterate over its contents, just as with an STL vector.

**Listing 7.5 Use of the ObjectVector templated class.**

```cpp
1: typedef ObjectVector<MyTrack> MyTrackVector;
2: MyTrackVector *tracks;
3: MyTrackVector::iterator it;
4: for( it = tracks->begin(); it != tracks->end(); it++ ) {  
5:     double energy = (*it)->fourMomentum().e();
6:     m_hEnergyDist->fill( energy, 1. );
7: }
```

The variable `tracks` is set to point to an object in the event data store of type `ObjectVector<MyTrack>` with a dynamic cast (not shown above). An iterator (i.e. a pointer-like object for looping over the contents of the container) is defined on line 3 and this is used within the loop to point consecutively to each of the contained objects. In this case the objects contained within the `ObjectVector` are of type “pointer to MyTrack”. The iterator returns each object in turn and in the example, the energy of the object is used to fill a histogram.
7.6 Data access checklist

A little reminder:

- Do not delete objects that you have registered.
- Do not delete objects that are contained within an object that you have registered.
- Do not register local objects, i.e. objects NOT created with the new operator.
- Do not delete objects which you got from the store via findObject() or retrieveObject().
- Do delete objects which you create on the heap, i.e. by a call to new, and which you do not register into a store.

7.7 Defining new data types

Most of the data types which will be used within Athena will be used by everybody and thus packaged and documented centrally. However, for your own private development work you may wish to create objects of your own types which of course you can always do with C++ (or Java). However, if you wish to place these objects within a store, either so as to pass them between algorithms or to have them later saved into a database or file, then you must derive your type from either the DataObject or ContainedObject base class.

Consider the example below:

```c++
const static CLID CLID_UDO = 135; // Collaboration wide Unique number

class UDO : public DataObject {
public:
    UDO() : DataObject(), m_n(0) {} 
    
    static const CLID& classID() { return CLID_UDO; }
    virtual const CLID& cID() const { return classID(); }

    int n(){ return m_n; }
    void setN(int n){ m_n = n; }

private:
    int m_n;
};
```

This defines a class UDO which since it derives from DataObject may be registered into, say, the event data store. (The class itself is not very useful as its sole attribute is a single integer and it has no behaviour).
The thing to note here is that if the appropriate converter is supplied, as discussed in Chapter 15, then this class may also be saved into a persistent store (e.g. a ROOT file or an Objectivity database) and read back at a later date. In order for the persistency to work the following are required: the unique class identifier number (CLID_UDO in the example), and the clID() and classID() methods which return this identifier.

The procedure for allocating unique class identifiers is, for the time being, experiment specific.

Types which are derived from ContainedObject are implemented in the same way, and must have a CLID in the range of an unsigned short. Contained objects may only reside in the store when they belong to a container, e.g. an ObjectVector<T> which is registered into the store. The class identifier of a concrete object container class is calculated (at run time) from the type of the objects which it contains, by setting bit 16. The static classID() method is required because the container may be empty.

### 7.8 The SmartDataPtr/SmartDataLocator utilities

The usage of the data services is simple, but extensive status checking and other things tend to make the code difficult to read. It would be more convenient to access data items in the store in a similar way to accessing objects with a C++ pointer. This is achieved with smart pointers, which hide the internals of the data services.

#### 7.8.1 Using SmartDataPtr/SmartDataLocator objects

The SmartDataPtr and a SmartDataLocator are smart pointers that differ by the access to the data store. SmartDataPtr first checks whether the requested object is present in the transient store and loads it if necessary (similar to the retrieveObject method of IDataProviderSvc). SmartDataLocator only checks for the presence of the object but does not attempt to load it (similar to findObject).

Both SmartDataPtr and SmartDataLocator objects use the data service to get hold of the requested object and deliver it to the user. Since both objects have similar behaviour and the same user interface, in the following only the SmartDataPtr is discussed.
An example use of the `SmartDataPtr` class is shown below.

**Listing 7.6** Use of a `SmartDataPtr` object.

```cpp
1: StatusCode myAlgo::execute() {
2:     MsgStream log(msgSvc(), name());
3:     SmartDataPtr<Event> evt(eventSvc(),"/Event");
4: if ( evt ) {
5:         // Print the event number
6:         log << MSG::INFO << " Run:" << evt->run()
7:             << " Event:" << evt->event() << endreq;
8:     }
9: else {
10:        log << MSG::ERROR << "Error accessing event" << endreq;
11:        return StatusCode::FAILURE;
12:    }
13: }
```

The `SmartDataPtr` class can be thought of as a normal C++ pointer having a constructor. It is used in the same way as a normal C++ pointer.

The `SmartDataPtr` and `SmartDataLocator` offer a number of possible constructors and operators to cover a wide range of needs when accessing data stores. Check the online reference documentation [2] for up-to-date information concerning the interface of these utilities.

### 7.9 Smart references and Smart reference vectors

Smart references and Smart reference vectors are similar to smart pointers, they are used within data objects to reference other objects in the transient data store. They provide safe data access and automate the loading on demand of referenced data, and should be used instead of C++ pointers. For example, suppose that `MCParticles` are already loaded but `MCVertices` are not, and that an algorithm dereferences a variable pointing to the origin vertex: if a smart reference is used, the `MCVertices` would be loaded automatically and only after that would the variable be dereferenced. If a C++ plain pointer were used instead, the program would crash. Smart references provide an automatic conversion to a pointer to the object and load the object from the persistent medium during the conversion process.
Smart references and Smart reference vectors are declared inside a class as:

```cpp
#include "GaudiKernel/SmartRef.h"
#include "GaudiKernel/SmartRefVector.h"

class MCParticle {
    private:
        /// Smart reference to origin vertex
        SmartRef<MCVertex> m_originMCVertex;
        /// Vector of smart references to decay vertices
        SmartRefVector<MCVertex> m_decayMCVertices;
    public:
        /// Access the origin Vertex
        /// Note: When the smart reference is converted to MCVertex* the object
        /// will be loaded from the persistent medium.
        MCVertex* originMCVertex() { return m_originMCVertex; }
};
```

The syntax of usage of smart references is identical to plain C++ pointers. The Algorithm only sees a pointer to the MCVertex object:

```cpp
#include "GaudiKernel/SmartDataPtr.h"

// Use a SmartDataPtr to get the MC particles from the event store
SmartDataPtr<MCParticleVector> particles(eventSvc(),"/Event/MC/MCParticles");
MCParticleVector::const_iterator iter;

// Loop over the particles to access the MCVertex via the SmartRef
for( iter = particles->begin(); iter != particles->end(); iter++ ) {
    MCVertex* originVtx = (*iter)->originMCVertex();
    if( 0 != originVtx ) {
        std::cout << "Origin vertex = " << *(*iter) << std::endl;
    }
}
```

SmartRef offers a number of possible constructors and operators, see the online reference documentation [2].

### 7.10 Saving data to a persistent store

Suppose that you have defined your own data type as discussed in section 7.7. Suppose furthermore that you have an algorithm which creates instances of your object type which you then register into the transient event store. How can you save these objects for use at a later date?

You must do the following:

- Write the appropriate converter (see Chapter 15)
- Put some instructions (i.e. options) into the job option file (see Listing 7.7)

- Register your object in the store as usual, typically in the `execute()` method of your algorithm.

```c
// myAlg implementation file

StatusCode myAlg::execute() {
    // Create a UDO object and register it into the event data store
    UDO* p = new UDO();
    eventSvc->registerObject("/Event/myStuff/myUDO", p);
}
```

In order to actually trigger the conversion and saving of the objects at the end of the current event processing it is necessary to inform the application manager. This requires some options to be specified in the job options file:

**Listing 7.7** Job options for output to persistent storage

```
ApplicationMgr.OutStream = { "DstWriter" };
DstWriter.ItemList = { "/Event#1", "/Event/MyTracks#1"};
DstWriterEvtDataSvc = "EventDataSvc";
DstWriter.Output = "DATAFILE='result.root' TYP='ROOT'";
ApplicationMgr.DLLs += { "DbConverters", "RootDb"};
ApplicationMgr.ExtSvc += { "DbEventCnvSvc/RootEvtCnvSvc" };
EventPersistencySvc.CnvServices += { "RootEvtCnvSvc" };
RootEvtCnvSvc.DbType = "ROOT";
```

The first option tells the application manager that you wish to create an output stream called “DstWriter”. You may create as many output streams as you like and give them whatever name you prefer.

For each output stream object which you create you must set several properties. The `ItemList` option specifies the list of paths to the objects which you wish to write to this output stream. The number after the “#” symbol denotes the number of directory levels below the specified path which should be traversed. The (optional) `EvtDataSvc` option specifies in which transient data service the output stream should search for the objects in the `ItemList`, the default is the standard transient event data service `EventDataSvc`. The `Output` option specifies the name of the output data file and the type of persistency technology, ROOT in this example. The last three options are needed to tell the Application manager to instantiate the `RootEvtCnvSvc` and to associate the ROOT persistency type to this service.

An example of saving data to a ROOT persistent data store is available in the `RootIO` example distributed with the framework.
Chapter 8

StoreGate - the event data access model

8.1 Overview

The event data access model (EDM) is a crucial element of the overall infrastructure that defines the management and use of DataObjects in its transient state. Based on the experience in using the Gaudi infrastructure and subsequent discussions within the EDM working group, we are developing the StoreGate to satisfy the ATLAS requirements outlined in the next section. We have implemented and tested some of the proposed design features stressing on the interface to the client's access to Data Objects while currently maintaining the underlying Gaudi infrastructure. This Chapter describes some of the proposed elements of the StoreGate design and provides detailed documentation on the use of the prototype.

8.2 The StoreGate design

The ATLAS software architecture belongs to the blackboard family: data objects produced by knowledge objects (e.g., reconstruction modules) are posted to a common in-memory database from where other modules can access them and produce new data objects. This model greatly reduces the coupling between knowledge objects containing the algorithmic code for analysis and reconstruction. A knowledge object does not need to know which specific module can produce the information it needs, nor which protocol it must use to obtain it. Algorithmic code is known to be the least stable component of HEP software systems and the blackboard approach has been very effective at reducing the impact of this instability, from the Zebra system of the Fortran days to the InfoBus Java components architecture. The trade-off for the data/knowledge objects separation is that knowledge objects have to identify data objects they want to post or retrieve from the blackboard. It is crucial to develop a data model optimized for the required access patterns and yet flexible enough to accommodate the unexpected ones. Starting from the recent Event Data Model workshops and discussions as well as from the experience of other
HENP systems such as BaBar, CDF, CLEO, D0, we identified some requirements that would not be readily satisfied using the existing Gaudi Event Data Model:

- Identify (collections of) DataObjects based on their type.
- Identify (collections of) DataObjects based on the identifier of the Algorithm which added them to the Transient Data Store (TDS).
- A scheme that allows developers optionally to define key classes tailored to their DataObjects and to use the keys to store and retrieve the objects from the TDS.
- A well-defined and supported access control policy: as will be discussed later on, objects stored into the TDS will be “almost read-only” with the adoption of a lock mechanism.
- A mechanism to hide as much as possible the details of the TDS access from the algorithmic code, using the standard C++ iterator and/or pointer syntax.
- The same mechanism should be used to express associations among Data Objects.
- A mechanism to group related DataObjects into a developer-defined view that provides a high-level alternative access scheme to the store.
- A mechanism to define a flexible “cache-fault policy” (that is to say a way to create an object requested by the user and not yet in the TDS). This should include the functionality to reconstruct a DataObject on demand.

### 8.2.1 System Features and Entities

A StoreGate will allow algorithmic code to interact with the TDS. When an Algorithm invokes the StoreGate retrieve method, it is returned a DataHandle which points to the desired DataObject in the Transient Data Store. The DataHandle defines the interface to access the DataObjects retrieved. From the user perspective a DataHandle behaves as a C++ pointer/iterator, and its first implementation pointer is indeed a C++ const pointer. It will have to evolve into a more complex object to satisfy some of the requirements mentioned above. A complete implementation of the DataHandle should include:

- Begin/end methods providing iterators over contained objects,
- Enforcement of an almost-const access policy, whereby the DataHandle checks, upon dereference, if the client is authorized to modify a DataObject in the TDS.
- An update method used (presumably by the TDS) to mark the current Data Object the DataHandle refers to as “obsolete” when, for example, a new event is read in.
- Support for persistable object associations (e.g. track associatedHits),
- The ability for the client to view the data in different ways.
- A user-defined Key can be used while recording/retrieving the DataObject. In the prototype, the key object can be a simple string passed to the StoreGate record method or a more complex object that defines the necessary operators to return a string. Behind the scenes, the string is added to the Event Data Service object “pathname” thus providing the backward Gaudi compatibility.
A user-defined Selector can be provided that can either select amongst several DataObjects of the same type OR select on the ContainedObjects of a DataObject (provided the DataObject is a collection of ContainedObjects).

A virtual Proxy mechanism is used to represent Data Object instances that are not yet in the TDS. The Proxy will define the procedure to create these instances, either by reading them from persistent storage or by reconstructing them on demand. The Proxy can also be used, in conjunction with the DataHandle, to implement, if required, complex access control policies for the Data Objects.

8.2.2 Characteristics of the Transient Data Store

All objects in the transient data store inherit from a common base class (DataObject). The TDS supports storage of either collections (of several contained objects) or single objects. It is, however, assumed that in most instances, clients will store collections in the Transient Store; for example a collection of Track Objects (TrackCollection) or a collection of electron candidates. A collection contains objects of only one type. However, several types of objects can inherit from a base-type and be stored in the collection as the base-type. Multiple instances of the same collection or objects can be recorded with the TDS. The TDS contains DataObjects that are either persistable or required for communicating between Algorithms. An almost const-access policy will be established for accessing data objects in the Transient Store. The DataObject recorded to the TDS can be modified until locked by the client. Once locked, it becomes a read-only DataObject. Refer to Section Appendix 8.7 for details on how clients are expected to adhere to the Store Access Policy.

Each collection or object in the TDS will be associated with a History object. The History object will contain specific information on which Algorithms created the object and the configuration of these as defined by their Properties and other environmental information such as calibration and alignment information. The purpose of the History object is two-fold: it will precisely define how the object was created such that it is reproducible at a later stage, and the client can select on the basis of the information in the History Object. An example of the latter is as follows:

“Clusters in the EM calorimeter are found in two different ways in the reconstruction process, using a cone algorithm and a nearest neighbour algorithm. In each instance, cluster objects are stored in a ClusterCollection. Hence there will be two ClusterCollections (same type) in the TDS each simple reflecting a different algorithm. Since the History Object carries this information, a downstream client will be able to request EM cluster made using a specific algorithm.”

Tagging an object in the TDS by a downstream client will be forbidden. An example of a tag may be to simply mark it as having been used for the convenience of a single client or associating it with objects which have been produced at a later time in the reconstruction chain (so called forward pointing of objects). That is a “hit object” can not be explicitly tagged as used or modified to reflect which track (constructed later in time) it is associated with. This is simply because the same hit collection may be used by other clients for whom such tagging may be irrelevant. However tagging or forward pointing of DataObjects must be supported in other ways such as with association objects.
8.3 The StoreGate Prototype

The StoreGate defines the interface through which the algorithmic code will access the Transient Data Store (TDS). The prototype StoreGate extends the existing Gaudi TDS, providing the following extra functionalities:

- Type-safe access to DataObjects.
- Keyed access to DataObjects.
- Controlled access to DataObjects
- Simultaneous access to all DataObjects of a given type in the TDS
- Since in the medium term the StoreGate may replace or be merged with the existing Gaudi EventDataService, it is important to maintain compatibility with the current interface. In the prototype, a new Athena Service - called StoreGateSvc, provides templated methods to record and retrieve Data Objects by type. It is implemented over the existing Gaudi EventDataService.

Additional functionalities, such as AutoHandles, user defined data views, Smart inter object relationships, as mentioned in the StoreGate Design Document [5] are not yet available. This document will be updated as and when such functionalities are introduced.

- DataObjects in the TDS are organized by type. A DataObject can be assigned a client-defined key while recording. When an object is recorded with StoreGate, the TDS becomes the owner of the data object. *The delete method of the Data Object must not be called by the client.* See section 8.5 for more information.

- For multiple objects of the same type recorded in the TDS, the key (if one is assigned) must be unique. That is the same key can not be assigned to two objects in the TDS of the same type. Note that the key is optional - you need not assign a key while recording.

- There are several ways to retrieve data objects from the TDS. Either a single DataObject (the last object entered in the store) or a Keyed DataObject or a list of DataObjects of the same type can be retrieved from the TDS. See the section on how to retrieve DataObjects from the Transient Data Store for more details.

- The granularity of the object registered in the TDS is a DataObject. The DataObject can be a Collection of `ContainedObjects` or a single Object.

The StoreGate WriteData example demonstrates how to create a DataObject (MyDataObj) and record it in the TDS with and without a key. It records 3 data objects of the same type, one with a key. In addition, it also shows how to create a collection of contained objects (MyContObj) and record it in the TDS. The ReadData algorithm example shows how to retrieve these DataObjects in different ways from the TDS.
8.4 Creating a DataObject

The DataObject to be recorded in the TDS may either be a single object or a Collection. For the latter case, the collection consists of several ContainedObjects. The following examples show how to create a DataObject (single or Collection) and a ContainedObject.

8.4.1 Creating a single DataObject

Listing 8.1 is an example on how to create a single DataObject called “MyDataObj”.

The code sample below and the following ones are taken from the ReadData and WriteData example algorithms of StoreGate package, available in the Atlas repository at offline/Control/StoreGate/example.

- The single object to be recorded to the TDS, MyDataObj, must inherit from DataObject.
- It must also define a static constant Class ID, which must be unique for each class.

- The following DataObject has a single data member (m_dat). Note that it provides an accessor method val() to the data member. It is important that these methods are declared “const” (as shown below) to allow access to these data members when the Store Access Policy comes into effect. Const methods are methods that do not change the Object but simply allow access to the information in the object. Thus these methods are in effect “readonly” methods.

Listing 8.1 Example DataObject

```cpp
#include "Gaudi/Kernel/DataObject.h"

const static CLID CLID_MDO = 214222;

class MyDataObj : public DataObject
{
    public:

        MyDataObj() : DataObject () { };

        CLID& classID() { return CLID_MDO; }
        virtual CLID& clID() { return CLID_MDO; }
        int val() const {return m_dat; }
        void val(int i) { m_dat = i; }
        virtual ~MyDataObj() { };

    private:

        int m_dat;

};
```
8.4.2 Creating a Collection of ContainedObjects

Listing 8.2 shows an example on how to create a DataObject called “MyDataObj” that is a collection of "MyContObj" objects which inherit from ContainedObject. The ContainedObject in this example is “MyContObj” - an example of which is shown in Section 8.4.3.

- The DataObject to be recorded to the TDS, MyDataObj, must inherit from a templated Collection Base Class that Gaudi provides. There are two such base classes available: ObjectVector and ObjectList. Both are templated in the type of the ContainedObject (In this case “MyContObj”).

- As with the single object, your collection DataObject must define a static constant Class ID, which must be unique for each class.

- Note that the DataObject itself may have data members, in addition to being a collection of ContainedObjects. Listing 8.2 shows how to create a MyDataObj which has a single data member (m_dat). It provides a const accessor method val() which returns the data member. It is important that these methods are declared “const” (as shown below) to allow access to these data members when the Store Access Policy comes into effect. Const methods are methods that do not change the Object but simply allow access to the information in the object. Thus these methods are in effect “readonly” methods.

- See Section 8.6.1 on retrieving data objects on how to access the contained objects within a collection.

Listing 8.2 Example Collection of Contained Objects

```cpp
#include "Gaudi/Kernel/DataObject.h"
#include "Gaudi/Kernel/ObjectVector.h"

const static CLID CLID_MDO = 214234;

class MyContObj;

class MyDataObj : public ObjectVector<MyContObj>
{
public:
    MyDataObj() : DataObject () { };

    CLID& classID() { return CLID_MDO;}
    virtual CLID& clID() { return CLID_MDO; }

    int val() const {return m_dat;}
    void val(int i) { m_dat = i;}
    virtual ~MyDataObj() { };

private:
    int m_dat;
};
```
8.4.3 Creating a ContainedObject

In the previous sub-section, we created a DataObject ("MyDataObj") that was a collection of ContainedObjects ("MyContObj"). The following example shows you how to create the ContainedObject. Note that the ContainedObject is never directly inserted into the TDS, but must be “pushed back” into the Collection “MyDataObj”.

- The ContainedObject shown in Listing 8.3 has two data members: time and ID, a set method (which is non-const and initializes the data members), and const accessor methods time(), id() methods to the data members. As indicated earlier, declare all accessor methods as “const”.

- Unlike the "MyDataObj", it inherits from ContainedObject. Like “MyDataObj”, it also provides a static const class ID and methods to retrieve them.

Listing 8.3 Example ContainedObject

```cpp
#include "Gaudi/Kernel/ContainedObject.h"

class MyContObj: public ContainedObject {

public:

    MyContObj(): ContainedObject(){};
    ~MyContObj(){};

    static const CLID& classID();
    virtual const CLID& clID() const;

    void set(float t, int i) { m_time = t; m_channelID = i; }

    float time() const { return m_time; }
    int id() const { return m_channelID; }

private:

    float m_time;
    int m_channelID;
};

//MyContObj.cxx
#include "MyContObj.h"

static const CLID CLID_MYCONTOBJ = 214215;

static const CLID& MyContObj::classID() { return CLID_MYCONTOBJ; }
const CLID& MyContObj::clID() const { return CLID_MYCONTOBJ; }
```
8.5 Recording a DataObject

A DataObject may be recorded to the TransientStore using the StoreGateSvc. Each Algorithm has access to the storeGateSvc() and can therefore invoke the record method. Once recorded, the TDS has ownership of the DataObject. Hence clients must not call the delete of the DataObject.

All recorded DataObjects are organized according to their type. A key can be optionally associated to a DataObject when this is recorded. At present it is the client responsibility to explicitly lock (set constant) the DataObjects which should no longer be modified.

8.5.1 Recording DataObjects without keys

In the execute method of WriteData, create a DataObject of type MyDataObj, and set its member data:

**Listing 8.4** Fragment 1 of WriteData Algorithm

```c++
MyDataObj* dobj = new MyDataObj();
dobj -> val(10);
```

Record it in the TDS as shown in Listing 8.5: (Note that the pointer to storeGateSvc() is available to you if you are an "Algorithm"

**Listing 8.5** Fragment 2 of WriteData Algorithm

```c++
StatusCode sc = storeGateSvc()->record(dobj);
if (sc.isFailure())
{
    log << MSG::ERROR << "Error recording dobj in TDS" << endreq;
}
```

To ensure that no downstream algorithms can accidentally modify it, you must lock the recorded DataObject, presumably at the end of the algorithm who made it.

**Listing 8.6** Fragment 3 of WriteData Algorithm

```c++
dobj->setConst();
```

In a future release, we will introduce a mechanism which will lock by default every new DataObject recorded by an Algorithm in its execute() method.
8.5.2 Recording DataObjects with keys

Record a DataObject using a string key as shown in Listing 8.7:

Listing 8.7 Record a DataObject with a string key

```cpp
std::string keystring = "MyDobjName";
StatusCode sc = storeGateSvc()->record(dobj, keystring);
if (sc.isFailure())
   { log << MSG::ERROR << "Error recording dobj in TDS" << endreq; }
   ....
//lock dobj to prevent any further non-const access
dobj->setConst()
```

You need not necessarily use a string as a key. Every class that defines conversion operators to and from a string is a valid key type. Listing 8.8 shows an example of this:

Listing 8.8 Example of a valid key

```cpp
class ValidKey
{
   public:

   ValidKey(const std::string aString);
   operator std::string() const;
   ....
};
```

In the future these two constraints may be relaxed and a valid key will only have to provide an ordering operator (e.g. operator <).

8.6 Retrieving a DataObject

A DataObject in StoreGate is accessed via a DataHandle, which is templated in the type of the object and has a syntax similar to a standard iterator over a list of DataObjects retrieved from the TDS.

DataObjects may be retrieved by:

- Providing no keys, in which case the last DataObject recorded in the TDS will be returned to you.
- By providing a key, in which case there will be a unique match. Note that a keyed access is possible only if you recorded the DataObject with a key. Since two DataObjects of the same type cannot have the same key, you are assured of a unique DataObject (provided of course it exists).

- You can ask for a list of DataObjects of a given type. You will be returned a begin and end iterator over all the DataObjects of that type (whether or not it was recorded with a key).

For each of these three modes you have the option to require const access to the retrieved objects or non constant access. In the latter case, if the DataObject has already been locked, as it will normally be the case, the retrieve operation will not return the locked object (and will fail if no unlocked matching objects are available).

### 8.6.1 Retrieving DataObjects without a key

In the execute method of ReadData, get the last MyDataObj recorded in the TDS (recall that we recorded three MyDataObj in TDS, one with a key). The example shown in Listing 8.9 will retrieve the last of the three MyDataObj from the TDS:

**Listing 8.9** Retrieve a DataObject without a key

```cpp
const DataHandle<MyDataObj> dhandle;
StatusCode sc = storeGateSvc()->retrieve(dhandle); [1]
if (sc.isFailure())
{
    log << MSG::ERROR << "Could not retrieve MyDataObj" << endreq;
}
log << MSG::INFO << "Data Object Value: "
    << dhandle->val() << endreq;
    dhandle->use_me(); [3]
```

Notes:

1. This retrieves the DataObject of type MyDataObj
2. You use a DataHandle just like a pointer. This prints the value.
3. This calls the function member "use_me" of MyDataObj.
Typically MyDataObj will be a collection containing several objects. You may iterate over these Contained Objects using the code shown in Listing 8.10:

Listing 8.10  Iterate over Contained Objects

```cpp
MyDataObj::iterator first = dhandle->begin();
MyDataObj::iterator last = dhandle->end();
for (; first != last; ++first)
{
  (*first)->use_me();
}
```

where (*first) is the pointer to the Contained Object; use_me() is a method in this Contained Object and ++first iterates over the Contained Objects.

8.6.2 Retrieving a keyed DataObject

In the WriteData example, three MyDataObj were recorded in the TDS, but only one with a key. The key was a string key containing “MyDataObjName”. Here we will show how to retrieve that DataObject. Note that since no other MyDataObj can be registered with a key “MyDataObjName”, you will retrieve at most one object. Note further that this is not necessarily the last DataObject of this type recorded in the TDS.

Listing 8.11  Retrieve a keyed DataObject

```cpp
std::string keystring = "MyDataObjName";
DataHandle<MyDataObj> dhandle; //non-const access required. May fail!!!
StatusCode sc = storeGateSvc()->retrieve(dhandle,keystring);
if (sc.isFailure())
{
  log << MSG::ERROR << "Error retrieving keyed MyDataObj" << endreq;
}
```

If the key does not match any MyDataObj in the store, you will get an error and the returned handle will not point to any MyDataObj. You will also get an error and a null handle, if you provide a non-const DataHandle (as in the example above), and the required object has already been locked. Unless you know what you are doing, always provide a const DataHandle as argument to the retrieve methods.

Use the retrieved handle as described in the previous paragraph: the handle behaves as a pointer: dhandle->use_me();

Also remember that the DataObject the handle points to will typical be a collection of ContainedObjects
8.6.3 Retrieving all DataObjects of a given type

Recall that there were three MyDataObj recorded in the example: WriteData.cxx, one with a key. We can retrieve all three of these DataObjects as shown in Listing 8.12:

Listing 8.12 Retrieve all DataObjects of a given type

```cpp
const DataHandle<MyDataObj> dbegin;
const DataHandle<MyDataObj> dend;
StatusCode sc = storeGateSvc()->retrieve(dbegin, dend);
if (sc.isFailure())
{
    log << MSG::ERROR << "Error Retrieving MyDataObj's" << endreq;
}
```

Note that `dbegin` points to the first DataObject. Hence you can use it just as before:

```cpp
dbegin->use_me();
```

Since DataHandle has the syntax of an iterator, `dend` points past the end of the list of returned objects: `dend->use_me()` will have unpredictable, but most likely fatal, results.

Since you have retrieved all the DataObjects of the same type, you can iterate over these DataObjects as shown in Listing 8.13: Note that you are now iterating over DataObjects - not ContainedObjects. Each of these DataObjects may be a collection of ContainedObjects.

Listing 8.13 Iterate over all retrieved DataObjects

```cpp
for (; dbegin != dend; ++dbegin) // Loop Over DataObjects
{
    dbegin->invoke_method_in_DataObject();
}
```

Note again that `dbegin` always points to a DataObject, never a ContainedObject. To iterate over the ContainedObjects, do as before (this piece of code is obviously within the loop over DataObject):

Listing 8.14 Iterate over Contained Objects

```cpp
MyDataObj::iterator first = dbegin->begin();
MyDataObj::iterator last = dbegin->end();
for (; first != last; ++first)
{
    (*first)->invoke_method_in_ContObj();
}
```
8.7 Store Access Policy

The current release (Release 1.3.2) has some of the tools to implement a flexible access policy to objects stored in the TDS. The TDS owns the DataObjects that have been recorded into it. A DataObject in the store may not be deleted by the client code. Client code is allowed to modify objects in the store but only until the DataObject is declared "complete" and "set constant" (locked) by its creator.

Locking an object prevents the clients downstream of the creator to modify the reconstructed objects by mistake. For example, a tracking algorithm that retrieves a hit collection from the store does not own those hits and therefore may not modify them. Remember that these hits can be used later by another tracking algorithm for which your modifications may be meaningless or even plain wrong. We do not want the results of the second tracking algorithm to change when you run the second algorithm before or after the first! If we want to have reproducible results from several million lines of reconstruction code it is vital to preserve the state of every DataObject which has been reconstructed and "published" in the TDS to be used by others. However, well controlled modifications are desirable. The access policy allows multiple algorithms to collaborate in reconstructing a DataObject. As an example, consider a Calorimeter Clustering package makes cluster objects and subsequently may make corrections, based on the Calorimeter information alone that are independent of any downstream analysis. Since cluster finding and corrections are more or less independent, it may be desirable to have the two decoupled: a cluster-finding sub-algorithm and a correction sub-algorithm executed in some Cluster-Maker Algorithm environment. In such a scenario:

- The cluster-finding sub-algorithm will create the cluster object and record it in the TDS.
- The correction sub-algorithm will retrieve the cluster object from the TDS and perform some correction (hence modifying the object in the TDS).
- The main Cluster-Maker algorithm that coordinates the various factions of the calorimeter clustering will lock the object before returning control to the framework
- A downstream electron-finding algorithm can use these cluster objects in a readonly mode and create a new electron object. Additional corrections may be determined and included in the calculation of quantities of the final electron object, but the data of the original cluster object cannot be changed. These additional correction, if necessary, must be preserved elsewhere.
- Note that it may be necessary for downstream algorithms to redo the clustering based on new information that has become available only at a later stage. This is indeed possible - it may do so, by creating a new cluster collection in the TDS. Since cluster collections in TDS can be keyed (and later can also be associated with a history object), these are uniquely identifiable objects in the TDS.
In general, the strategy that is proposed to be adopted by algorithms creating DataObjects to be recorded in the TDS is shown in Listing 8.15:

**Listing 8.15 Example of store access strategy**

```cpp
MyMainAlgorithm::execute() {
    // Create a New Data Object:
    MyDataObj* dobj = new MyDataObj();

    // Record it in the Transient Data Store:
    StatusCode sc = storeGateSvc()->record(dobj);

    dobj->modify();    // OK, can modify data object
    sub_Algorithm_A->execute(); // OK, sub Algorithms may
    sub_Algorithm_B->execute(); // also modify "MyDataObj"
    dobj->modify_more();    // OK, more changes if necessary
    dobj->setLocked();    // Lock Data Object
    dobj->modify_after_lock();    // WRONG, NOT ALLOWED HERE.
}
```

Note that in addition to the main algorithm that created the Data Object, sub_Algorithm_A and sub_Algorithm_B can retrieve “MyDataObj” from the TDS and modify it as it has not yet been locked. Once the DataObject is locked, it can not be modified by any algorithm. We therefore expect:

- A specific package is responsible for the creation of DataObject(s). For example, a track fitting package makes use of Hit Collections and creates a Tracking Collection in the TDS. A Calorimeter Clustering package creates Cluster Collections in TDS. An electron-id package creates an electron collection containing electron candidates in the TDS.

- Each package may choose to break its algorithms into finer sub-algorithms, several of these sub-algorithms jointly responsible for the final output. The main algorithm will therefore create the collection and must lock it before it returns its control to the framework. All modifications to the Data Object are done either within the body of the main algorithm or by sub-algorithms executed before the lock has been established.

- If the client chooses to use a Collection created by a different package (such as the Track Fitting package using a Hit Collection made by a TrackHit package), it must access it in a readonly mode.

- To allow downstream algorithms to access your data objects in “readonly” mode, the dataobjects must supply “const” accessor methods as discussed in Section 8.4.

- An error condition will occur if the client attempts to retrieve a locked DataObject in a non-const mode and the DataObject is locked.

With the locking mechanism in place, we are now in a position to provide a default locking mechanism to enforce an experiment-wide policy. Whatever this policy will be (currently we are discussing whether the locking should happen at the end of an algorithm and/or sequence execution) it is good practice for the developers to explicitly add a “setConst” method invocation when they are finished working on a DataObject. This will document the author intent, and it will also insulate their codes from any future change to the default access policy.
Chapter 9
Data dictionary

9.1 Overview

In contrast to the other Chapters in this User Guide, this one does not describe existing functionality, but rather gives an overview to new functionality that will be made available in a future release. It describes the role, scope, and implementation of a "Data Dictionary" in the Athena Architecture. It is a condensation of a snapshot of a separate working document.

9.1.1 Definition of Terms

The term data dictionary is being used in ATLAS as a catch phrase for several related, but distinct concepts and techniques. We categorize these concepts into three general categories:

- **Introspection/Reflection/Object Description/Run-Time Typing**
  This refers to objects in program memory with the ability to describe themselves in a programmatic way through a public API such that they can be manipulated without a priori knowledge of the specific class/type of the object.

- **Code Generation**
  This refers to a process of generating code for performing a specific task from a generic description/input file.

- **Self-Describing External Data Representation (e.g. Data Files)**
  This refers to external data representations (e.g. file formats, on-wire data formats) which contain metadata describing the payload of the data file, etc.

Each of these concepts plays a different set of roles in an architecture dependent upon a data dictionary.

NB: Throughout this document we will use the word *object* without (necessarily) referring to an actual instance of a C++/Java/other class. A potentially better phrase
might be programmatic entity or construct (which could denote components, structs, common blocks, streams, files, etc.). However, since it is almost certain that most, if not all, such entities in Athena will in fact correspond to real objects, we will not dwell on this distinction.

**NB:** The term data dictionary connotes a certain technical implementation which we do not (necessarily) advocate. This term implies a central repository containing the information about those objects being described. This is one possible implementation of some of the concepts in this paper. However, it is also possible (perhaps even desirable for some purposes) that the information resides internal to the object (for example). In this Chapter we use the term Data Dictionary as a generic term denoting the broad concept and not as an indication of where the object description resides.

### 9.1.2 Roles of a Data Dictionary (DD)

The motivation for implementing a DD in Athena can be illustrated by listing the potential roles that a DD can play within the Architecture. These roles include:

- **Data Tools Integration**
  Tools needed to perform many tasks within the overall system which are not specific to a particular data object type, including browsing and editing, visualization, simple or standard transformations, etc.

- **Interactive Data Queries**
  The ability to interrogate a data object as to its shape and content from the interactive user interface (e.g. scripting language interface).

- **Automatic/Semiautomatic Persistency**
  The ability to apply generic converters and/or to automatically generate specialized converters and/or to generate converter skeletons for subsequent, manual customization.

- **Schema Evolution (Version-Safe Persistency)**
  Most data objects in the Event Data Model (EDM) typically change multiple times throughout their lifecycle within a HEP experiment. Support for evolution of the EDM schema is critical.

- **Multi-Language Support**
  Components written in multiple programming languages (e.g. C++, Java, FORTRAN) will be used in the context of the Athena framework. These components need to exchange data in a language independent interchange representation.

- **Component Independence, Stability, & Robustness**
  The ability to write code to a reflection API such that changes external to a component do not necessitate any code changes within the component leads naturally to code stability and, consequently, code robustness. Architectural attention to physical interdependencies of framework components also benefits from the use of such an API.
9.1.3 Implementation Strategy

Because the Data Dictionary can be used for a wide variety of purposes, a strategic choice must be made for the implementation plan. The choice can be stated as: Do we first implement only a single DD function (such as binding to a particular persistency service) as fully as possible? Or do we first implement multiple DD functions (e.g. multiple persistency services, multi-language support, binding to multiple general tools, etc) but with each one minimally functional?

There are good arguments for both approaches. However, we believe that the first option is the most attractive for two reasons.

1. The full range of possible functions of the Data Dictionary cannot be defined at the beginning of the development of the Data Dictionary.

   Though we have enumerated above some of the roles that the DD can play, most are general roles and not specific tasks. Implementing these functions one-by-one helps to ensure that we do not lock ourselves into an approach which is difficult to extend to other, unforeseen DD functions.

2. Providing a single, fully functional (or almost fully functional) tool will encourage physicists to begin using the tool immediately.

   If we initially provide multiple functions which illustrate the eventual functionality of the DD, but which are not fleshed out enough to be immediately useful to end-users, we run the risk of discouraging such physicists from using the DD until it reaches a more mature stage of development. If, on the other hand, we provide a single DD function and make it sufficiently complete to actually improve end-users' productivity, we both demonstrate the utility of the DD and provide immediate help to users doing work today. Thus encouraging immediate adoption of the DD by users.

9.1.4 Data Dictionary Language

One of the most visible implementation decisions of a DD for Athena is the choice of the computer language used in the dictionary. A very non-comprehensive list of choices includes:

- Declarative C++ (i.e. C++ headers)
- IDL - Interface Definition Language (http://www.omg.org/)
- Declarative Java (http://java.sun.com/)
- ODL - Object Definition Language (http://www.odmg.org/)
- DDL - Data Definition Language (http://www.objectivity.com/)
- XML - Extensible Markup Language (http://www.w3.org/)
- Home grown solution

Each of these choices have Pros and Cons. Although the choice of Data Dictionary Language (In this document we will use ADL generically for Athena Dictionary Language without implying a concrete
choice.) is an important decision, it is arguably not a make-or-break decision. However, because of the extreme visibility of such a choice and the likelihood that any choice will draw criticism (warranted or not) from some quarter, the decision-making process must be very well documented and technically motivated.

### 9.1.5 Code Generation

Code generation tools are parser-based tools which process the ADL, construct an Abstract Syntax Tree (AST), and drive compiler backends (emitters). Often a code generation tool can be used to eliminate tedious and error-prone rote programming.

With the choice of a real computer language as the basis of the Data Dictionary, it becomes imperative that a real parser be used to compile the DD language and realize the DD functionality. Experience has shown that multiple back-ends (emitters) for the parser are necessary (see Figure 1). The reality of a possible evolution of ADL suggests that the compiler front-end should be replaceable.

![Figure 1](image_url)

**Figure 9.1** ADL Parser with multiple back-ends

### 9.1.6 Data Dictionary Design

The Data Dictionary will be used at multiple ATLAS sites by a large number of people. Scalability, distributability, and ease of use of the DD are important design criteria.

The ease of use of the Data Dictionary depends largely upon the target audience. For the typical physicist, the DD must be easy to use and must have clear benefits or it will not be used. For more sophisticated users (e.g. Core Programmers), the burden of learning a new system or language must be outweighed by the long-term benefits.

One objection which should guide our thinking on ease of use is the sometimes heard statement: "Physicists do not want to learn a new language.". This argument, out of context, can be compelling. Physicists don't want to learn new, complicated languages to do the same thing they can do with an already familiar language. Some of this resistance can be seen in the slower than expected move to C++ in ATLAS.

However, experience has shown that if the new language is sufficiently simple and/or intuitive, and the benefits are obvious, a new language will quickly become popular and widely used by those
Physicists most active in software development. Once this happens, the barrier for subsequent physicists becomes quite low as there are many experts to whom she can go for help and advice and many examples from which to learn.

9.1.7 Time Line & Milestones

The development and deployment plan for the Athena Data Dictionary is:

- January 2001
  - Evaluation of ADL Candidates & Tools Complete
  - Athena Dictionary Language (ADL) Chosen
  - ADL Compiler Front End (CFE) Chosen
  - Subset of ADL Compiler Back Ends (CBEs) Defined
- April 2001
  - Prototype DD Implementation
  - Standalone ADL CFE Functional
  - One ADL CBE Implemented & Associated Functionality Integrated in Athena
  - Athena Dictionary Language (ADL) Frozen
- September 2001
  - Data Dictionary Deployed
  - ADL CFE Integrated with ATLAS Release Tools
  - ADL Reflection API Available in Athena
  - Multiple ADL CBEs Implemented & Available & Integrated in Athena

9.1.8 Bibliography

http://www.javaworld.com/
http://iago.lbl.gov/dsl/
http://www.swig.org/
http://electra.lbl.gov/papers/CDFDBCodeGen.html
http://electra.lbl.gov/talks/Cdf_SWDec98_codegen.ppt
http://www.hep.net/chep95/html/abstract/abs_18.htm
http://www.hep.net/chep95/html/slides/t78/index.htm
http://www.hep.net/chep95/html/abstract/abs_78.htm
http://www.ifh.de/CHEP97/paper/322.ps
http://sources.redhat.com/sourcenav/
Chapter 10
Detector Description

10.1 Overview

This chapter is a place holder for documenting how to access the detector description data in the Athena transient detector data store. A detector description implementation based on XML exists in the LHCb extensions to Gaudi but it is not distributed with the framework.

The Gaudi architecture aims to shield the applications from the details of the persistent detector description and calibration databases. Ideally, the detector will be described in a logically unique detector description database (DDDB), containing data from many sources (e.g. editors and CAD tools for geometry data, calibration and alignment programs, detector control system for environmental data) as shown in Figure 10.1. The job of the Gaudi detector data service is to populate the transient detector data store with a snapshot of the detector description, which is valid for the event currently being analysed. Conversion services can be invoked to provide different transient representations of the same persistent data, appropriate to the specific application. For example, detector simulation, reconstruction and event display all require a geometry description of the detector, but with different levels of detail. In the Gaudi architecture it is possible to have a single, generic, persistent geometry description, from which a set of different representations can be extracted and made available to the data processing applications.

The LHCb implementation of the detector description database describes the logical structure of the detector in terms of a hierarchy of detector elements and the basic geometry in terms of volumes, solids and materials, and provides facilities for customizing the generic description to many specific detector needs. This should allow to develop detector specific code which can provide geometry answers to questions from the physics algorithms. The persistent representation of the LHCb detector description is based on text files in XML format. An XML editor that understands the detector description semantics has been developed.
Figure 10.1 Overview of the Detector Description model.
Chapter 11
Histogram facilities

11.1 Overview

The histogram data store is one of the data stores discussed in Chapter 2. Its purpose is to store statistics based data and user created objects that have a lifetime of more than a single event (e.g. histograms).

As with the other data stores, all access to data is via a service interface. In this case it is via the IHistogramSvc interface, which is derived from the IDaProviderSvc interface discussed in Chapter 7. The user asks the Histogram Service to book a histogram and register it in the histogram data store. The service returns a pointer to the histogram, which can then be used to fill and manipulate the histogram.

The histograms themselves are booked and manipulated using four interfaces as defined by the AIDA (Abstract Interfaces for Data Analysis) project. These interfaces are documented on the AIDA web pages: http://wwwinfo.cern.ch/asd/lhc++/AIDA/. The Athena implementation uses the transient part of HTL (Histogram Template Library, http://wwwinfo.cern.ch/asd/lhc++/HTL/), provided by LHC++.

The histogram data model is shown in Figure 11.1. The interface IHistogram is a base class, which is used for management purposes. It is not a complete histogram interface, it should not be used by the users. Both interfaces IHistogram1D and IHistogram2D are derived from IHistogram, and use by reference the IAxis interface. Users can book their 1D or 2D histograms in the histogram data store in the same type of tree structure as the event data. Concrete 1D and 2D histograms derive from the DataObject in order to be storable.
11.2 The Histogram service.

An instance of the histogram data service is created by the application manager. After the service has been initialised, the histogram data store will contain a root directory “/stat” in which users may book histograms and/or create sub-directories (for example, in the code fragment below, the histogram is stored in the subdirectory “/stat/simple”). A suggested naming convention for the sub-directories is given in Section 1.2.3.

As discussed in Section 5.2, the Algorithm base class defines a member function

IHistogramSvc* histoSvc()
which returns a pointer to the IHistogramSvc interface of the standard histogram data service. Access to any other non-standard histogram data service (if one exists) must be sought via the ISvcLocator interface of the application manager as discussed in section 13.2.

### 11.3 Using histograms and the histogram service

An example code fragment illustrating how to book a 1D histogram and place it in a directory within the histogram data store, and a simple statement which fills that histogram is shown here:

```cpp
// Book 1D histogram in the histogram data store
m_hTrackCount = histoSvc()->book( "/stat/simple", 1, "TrackCount", 100, 0., 3000. );
SmartDataPtr<MyTrackVector> particles( eventSvc(), "/Event/MyTracks" )
if ( 0 != particles ) {
    // Filling the track count histogram
    m_hTrackCount->fill(particles->size(), 1.);
}
```

The parameters of the book function are the directory in which to store the histogram in the data store, the histogram identifier, the histogram title, the number of bins and the lower and upper limits of the X axis. 1D histograms with fixed and variable binning are available. In the case of 2D histograms, the book method requires in addition the number of bins and lower and upper limits of the Y axis.

If using HBOOK for persistency, the histogram identifier should be a valid HBOOK histogram identifier (number), must be unique and, in particular, must be different from any n-tuple number. Even if using another persistency solution (e.g. ROOT) it is recommended to comply with the HBOOK constraints in order to make the code independent of the persistency choice.

The call to histoSvc()->book(...) returns a pointer to an object of type IHistogram1D (or IHistogram2D in the case of a 2D histogram). All the methods of this interface can be used to further manipulate the histogram, and in particular to fill it, as shown in the example. Note that this pointer is guaranteed to be non-null, the algorithm would have failed the initialisation step if the histogram data service could not be found. On the contrary the user variable particles may be null (in case of absence of tracks in the transient data store and in the persistent storage), and the fill statement would fail - so the value of particles must be checked before using it.

Algorithms that create histograms will in general keep pointers to those histograms, which they may use for filling operations. However it may be that you wish to share histograms between different algorithms. Maybe one algorithm is responsible for filling the histogram and another algorithm is responsible for fitting it at the end of the job. In this case it may be necessary to look for histograms within the store. The mechanism for doing this is identical to the method for locating event data objects within the event data store, namely via the use of smart pointers, as discussed in section 7.8.
11.4 Persistent storage of histograms

By default, Athena does not produce a persistent histogram output. The options exist to write out histograms either in HBOOK or in ROOT format.

11.4.1 HBOOK persistency

The HBOOK conversion service converts objects of types IHistogram1D and IHistogram2D into a form suitable for storage in a standard HBOOK file. In order to use it you first need to tell Athena where to find the HbookCnv shared library. If you are using CMT, this is done by adding the following line to the CMT requirements file:

```
use HbookCnv v9*
```

You then have to tell the application manager to load this shared library and to create the HBOOK conversion service, by adding the following lines to your job options file:

```
ApplicationMgr.DLLs += {"HbookCnv"};
ApplicationMgr.HistogramPersistency = "HBOOK";
```

Finally, you have to tell the histogram persistency service the name of the output file:

```
HistogramPersistencySvc.OutputFile = "histo.hbook";
```

11.4.2 ROOT persistency

The ROOT conversion service converts objects of types IHistogram1D and IHistogram2D into a form suitable for storage in a standard ROOT file. In order to use it you first need to tell Athena where to find the RootHistCnv shared library. If you are using CMT, this is done by adding the following line to the CMT requirements file:

```
use RootHistCnv v3*
```
You then have to tell the application manager to load this shared library and to create the HBOOK conversion service, by adding the following lines to your job options file:

```cpp
ApplicationMgr.DLLs += {"RootHistCnv"};
ApplicationMgr.HistogramPersistency = "ROOT";
```

Finally, you have to tell the histogram persistency service the name of the output file:

```cpp
HistogramPersistencySvc.OutputFile = "histo.rt";
```
Chapter 12
N-tuple and Event Collection facilities

12.1 Overview

In this chapter we describe facilities available in Athena to create and retrieve n-tuples. We discuss how Event Collections, which can be considered an extension of n-tuples, can be used to make preselections of event data. Finally, we explore some possible tools for the interactive analysis of n-tuples.

12.2 N-tuples and the N-tuple Service

User data - so called n-tuples - are very similar to event data. Of course, the scope may be different: a row of an n-tuple may correspond to a track, an event or complete runs. Nevertheless, user data must be accessible by interactive tools such as PAW or ROOT.

Athena n-tuples allow to freely format structures. Later, during the running phase of the program, data are accumulated and written to disk.

The transient image of an n-tuple is stored in a Athena data store which is connected to the n-tuple service. Its purpose is to store user created objects that have a lifetime of more than a single event.

As with the other data stores, all access to data is via a service interface. In this case it is via the INTupleSvc interface which extends the IDataProviderSvc interface. In addition the interface to the n-tuple service provides methods for creating n-tuples, saving the current row of an n-tuple or retrieving n-tuples from a file. The n-tuples are derived from DataObject in order to be storable, and are stored in the same type of tree structure as the event data. This inheritance allows to load and locate n-tuples on the store with the same smart pointer mechanism as is available for event data items (c.f. Chapter 7).
12.2.1 Access to the N-tuple Service from an Algorithm.

The Algorithm base class defines a member function which returns a pointer to the INTupleSvc interface.

```
INTupleSvc* ntupleSvc()
```

The n-tuple service provides methods for the creation and manipulation of n-tuples and the location of n-tuples within the persistent store.

The top level directory of the n-tuple transient data store is called “/NTUPLES”. The next directory layer is connected to the different output streams: e.g. “/NTUPLES/FILE1”, where FILE1 is the logical name of the requested output file for a given stream. There can be several output streams connected to the service. In case of persistency using HBOOK, “FILE1” corresponds to the top level RZ directory of the file (...the name given to HROPEN). From then on the tree structure is reflected with normal RZ directories (caveat: HBOOK only accepts directory names with less than 8 characters! It is recommended to keep directory names to less than 8 characters even when using another technology (e.g. ROOT) for persistency, to make the code independent of the persistency choice.).

12.2.2 Using the N-tuple Service.

When defining an n-tuple the following steps must be performed:

- The n-tuple tags must be defined.
- The n-tuple must be booked and the tags must be declared to the n-tuple.
- The n-tuple entries have to be filled.
- The filled row of the n-tuple must be committed.
- Persistent aspects are steered by the job options.

In the following an attempt is made to explain the different steps. Please note that when using HBOOK for persistency, the n-tuple number must be unique and, in particular, that it must be different from any histogram number. This is a limitation imposed by HBOOK. It is recommended to keep this number unique even when using another technology (e.g. ROOT) for persistency, to make the code independent of the persistency choice.

12.2.2.1 Defining N-tuple tags

When creating an n-tuple it is necessary to first define the tags to be filled in the n-tuple. Typically the tags belong to the filling algorithm and hence should be provided in the Algorithm’s header file. Currently the following data types are supported: `bool`, `long`, `float` and `double`. `double` types (Fortran REAL*8) need special attention if using HBOOK for persistency: the n-tuple structure must be
defined in a way that aligns double types to 8 byte boundaries, otherwise HBOOK will complain. In addition PAW cannot understand double types. Listing 12.1 illustrates how to define n-tuple tags:

Listing 12.1 Definition of n-tuple tags from the Ntuples.WriteAlg.h example header file.

```cpp
1: NTuple::Item<long> m_ntrk; // A scalar item (number)
2: NTuple::Array<bool> m_flag; // Vector items
3: NTuple::Array<long> m_index;
4: NTuple::Array<float> m_px, m_py, m_pz;
5: NTuple::Matrix<long> m_hits; // Two dimensional tag
```

12.2.2.2 Booking and Declaring Tags to the N-tuple

When booking the n-tuple, the previously defined tags must be declared to the n-tuple. Before booking, the proper output stream (file) must be accessed. The target directory is defined automatically.

Listing 12.2 Creation of an n-tuple in a specified directory and file.

```cpp
1: // Access the output file
2: NTupleFilePtr file1(ntupleSvc(), "/NTUPLES/FILE1");
3: if ( file1 ) {
4: // First: A column wise N tuple
5: NTuplePtr nt(ntupleSvc(), "/NTUPLES/FILE1/MC/1");
6: if ( !nt ) { // Check if already booked
7: nt=ntupleSvc()->book("/NTUPLES/FILE1/MC",1,CLID_ColumnWiseTuple,
8: "Hello World");
9: if ( nt ) {
10: // Add an index column
11: status = nt->addItem ("Ntrack", m_ntrk, 0, 5000 );
12: // Add a variable size column of type float (length=length of index col)
13: status = nt->addItem ("px", m_ntrk, m_px);
14: status = nt->addItem ("py", m_ntrk, m_py);
15: status = nt->addItem ("pz", m_ntrk, m_pz);
16: // Another one, but this time of type bool
17: status = nt->addItem ("flg",m_ntrk, m_flag);
18: // Another one, type integer, numerical numbers must be within [0, 5000]
18: status = nt->addItem ("idx",m_ntrk, m_index, 0, 5000 );
19: // Add 2-dim column: [0:m_ntrk][0:2]; numerical numbers within [0, 8]
20: status = nt->addItem ("hit",m_ntrk, m_hits, 2, 0, 8 );
21: }
22: else { // did not manage to book the N tuple....
23: return StatusCode::FAILURE;
24: }
25: }
```

Tags which are not declared to the n-tuple are invalid and will cause an access violation at run-time.

12.2.2.3 Filling the N-tuple

Tags are usable just like normal data items, where
Items<TYPE> are the equivalent of numbers: bool, long, float.

Array<TYPE> are equivalent to 1 dimensional arrays: bool[size], long[size], float[size]

Matrix<TYPE> are equivalent to an array of arrays or matrix: bool[dim1][dim2].

There is no implicit bounds checking possible without a rather big overhead at run-time. Hence it is up to the user to ensure the arrays do not overflow.

When all entries are filled, the row must be committed, i.e. the record of the n-tuple must be written.

**Listing 12.3** Filling an n-tuple.

```cpp
1: m_ntrk = 0;
2: for( MyTrackVector::iterator i = mytracks->begin(); i != 
   mytracks->end(); i++ ) {
3:   const HepLorentzVector& mom4 = (*i)->fourMomentum();
4:   m_px[m_ntrk] = mom4.px();
5:   m_py[m_ntrk] = mom4.py();
6:   m_pz[m_ntrk] = mom4.pz();
7:   m_index[m_ntrk] = cnt;
8:   m_flag[m_ntrk] = (m_ntrk%2 == 0) ? true : false;
9:   m_hits[m_ntrk][0] = 0;
10:  m_hits[m_ntrk][1] = 1;
11:  m_ntrk++;
12:  // Make sure the array(s) do not overflow.
13:  if ( m_ntrk > m_ntrk->range().distance() ) break;
14: }
15: // Commit N tuple row.
16: status = ntupleSvc()->writeRecord("/NTUPLES/FILE1/MC/1");
17: if ( !status.isSuccess() ) {
18:   log << MSG::ERROR << "Cannot fill id 1" << endreq;
19: }
20: }
```

### 12.2.2.4 Reading N-tuples

Although n-tuples intended for interactive analysis, they can also be read by a regular program. An example of reading back such an n-tuple is given in Listing 12.4. Notice line 8, where an example is given of preselecting rows of the n-tuple according to given criteria. This option is only possible if supported by the underlying database used to make the n-tuple persistent. Currently it is possible to preselect rows from n-tuples written in ROOT format and from relational databases using ODBC\(^1\). Note that the syntax of the query is also affected by the underlying technology: while an ODBC database will accept any SQL query, the ROOT implementation understands only the "And" and "Or" SQL operators - but it does understand the full C++ syntax (an example is given in section 12.3.2).

---

1. The ODBC implementation exists in the LHCb extensions to Gaudi. It is not distributed with Athena
12.2.3 N-tuple Persistency

12.2.3.1 Input and Output File Specification

Conversion services exist to convert n-tuple objects into a form suitable for persistent storage in a number of storage technologies. In order to use this facility it is necessary to add the following line in the job options file:

```
NTupleSvc.Output = {"FILE1 DATAFILE='tuples.hbook' TYP='HBOOK' OPT='NEW'",
"FILE2 ...",
... "FILEN ...");
```

where `<tuples.hbook>` should be replaced by the name of the file to which you wish to write the n-tuple. FILE1 is the logical name of the output file - it could be any other string. A similar option `NTupleSvc.Input` exists for n-tuple input.

The following is a complete overview of all possible options:

- **DATAFILE=’<file-specs>’**
  Specifies the datafile (file name) of the output stream.

- **TYP=’<typ-spec>’**
  Specifies the type of the output stream. Currently supported types are:
  - **HBOOK**: Write in HBOOK RZ format.
  - **ROOT**: Write as a ROOT tree.
There is also weak support for the following database types:\footnote{The implementation for MS Access and other ODBC compliant databases is available in the LHCb extensions to Gaudi. It is not distributed with Athena.}

- SQL Server
- MySQL
- Oracle ODBC

These database technologies are supported through their ODBC interface. They were tested privately on local installations. However all these types need special setup to grant access to the database.

Except for the HBOOK data format, you need to specify the use of the technology specific persistency package (i.e. GaudiRootDb) in your CMT requirements file and to load explicitly in the job options the DLLs containing the generic (GaudiDb) and technology specific (GaudiRootDb) implementations of the database access drivers:

```
ApplicationMgr.DLLs += { "GaudiDb", "GaudiRootDb" };  
```

- OPT='<opt-spec>'
  - NEW, CREATE, WRITE: Create a new data file. Not all implementations allow to over-write existing files.
  - OLD, READ: Access an existing file for read purposes
  - UPDATE: Open an existing file and add records. It is not possible to update already existing records.

- SVC='<service-spec>' (optional)
  Connect this file directly to an existing conversion service. This option however needs special care. It should only be used to replace default services.

- AUTHENTICATION='<authentication-specs>' (optional)
  For protected datafiles (e.g. Microsoft Access) it can happen that the file is password protected. In this case the authentication string allows to connect to these databases. The connection string in this case is the string that must be passed to ODBC, for example:

  `AUTH=SERVER=server_host;UID=user_name;PWD=my_password;'`

  - All other options are passed without any interpretation directly to the conversion service responsible to handle the specified output file.

For all options at most three leading characters are significant: DATAFILE=<...>, DATABASE=<...> or simply DATA=<...> would lead to the same result.

The handling of row wise n-tuples does not differ. However, only individual items (class NTuple::Item) can be filled, no arrays and no matrices. Since the persistent representation of row
wise n-tuples in HBOOK is done by floats only, the first row of each row wise n-tuple contains the type information - when looking at a row wise n-tuple with PAW make sure to start at the second event!

12.3 Event Collections

Event collections or, to be more precise, event tag collections, are used to minimize data access by performing preselections based on small amounts of data. Event tag data contain flexible event classification information according to the physics needs. This information could either be stored as flags indicating that the particular event has passed some preselection criteria, or as a small set of parameters which describe basic attributes of the event. Fast access is required for this type of event data.

Event tag collections can exist in several versions:

- Collections recorded during event processing stages from the online, reconstruction, reprocessing etc.
- Event collections defined by analysis groups with pre-computed items of special interest to a given group.
- Private user defined event collections.

Starting from this definition an event tag collection can be interpreted as an n-tuple which allows to access the data used to create the n-tuple. Using this approach any n-tuple which allows access to the data is an event collection.

Event collections allow pre-selections of event data. These pre-selections depend on the underlying storage technology.

First stage pre-selections based on scalar components of the event collection. First stage preselection is not necessarily executed on your computer but on a database server e.g. when using ORACLE. Only the accessed columns are read from the event collection. If the criteria are fulfilled, the n-tuple data are returned to the user process. Preselection criteria are set through a job options, as described in section 12.3.2.

The second stage pre-selection is triggered for all items which passed the first stage pre-selection criteria. For this pre-selection, which is performed on the client computer, all data in the n-tuple can be used. The further preselection is implemented in a user defined function object (functor) as described in section 12.3.2. Athena algorithms are called only when this pre-selector also accepts the event, and normal event processing can start.

12.3.1 Writing Event Collections

Event collections are written to the data file using a Athena sequencer. A sequencer calls a series of algorithms, as discussed in section 5.2. The execution of these algorithms may terminate at any point of
the series (and the event not selected for the collection) if one of the algorithms in the sequence fails to pass a filter.

12.3.1.1 Defining the Address Tag

The event data is accessed using a special n-tuple tag of the type

```cpp
NTuple::Item<IOpaqueAddress*> m_evtAddress
```

It is defined in the algorithm’s header file in addition to any other ordinary n-tuple tags, as described in section 12.2.2.1. When booking the n-tuple, the address tag must be declared like any other tag, as shown in Listing 12.1. It is recommended to use the name "Address" for this tag.

**Listing 12.1** Connecting an address tag to an n-tuple.

```cpp
1: NTuplePtr nt(ntupleSvc(), "/NTUPLES/EvtColl/Collection");
1: ... Book N-tuple
2: // Add an event address column
3: StatusCode status = nt->addItem ("Address", m_evtAddress);
```

The usage of this tag is identical to any other tag except that it only accepts variables of type IOpaqueAddress - the information necessary to retrieve the event data.

12.3.1.2 Filling the Event Collection

At fill time the address of the event must be supplied to the Address item. Otherwise the n-tuple may be written, but the information to retrieve the corresponding event data later will be lost. Listing 12.2 also demonstrates the setting of a filter to steer whether the event is written out to the event collection.

**Listing 12.2** Fill the address tag of an n-tuple at execution time:

```cpp
1: SmartDataPtr<Event> evt(eventSvc(),"/Event");
2: if ( evt ) {
3: ... Some data analysis deciding whether to keep the event or not
4: // keep_event=true if event should be written to event collection
5: setFilterPassed( keep_event );
6: m_evtAddrColl = evt->address();
7: }
```

12.3.1.3 Writing out the Event Collection

The event collection is written out by an EvtCollectionStream, which is the last member of the event collection Sequencer. Listing 12.3 (which is taken from the job options of EvtCollection example), shows how to set up such a sequence consisting of a user written Selector algorithm (which could for example contain the code in Listing 12.2), and of the EvtCollectionStream.
12.3.2 Reading Events using Event Collections

Reading event collections as the input for further event processing in Athena is transparent. The main change is the specification of the input data to the event selector:

Listing 12.4 Connecting an address tag to an n-tuple.

```
1: EventSelector.Input = {
2: "COLLECTION='Collection' ADDRESS='Address'
   DATAFILE='MyEvtCollection.root' TYP='ROOT' SEL='(Ntrack>80)'
   FUN='EvtCollectionSelector'"
3: };
```

These tags need some explanation:

- **COLLECTION**
  Specifies the sub-path of the n-tuple used to write the collection. If the n-tuple which was written was called e.g. "/NTUPLES/FILE1/Collection", the value of this tag must be "Collection".

- **ADDRESS (optional)**
  Specifies the name of the n-tuple tag which was used to store the opaque address to be used to retrieve the event data later. This is an optional tag, the default value is "Address". Please use this default value when writing, conventions are useful!

- **SEL (optional):**
  Specifies the selection string used for the first stage pre-selection. The syntax depends on the database implementation; it can be:

  - SQL like, if the event collection was written using ODBC.
    Example: (NTrack>200 AND Energy>200)

  - C++ like, if the event collection was written using ROOT.
    Example: (NTrack>200 && Energy>200).
    Note that event collections written with ROOT also accept the SQL operators 'AND' instead of '&&' as well as 'OR' instead of '| |'. Other SQL operators are not supported.
- **FUN (optional)**
  Specifies the name of a function object used for the second-stage preselection. An example of a such a function object is shown in Listing 12.5. Note that the factory declaration on line 16 is mandatory in order to allow Athena to instantiate the function object.

- The **DATAFILE** and **TYP** tags, as well as additional optional tags, have the same meaning and syntax as for n-tuples, as described in section 12.2.3.1.

Listing 12.5  Example of a function object for second stage pre-selections.

```cpp
1: class EvtCollectionSelector : public NTuple::Selector {
2:   NTuple::Item<long> m_ntrack;
3:   public:
4:     EvtCollectionSelector(IInterface* svc) : NTuple::Selector(svc) { }
5:     virtual ~EvtCollectionSelector() { }
6:     /// Initialization
7:     virtual StatusCode initialize(NTuple::Tuple* nt) {
8:       return nt->item("Ntrack", m_ntrack);
9:     }
10:    /// Specialized callback for NTuples
11:    virtual bool operator()(NTuple::Tuple* nt) {
12:      return m_ntrack>cut;
13:    }
14:  }
15: };
16:  ObjectFactory<EvtCollectionSelector> EvtCollectionSelectorFactory
```

### 12.4 Interactive Analysis using N-tuples

n-tuples are of special interest to the end-user, because they can be accessed using commonly known tools such as PAW, ROOT or Java Analysis Studio (JAS). In the past it was not a particular strength of the software used in HEP to plug into many possible persistent data representations. Except for JAS, only proprietary data formats are understood. For this reason the choice of the output format of the data depends on the preferred analysis tool/viewer. In the following an overview is given over the possible data formats.

In the examples below the output of the GaudiExample/NTuple.write program was used.

#### 12.4.1 HBOOK

This data format is used by PAW. PAW can understand this and only this data format. Files of this type can be converted to the ROOT format using the h2root data conversion program. The use of PAW in the long term is deprecated.
12.4.2 ROOT

This data format is used by the interactive ROOT program. Using the helper library TBlob located in the package GaudiRootDb it is possible to interactively analyse the n-tuples written in ROOT format. However, access is only possible to scalar items (int, float, ...) not to arrays.

Analysis is possible through directly plotting variables:

```root
1: gSystem->Load("D:/mycmt/GaudiRootDb/v3/Win32Debug/TBlob");
2: TFile* f = new TFile("tuple.root");
3: TTree* t = (TTree*)f->Get("<local>_MC_ROW_WISE_2");
4: t->Draw("pz");
```

or using a ROOT macro interpreted by ROOT's C/C++ interpreter (see for example the code fragment interactive.C shown in Listing 12.6):

```root
0: gSystem->Load("D:/mycmt/GaudiRootDb/v3/Win32Debug/TBlob");
1: .L ./v8/NTuples/interactive.C
2: interactive("./v8/NTuples/tuple.root");
```

More detailed explanations can be found in the ROOT tutorials (http://root.cern.ch).

**Listing 12.6** Interactive analysis of ROOT n-tuples: interactive.C

```c
1: void interactive(const char* fname) {
2:   TFile *input = new TFile(fname);
3:   TTree *tree = (TTree*)input->Get("<local>_MC_ROW_WISE_2");
4:   if (0 == tree) {
5:     printf("Cannot find the requested tree in the root file!\n");
6:     return;
7:   }
8:   Int_t ID, OBJSIZE, NUMLINK, NUMSYM;
9:   TBlob *BUFFER = 0;
10:  Float_t px, py, pz;
11:  tree->SetBranchAddress("ID", &ID);
12:  tree->SetBranchAddress("OBJSIZE", &OBJSIZE);
13:  tree->SetBranchAddress("NUMLINK", &NUMLINK);
14:  tree->SetBranchAddress("NUMSYM", &NUMSYM);
15:  tree->SetBranchAddress("BUFFER", &BUFFER);
16:  tree->SetBranchAddress("px", &px);
17:  tree->SetBranchAddress("py", &py);
18:  tree->SetBranchAddress("pz", &pz);
19:  Int_t nbytes = 0;
20:  for (Int_t i = 0, nentries = tree->GetEntries(); i<nentries; i++) {
21:    nbytes += tree->GetEntry(i);
22:    printf("Trk#=%d PX=%f PY=%f PZ=%f\n", i, px, py, pz);
23:  }
24:  printf("I have read a total of %d Bytes.\n", nbytes);
25:  delete input;
26: }
```
Chapter 13
Framework services

13.1 Overview

Services are generally sizeable components that are setup and initialized once at the beginning of the job by the framework and used by many algorithms as often as they are needed. It is not desirable in general to require more than one instance of each service. Services cannot have a “state” because there are many potential users of them so it would not be possible to guarantee that the state is preserved in between calls.

In this chapter we describe how services are created and accessed, and then give an overview of the various services, other than the data access services, which are available for use within the Athena framework. The Job Options service, the Message service, the Particle Properties service, the Chrono & Stat service, the Auditor service, the Random Numbers service and the Incident service are available in this release. The Tools service is described in Chapter 14.

We also describe how to implement new services for use within the Athena environment. We look at how to code a service, what facilities the Service base class provides and how a service is managed by the application manager.

13.2 Requesting and accessing services

The Application manager creates a certain number of services by default. These are the default data access services (EventDataSvc, DetectorDataSvc, HistogramDataSvc, NTupleSvc), the default data persistency services (EventPersistencySvc, DetectorPersistencySvc, HistogramPersistencySvc) and the framework services described in this chapter and in Chapter 14 (JobOptionsSvc, MessageSvc,
ParticlePropertySvc, ChronoStatSvc, AuditorSvc, RndmGenSvc, IncidentSvc, ToolSvc).

Additional services can be requested via the job options file, using the property ApplicationMgr.ExtSvc. In the example below this option is used to create a specific type of persistency service:

**Listing 13.1** Job Option to create additional services

```
ApplicationMgr.ExtSvc += { "DbEventCnvSvc/RootEvtCnvSvc" };  
```

Once created, services must be accessed via their interface. The Algorithm base class provides a number of accessor methods for the standard framework services, listed on lines 25 to 35 of Listing 5.1 on page 24. Other services can be located using the templated service function. In the example below we use this function to return the IParticlePropertySvc interface of the Particle Properties Service:

**Listing 13.2** Code to access the IParticlePropertySvc interface from an Algorithm

```
#include "GaudiKernel/IParticlePropertySvc.h"
...
IParticlePropertySvc* m_ppSvc;
StatusCode sc = service("ParticlePropertySvc", m_ppSvc);
if (sc.isFailure) {
  ...
```

In components other than Algorithms, which do not provide the service function, you can locate a service using the serviceLocator function:

**Listing 13.3**

```
#include "GaudiKernel/IParticlePropertySvc.h"
...
IParticlePropertySvc* m_ppSvc;
StatusCode sc = serviceLocator()->getService("ParticlePropertySvc",
  IID_IParticlePropertySvc,
  reinterpret_cast<IInterface*&>(m_ppSvc));
if (sc.isFailure) {
  ...
```
13.3 The Job Options Service

The Job Options Service is a mechanism which allows to configure an application at run time, without
the need to recompile or relink. The options, or properties, are set via a job options file, which is read in
when the Job Options Service is initialised by the Application Manager. In what follows we describe
the format of the job options file, including some examples.

13.3.1 Algorithm, Tool and Service Properties

In general a concrete Algorithm, Tool or Service will have several data members which are used to
control execution. These data members can be of a basic data type (int, float, etc.) or class
(Property) encapsulating some common behaviour and higher level of functionality.

13.3.1.1 SimpleProperties

Simple properties are a set of classes that act as properties directly in their associated Algorithm, Tool
or Service, replacing the corresponding basic data type instance. The primary motivation for this is to
allow optional bounds checking to be applied, and to ensure that the Algorithm, Tool or Service itself
doesn't violate those bounds. Available SimpleProperties are:

- int ==> IntegerProperty or SimpleProperty<int>
- double ==> DoubleProperty or SimpleProperty<double>
- bool ==> BooleanProperty or SimpleProperty<bool>)
- std::string ==> StringProperty or SimpleProperty<std::string>

and the equivalent vector classes

- std::vector<int> ==> IntegerArrayProperty or
  SimpleProperty<std::vector<int>>
- etc.
The use of these classes is illustrated by the EventCounter class.

Listing 13.4  EventCounter.h

```cpp
#include "GaudiKernel/Algorithm.h"
#include "GaudiKernel/Property.h"

class EventCounter : public Algorithm {
public:
    EventCounter( const std::string& name, ISvcLocator* pSvcLocator );
    ~EventCounter( );
    StatusCode initialize();
    StatusCode execute();
    StatusCode finalize();
private:
    IntegerProperty m_frequency;
    int m_skip;
    int m_total;
};
```

Listing 13.5  EventCounter.cpp

```cpp
#include "GaudiAlg/EventCounter.h"
#include "GaudiKernel/MsgStream.h"
#include "GaudiKernel/AlgFactory.h"

static const AlgFactory<EventCounter> Factory;
const IAlgFactory& EventCounterFactory = Factory;

EventCounter::EventCounter(const std::string& name, ISvcLocator* pSvcLocator) :
    Algorithm(name, pSvcLocator),
    m_skip ( 0 ),
    m_total( 0 )
{
    declareProperty( "Frequency", m_frequency=1 ); // [1]
    m_frequency.setBounds( 0, 1000 ); // [2]
}

StatusCode
EventCounter::initialize()
{
    MsgStream log(msgSvc(), name());
    log << MSG::INFO << name( )
    << "::initialize - Frequency: "
    << m_frequency << endreq; // [3]
    return StatusCode::SUCCESS;
}
```

Notes:
1. A default value may be specified when the property is declared.
2. Optional upper and lower bounds may be set (see later).
3. The value of the property is accessible directly using the property itself.

In the Algorithm constructor, when calling declareProperty, you can optionally set the bounds using any of:

```cpp
setBounds( const T& lower, const T& upper );
setLower ( const T& lower );
setUpper ( const T& upper );
```

There are similar selectors and modifiers to determine whether a bound has been set etc., or to clear a bound.

```cpp
bool hasLower( )
bool hasUpper( )
T lower( )
T upper( )
void clearBounds( )
void clearLower( )
void clearUpper( )
```

You can set the value using the "=" operator or the set functions

```cpp
bool set( const T& value )
bool setValue( const T& value )
```

The function value indicates whether the new value was within any bounds and was therefore successfully updated. In order to access the value of the property, use:

```cpp
m_property.value( );
```

In addition there's a cast operator, so you can also use `m_property` directly instead of `m_property.value()`.

### 13.3.1.2 CommandProperty

CommandProperty is a subclass of StringProperty that has a handler that is called whenever the value of the property is changed. Currently that can happen only during the job initialization so it is not terribly useful. Alternatively, an Algorithm could set the property of one of its sub-algorithms. However, it is envisaged that Athena will be extended with a scripting language such that properties can be modified during the course of execution.

The relevant portion of the interface to CommandProperty is:

```cpp
class CommandProperty : public StringProperty {
public:
    [...]
    virtual void handler( const std::string& value ) = 0;
    [...]
};
```
Thus subclasses should override the `handler()` member function, which will be called whenever the property value changes. A future development is expected to be a `ParsableProperty` (or something similar) that would offer support for parsing the string.

### 13.3.2 Job options file format

The job options file has a well-defined syntax (similar to a simplified C++-Syntax) without data types. The data types are recognised by the “Job Options Compiler”, which interprets the job options file according to the syntax (described in Appendix C together with possible compiler error codes).

The job options file is an ASCII-File, composed logically of a series of statements. The end of a statement is signalled by a semicolon “;” - as in C++.

Comments are the same as in C++, with ‘//’ until the end of the line, or between ‘/*’ and ‘*/’.

There are four constructs which can be used in a job options file:

- Assignment statement
- Append statement
- Include directive
- Platform dependent execution directive

#### 13.3.2.1 Assignment statement

An assignment statement assigns a certain value (or a vector of values) to a property of an object or identifier. An assignment statement has the following structure:

```plaintext
<Object / Identifier> . < Propertyname > = < value >;
```

The first token (`Object / Identifier`) specifies the name of the object whose property is to be set. This must be followed by a dot (‘.’)

The next token (`Propertyname`) is the name of the option to be set, as declared in the `declareProperty()` method of the `IProperty` interface. This must be followed by an assign symbol (‘=’).

The final token (`value`) is the value to be assigned to the property. It can be a vector of values, in which case the values are enclosed in array brackets (‘{,’}’), and separated by commas (,). The token must be terminated by a semicolon (‘;’).

The type of the value(s) must match that of the variable whose value is to be set, as declared in `declareProperty()`. The following types are recognised:
**Boolean-type, written as true or false.**

e.g. `true; false;`

**Integer-type, written as an integer value** (containing one or more of the digits '0', '1', '2', '3', '4', '5', '6', '7', '8', '9')

e.g.: `123; -923;` or in scientific notation, e.g.: `1.2e2;`

**Real-type (similar to double in C++), written as a real value** (containing one or more of the digits '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' followed by a dot '.' and optionally one or more of digits again)

e.g.: `123.; -123.45;` or in scientific notation, e.g. `12.5e7;`

**String type, written within a pair of double quotes ('"')**

e.g.: "I am a string"; (Note: strings without double quotes are not allowed!)

**Vector of the types above, within array-brackets ('{', '}'), separated by a comma (',')**

e.g.: `{true, false, true};
e.g.: `{124, -124, 135e2};
e.g.: `{123.53, -23.53, 123., 12.5e2};
e.g.: `{"String 1", "String 2", "String 3"};

A single element which should be stored in a vector must be within array-brackets without a comma

e.g. `{true};
e.g. `{"String"};

A vector which has already been defined earlier in the file (or in included files) can be reset to an empty vector

e.g. `{};

### 13.3.2.2 Append Statement

Because of the possibility of including other job option files (see below), it is sometimes necessary to extend a vector of values already defined in the other job option file. This functionality is provided by the append statement.

An append statement has the following syntax:

```
<Object / Identifier> . < Propertyname > += < value >;
```

The only difference from the assignment statement is that the append statement requires the '+=' symbol instead of the '=' symbol to separate the Propertyname and value tokens.
The value must be an array of one or more values

e.g. {true};
e.g. {"String"};
e.g.: {true, false, true};
e.g.: {124, -124, 135e2};
e.g.: {123.53, -23.53, 123., 12.5e2};
e.g.: {"String 1", "String 2", "String 3"};

The job options compiler itself tests if the object or identifier already exists (i.e. has already been
defined in an included file) and the type of the existing property. If the type is compatible and the object
exists the compiler appends the value to the existing property. If the property does not exists then the
append operation "+=" behaves as assignment operation "=".

### 13.3.2.3 Including other Job Option Files

It is possible to include other job option files in order to use pre-defined options for certain objects. This
is done using the `#include` directive:

```
#include "filename.ext"
```

The “filename.ext” can also contain the path where this file is located. The include directive can
be placed anywhere in the job option file, but it is strongly recommended to place it at the very top of
the file (as in C++).

It is possible to use environment variables in the `#include` statement, either standalone or as part of a
string. Both Unix style (“$environmentvariable”) and Windows style
(“%environmentvariable%”) are understood (on both platforms!)

As mentioned above, you can append values to vectors defined in an included job option file. The
interpreter creates these vectors at the moment he interprets the included file, so you can only append
elements defined in a file included before the append-statement!

As in C/C++, an included job option file can include other job option files. The compiler checks itself
whether the include file is already included or not, so there is no need for `#ifndef` statements as in C
or C++ to check for multiple including.

Sometimes it is necessary to over-ride a value defined previously (maybe in an include file). This is
done by using an assign statement with the same object and Propertyname. The last value assigned is
the valid value!
### 13.3.2.4 Platform dependent execution

The possibility exists to execute statements only according to the used platform. Statements within platform dependent clauses are only executed if they are asserted to the current used platform:

```c
#ifdef WIN32
(Platform-Dependent Statement)
#else (optional)
(Platform-Dependent Statement)
#endif
```

Only the variable WIN32 is defined! An `#ifdef WIN32` will check if the used platform is a Windows platform. If so, it will execute the statements until an `#endif` or an optional `#else`. On non-Windows platforms it will execute the code within `#else` and `#endif`. Alternatively one directly can check for a non-Windows platform by using the `#ifndef WIN32` clause.

### 13.3.3 Example

We have already seen an example of a job options file in Listing 4.2 on page 24. The use of the `#include` statement is demonstrated on line 2: the logical name `$STDOPTS` is defined in the `GaudiExamples` package, which contains a number of standard job options include files that can be used by applications.
13.4 The Standard Message Service

One of the components directly visible to an algorithm object is the message service. The purpose of this service is to provide facilities for the logging of information, warnings, errors etc. The advantage of introducing such a component, as opposed to using the standard `std::cout` and `std::cerr` streams available in C++ is that we have more control over what is printed and where it is printed. These considerations are particularly important in an online environment.

The Message Service is configurable via the job options file to only output messages if their “activation level” is equal to or above a given “output level”. The output level can be configured with a global default for the whole application:

```
// Set output level threshold (2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL)
MessageSvc.OutputLevel = 4;
```

and/or locally for a given client object (e.g. myAlgorithm):

```
myAlgorithm.OutputLevel = 2;
```

Any object wishing to print some output should (must) use the message service. A pointer to the IMessageSvc interface of the message service is available to an algorithm via the accessor method `msgSvc()`, see section 5.2. It is of course possible to use this interface directly, but a utility class called MsgStream is provided which should be used instead.

13.4.1 The MsgStream utility

The MsgStream class is responsible for constructing a Message object which it then passes onto the message service. Where the message is ultimately sent to is decided by the message service.

In order to avoid formatting messages which will not be sent because the verboseness level is too high, a MsgStream object first checks to see that a message will be printed before actually constructing it. However the threshold for a MsgStream object is not dynamic, i.e. it is set at creation time and remains the same. Thus in order to keep synchronized with the message service, which in principle could change its printout level at any time, MsgStream objects should be made locally on the stack when needed. For example, if you look at the listing of the HelloWorld class (see also Listing 13.1 below) you will note that MsgStream objects are instantiated locally (i.e. not using new) in all three of the IAlgorithm methods and thus are destructed when the methods return. If this is not done messages may be lost, or too many messages may be printed.

The MsgStream class has been designed to resemble closely a normal stream class such as `std::cout`, and in fact internally uses an ostrstream object. All of the MsgStream member functions write unformatted data; formatted output is handled by the insertion operators.
An example use of the `MsgStream` class is shown below.

**Listing 13.1 Use of a MsgStream object.**

```cpp
1: #include "GaudiKernel/MgsStream.h"
2: 
3: StatusCode myAlgo::finalize() {
4:  StatusCode status = Algorithm::finalise();
5:  MsgStream log(msgSvc(), name());
6:  if ( status.isFailure() ) {
7:     // Print a two line message in case of failure.
8:     log << MSG::ERROR << " Finalize failed" << endl
9:     << "Error initializing Base class." << endreq;
10:  } 
11:  else {
12:     log << MSG::DEBUG << "Finalize completed successfully" << endreq;
13:  }
14:  return status;
15: }
```

When using the `MsgStream` class just think of it as a configurable output stream whose activation is actually controlled by the first word (message level) and which actually prints only when “endreq” is supplied. For all other functionality simply refer to the C++ `ostream` class.

The “activation level” of the `MsgStream` object is controlled by the first expression, e.g. `MSG::ERROR` or `MSG::DEBUG` in the example above. Possible values are given by the enumeration below:

```cpp
enum MSG::Level { VERBOUS, DEBUG, INFO, WARNING, ERROR, FATAL };
```

Thus the code in Listing 13.1 will produce NO output if the print level of the message service is set higher than `MSG::ERROR`. In addition if the service’s print level is lower than or equal to `MSG::DEBUG` the “Finalize completed successfully” message will be printed (assuming of course it was successful).

**User interface**

What follows is a technical description of the part of the `MsgStream` user interface most often seen by application developers. Please refer to the header file for the complete interface.

**Insertion Operator**

The `MsgStream` class overloads the ‘<’ operator as described below.
MsgStream\& \texttt{operator \&\&}(\texttt{TYPE arg});

Insertion operator for various types. The argument is only formatted by the stream object
if the print level is sufficiently high and the stream is active. Otherwise the insertion
operators simply return. Through this mechanism extensive debug printout does not cause
large run-time overheads. All common base types such as \texttt{char}, \texttt{unsigned char},
\texttt{int}, \texttt{float}, etc. are supported

MsgStream\& \texttt{operator \&\&}(\texttt{MSG::Level level});

This insertion operator does not format any output, but rather (de)activates the stream’s
formatting and forwarding engine depending on the value of \texttt{level}.

\section*{Accepted Stream Manipulators}

The \texttt{MsgStream} specific manipulators are presented below, e.g. \texttt{endreq}: \texttt{MsgStream\&
endreq(MsgStream\& stream)}. Besides these, the common \texttt{ostream} and \texttt{ios} manipulators such
as \texttt{std::ends}, \texttt{std::endl},... are also accepted.

\begin{description}
\item[\texttt{endl}] Inserts a newline sequence. Opposite to the \texttt{ostream} behaviour this manipulator does not flush the
buffer. Full name: \texttt{MsgStream\& \: \texttt{endl}(MsgStream\& \: s)}
\item[\texttt{ends}] Inserts a null character to terminate a string. Full name: \texttt{MsgStream\& \: \texttt{ends}(MsgStream\& \: s)}
\item[\texttt{flush}] Flushes the stream’s buffer but does not produce any output! Full name: \texttt{MsgStream\&
flush(MsgStream\& \: s)}
\item[\texttt{endreq}] Terminates the current message formatting and forwards the message to the message service.
If no message service is assigned the output is sent to \texttt{std::cout}. Full name: \texttt{MsgStream\&
endreq(MsgStream\& \: s)}
\end{description}
13.5 The Particle Properties Service

The Particle Property service is a utility to find information about a named particle’s Geant3 ID, Jetset/Pythia ID, Geant3 tracking type, charge, mass or lifetime. The database used by the service can be changed, but by default is the same as that used by the LHCb SICB program. Note that the units conform to the CLHEP convention, in particular MeV for masses and ns for lifetimes. Any comment to the contrary in the code is just a leftover which has been overlooked!

13.5.1 Initialising and Accessing the Service

This service is created by adding the following line in the Job Options file:

```
// Create the particle properties service
ApplicationMgr.ExtSvc += { "ParticlePropertySvc" }; 
```

Listing 13.2 on page 96 shows how to access this service from within an algorithm.

13.5.2 Service Properties

The Particle Property Service currently only has one property: ParticlePropertiesFile. This string property is the name of the database file that should be used by the service to build up its list of particle properties. The default value of this property, on all platforms, is $LHCBDBASE/cdf/particle.cdf

13.5.3 Service Interface

The service implements the IParticlePropertySvc interface. In order to use it, clients must include the file GaudiKernel/IParticlePropertySvc.h.

The service itself consists of one STL vector to access all of the existing particle properties, and three STL maps, one to map particles by name, one to map particles by Geant3 ID and one to map particles by stdHep ID.

Although there are three maps, there is only one copy of each particle property and thus each property must have a unique particle name and a unique Geant3 ID. The third map does not contain all particles contained in the other two maps; this is because there are particles known to Geant but not to stdHep, such as Deuteron or Cerenkov. Although retrieving particles by name should be sufficient, the second and third maps are there because most often generated data stores a particle’s Geant3 ID or stdHep ID, and not the particle’s name. These maps speed up searches using the IDs.

---

1. This is an LHCb specific file. A generic implementation will be available in a future release of Athena
The IParticlePropertySvc interface provides the following functions:

**Listing 13.2 The IParticlePropertySvc interface.**

```cpp
// IParticlePropertySvc interface:
// Create a new particle property.
// Input: particle, String name of the particle.
// Input: geantId, Geant ID of the particle.
// Input: jetsetId, Jetset ID of the particle.
// Input: type, Particle type.
// Input: charge, Particle charge (/e).
// Input: mass, Particle mass (MeV).
// Input: tlife, Particle lifetime (ns).
// Return: StatusCode - SUCCESS if the particle property was added.
virtual StatusCode push_back( const std::string& particle, int geantId, int
jetsetId, int type, double charge, double mass, double tlife );

// Create a new particle property.
// Input: pp, a particle property class.
// Return: StatusCode - SUCCESS if the particle property was added.
virtual StatusCode push_back( ParticleProperty* pp );

// Get a const reference to the begining of the map.
virtual const_iterator begin() const;

// Get a const reference to the end of the map.
virtual const_iterator end() const;

// Get the number of properties in the map.
virtual int size() const;

// Retrieve a property by geant id.
// Pointer is 0 if no property found.
virtual ParticleProperty* find( int geantId );

// Retrieve a property by particle name.
// Pointer is 0 if no property found.
virtual ParticleProperty* find( const std::string& name );

// Retrieve a property by StdHep id
// Pointer is 0 if no property found.
virtual ParticleProperty* findByStdHepID( int stdHepId );

// Erase a property by geant id.
virtual StatusCode erase( int geantId );

// Erase a property by particle name.
virtual StatusCode erase( const std::string& name );

// Erase a property by StdHep id
virtual StatusCode eraseByStdHepID( int stdHepId );
```
The IParticlePropertySvc interface also provides some typedefs for easier coding:

```cpp
typedef ParticleProperty* mapped_type;
typedef std::map<int, mapped_type, std::less<int>> MapID;
typedef std::map<std::string, mapped_type, std::less<std::string>> MapName;
typedef std::map<int, mapped_type, std::less<int>> MapStdHepID;
typedef IParticlePropertySvc::VectPP VectPP;
typedef IParticlePropertySvc::const_iterator const_iterator;
typedef IParticlePropertySvc::iterator iterator;
```

### 13.5.4 Examples

Below are some extracts of code from the LHCb ParticleProperties example to show how one might use the service:

**Listing 13.3** Code fragment to find particle properties by particle name.

```cpp```
// Try finding particles by the different methods
log << MSG::INFO << "Trying to find properties by Geant3 ID..." << endreq;
ParticleProperty* pp1 = m_ppSvc->find( 1 );
if ( pp1 ) log << MSG::INFO << *pp1 << endreq;
log << MSG::INFO << "Trying to find properties by name..." << endreq;
ParticleProperty* pp2 = m_ppSvc->find( "e+" );
if ( pp2 ) log << MSG::INFO << *pp2 << endreq;
log << MSG::INFO << "Trying to find properties by StdHep ID..." << endreq;
ParticleProperty* pp3 = m_ppSvc->findByStdHepID( 521 );
if ( pp3 ) log << MSG::INFO << *pp3 << endreq;
```

**Listing 13.4** Code fragment showing how to use the map iterators to access particle properties.

```cpp```
// List all properties
log << MSG::DEBUG << "Listing all properties..." << endreq;
for( IParticlePropertySvc::const_iterator i = m_ppSvc->begin(); i != m_ppSvc->end(); i++ ) {
    if ( *i ) log << *(i) << endreq;
}
```

```cpp```
13.6 The Chrono & Stat service

The Chrono & Stat service provides a facility to do time profiling of code (Chrono part) and to do some statistical monitoring of simple quantities (Stat part). The service is created by default by the Application Manager, with the name “ChronoStatSvc” and service ID extern const CLID& IID_IChronoStatSvc To access the service from inside an algorithm, the member function chronoSvc() is provided. The job options to configure this service are described in Appendix B, Table B.19.

13.6.1 Code profiling

Profiling is performed by using the chronoStart() and chronoStop() methods inside the codes to be profiled, e.g:

```cpp
/// ...
IClonoStatSvc* svc = chronoSvc();
/// start
svc->chronoStart( "Some Tag" );
/// here some user code are placed:
... 
/// stop
svc->chronoStop( "SomeTag" );
```

The profiling information accumulates under the tag name given as argument to these methods. The service measures the time elapsed between subsequent calls of `chronoStart()` and `chronoStop()` with the same tag. The latter is important, since in the sequence of calls below, only the elapsed time between lines 3 and 5 lines and between lines 7 and 9 lines would be accumulated:.

```cpp
1: svc->chronoStop("Tag");  
2: svc->chronoStop("Tag");  
3: svc->chronoStart("Tag"); 
4: svc->chronoStart("Tag"); 
5: svc->chronoStop("Tag"); 
6: svc->chronoStop("Tag"); 
7: svc->chronoStart("Tag"); 
8: svc->chronoStart("Tag"); 
9: svc->chronoStop("Tag");
```

The profiling information could be printed either directly using the `chronoPrint()` method of the service, or in the summary table of profiling information at the end of the job.

Note that this method of code profiling should be used only for fine grained monitoring inside algorithms. To profile a complete algorithm you should use the Auditor service, as described in section 13.7.
13.6.2 Statistical monitoring

Statistical monitoring is performed by using the stat() method inside user code:

```c++
1: /// ... Flag and Weight to be accumulated:
2: svc->stat( " Number of Tracks " , Flag , Weight );
```

The statistical information contains the "accumulated" flag, which is the sum of all Flags for the given tag, and the "accumulated" weight, which is the product of all Weights for the given tag. The information is printed in the final table of statistics.

In some sense the profiling could be considered as statistical monitoring, where the variable Flag equals the elapsed time of the process.

13.6.3 Chrono and Stat helper classes

To simplify the usage of the Chrono & Stat Service, two helper classes were developed: class Chrono and class Stat. Using these utilities, one hides the communications with Chrono & Stat Service and provides a more friendly environment.

13.6.3.1 Chrono

Chrono is a small helper class which invokes the chronoStart() method in the constructor and the chronoStop() method in the destructor. It must be used as an automatic local object.

It performs the profiling of the code between its own creation and the end of the current scope, e.g:

```c++
1: #include GaudiKernel/Chrono.h
2: /// ...
3: { // begin of the scope
4:   Chrono chrono( chronoSvc() , "ChronoTag" ) ;
5:   /// some codes:
6:   ...
7:   ///
8: } // end of the scope
9: /// ...
```

If the Chrono & Stat Service is not accessible, the Chrono object does nothing
13.6.3.2 Stat

Stat is a small helper class, which invokes the stat() method in the constructor.

```cpp
1: GaudiKernel/Stat.h
2: /// ...
3: Stat stat( chronoSvc() , "StatTag" , Flag , Weight ) ;
4: /// ...
```

If the Chrono & Stat Service is not accessible, the Stat object does nothing.

13.6.4 Performance considerations

The implementation of the Chrono & Stat Service uses two std::map containers and could generate a performance penalty for very frequent calls. Usually the penalty is small relative to the elapsed time of algorithms, but it is worth avoiding both the direct usage of the Chrono & Stat Service as well as the usage of it through the Chrono or Stat utilities inside internal loops:

```cpp
1: /// ...
2: { /// begin of the scope
3: Chrono chrono( chronoSvc() , "Good Chrono"); /// OK
4: long double a = 0 ;
5: for( long i = 0 ; i < 1000000 ; ++i )
6: {
7: Chrono chrono( svc , "Bad Chrono"); /// not OK
8: /// some codes :
9: a += sin( cos( sin( cos( (long double) i ))) );
10: /// end of codes
11: Stat stat ( svc , "Bad Stat", a ); /// not OK
12: }/// end of codes
13: Stat stat ( svc , "Good Stat", a); /// OK
14: } /// end of the scope!
15: /// ...
```
13.7 The Auditor Service

The Auditor Service provides a set of auditors that can be used to provide monitoring of various characteristics of the execution of Algorithms. Each auditor is called immediately before and after each call to each Algorithm instance, and can track some resource usage of the Algorithm. Calls that are thus monitored are `initialize()`, `execute()` and `finalize()`, although monitoring can be disabled for any of these for particular Algorithm instances. Only the `execute()` function monitoring is enabled by default.

Several examples of auditors are provided. These are:

- **NameAuditor.** This just emits the name of the Algorithm to the Standard Message Service immediately before and after each call. It therefore acts as a diagnostic tool to trace program execution.

- **ChronoAuditor.** This monitors the cpu usage of each algorithm and reports both the total and per event average at the end of job.

- **MemoryAuditor.** This monitors the state of memory usage during execution of each Algorithm, and will warn when memory is allocated within a call without being released on exit. Unfortunately this will in fact be the general case for Algorithms that are creating new data and registering them with the various transient stores. Such Algorithms will therefore cause warning messages to be emitted. However, for Algorithms that are just reading data from the transient stores, these warnings will provide an indication of a possible memory leak. Note that currently the MemoryAuditor is only available for Linux.

- **MemStatAuditor.** The same as MemoryAuditor, but prints a table of memory usage statistics at the end.

13.7.1 Enabling the Auditor Service and specifying the enabled Auditors

The Auditor Service is enabled by the following line in the Job Options file:

```plaintext
// Enable the Auditor Service
ApplicationMgr.DLLs += { "GaudiAud" ];
```

Specifying which auditors are enabled is illustrated by the following example:

```plaintext
// Enable the NameAuditor and ChronoAuditor
AuditorSvc.Auditors = { "NameAuditor", "ChronoAuditor" };```
13.7.2 Overriding the default Algorithm monitoring

By default, only monitoring of the Algorithm execute() function is enabled by default. This default can be overridden for individual Algorithms by use of the following Algorithm properties:

```cpp
myAlgorithm.AuditInitialize = true;
myAlgorithm.AuditExecute = false;
myAlgorithm.AuditFinalize = true;
```

13.7.3 Implementing new Auditors

The relevant portion of the IAuditor abstract interface is shown below:

```cpp
virtual StatusCode beforeInitialize( IAlgorithm* theAlg ) = 0;
virtual StatusCode afterInitialize( IAlgorithm* theAlg ) = 0;

virtual StatusCode beforeExecute( IAlgorithm* theAlg ) = 0;
virtual StatusCode afterExecute( IAlgorithm* theAlg ) = 0;

virtual StatusCode beforeFinalize( IAlgorithm* theAlg ) = 0;
virtual StatusCode afterFinalize( IAlgorithm* theAlg ) = 0;
```

A new Auditor should inherit from the Auditor base class and override the appropriate functions from the IAuditor abstract interface. The following code fragment is taken from the ChronoAuditor:

```cpp
#include "GaudiKernel/Auditor.h"

class ChronoAuditor : virtual public Auditor {
public:
    ChronoAuditor(const std::string& name, ISvcLocator* pSvcLocator);
    virtual ~ChronoAuditor();
    virtual StatusCode beforeInitialize( IAlgorithm* alg );
    virtual StatusCode afterInitialize( IAlgorithm* alg );
    virtual StatusCode beforeExecute( IAlgorithm* alg );
    virtual StatusCode afterExecute( IAlgorithm* alg );
    virtual StatusCode beforeFinalize( IAlgorithm* alg );
    virtual StatusCode afterFinalize( IAlgorithm* alg );
};
```
13.8 The Random Numbers Service

When generating random numbers two issues must be considered:

- reproducibility and
- randomness of the generated numbers.

In order to ensure both, Athena implements a single service ensuring that these criteria are met. The encapsulation of the actual random generator into a service has several advantages:

- Random seeds are set by the framework. When debugging the detector simulation, the program could start at any event independent of the events simulated before. Unlike the random number generators that were known from CERNLIB, the state of modern generators is no longer defined by one or two numbers, but rather by a fairly large set of numbers. To ensure reproducibility the random number generator must be initialized for every event.
- The distribution of the random numbers generated is independent of the random number engine behind. Any distribution can be generated starting from a flat distribution.
- The actual number generator can easily be replaced if at some time in the future better generators become available, without affecting any user code.

The implementation of both generators and random number engines are taken from CLHEP. The default random number engine used by Athena is the RanLux engine of CLHEP with a luxury level of 3, which is also the default for Geant4, so as to use the same mechanism to generate random numbers as the detector simulation.

Figure 13.1 shows the general architecture of the Athena random number service. The client interacts with the service in the following way:

- The client requests a generator from the service, which is able to produce a generator according to a requested distribution. The client then retrieves the requested generator.
- Behind the scenes, the generator service creates the requested generator and initializes the object according to the parameters. The service also supplies the shared random number engine to the generator.
- After the client has finished using the generator, the object must be released in order to inhibit resource leaks.

![Figure 13.1 The architecture of the random number service. The client requests from the service a random number generator satisfying certain criteria.](image_url)
There are many different distributions available. The shape of the distribution must be supplied as a parameter when the generator is requested by the user.

Currently implemented distributions include the following. See also the header file GaudiKernel/RndmGenerators.h for a description of the parameters to be supplied.

- Generate random bit patterns with parameters \texttt{Rndm::Bit()}
- Generate a flat distribution with boundaries \([\text{min}, \text{max}]\) with parameters:\n  \texttt{Rndm::Flat(double min, double max)}
- Generate a gaussian distribution with parameters: \texttt{Rndm::Gauss(double mean, double sigma)}
- Generate a poissonian distribution with parameters: \texttt{Rndm::Poisson(double mean)}
- Generate a binomial distribution according to \(n\) tests with a probability \(p\) with parameters: \texttt{Rndm::Binomial(long n, double p)}
- Generate an exponential distribution with parameters: \texttt{Rndm::Exponential(double mean)}
- Generate a \(\text{Chi}^2\) distribution with \(n_{\text{dof}}\) degrees of freedom with parameters: \texttt{Rndm::Chi2(long n_{\text{dof}})}
- Generate a Breit-Wigner distribution with parameters: \texttt{Rndm::BreitWigner(double mean, double gamma)}
- Generate a Breit-Wigner distribution with a cut-off with parameters: \texttt{Rndm::BreitWignerCutOff (mean, gamma, cut-off)}
- Generate a Landau distribution with parameters: \texttt{Rndm::Landau(double mean, double sigma)}
- Generate a user defined distribution. The probability density function is given by a set of discrete points passed as a vector of doubles: \texttt{Rndm::DefinedPdf(const std::vector<double>& pdf, long intpol)}

Clearly the supplied list of possible parameters is not exhaustive, but probably represents most needs. The list only represents the present content of generators available in CLHEP and can be updated in case other distributions will be implemented.

Since there is a danger that the interfaces are not released, a wrapper is provided that automatically releases all resources once the object goes out of scope. This wrapper allows the use of the random number service in a simple way. Typically there are two different usages of this wrapper:
Within the user code a series of numbers is required only once, i.e. not every event. In this case the object is used locally and resources are released immediately after use. This example is shown in Listing 13.5.

**Listing 13.5** Example of the use of the random number generator to fill a histogram with a Gaussian distribution within a standard Athena algorithm

```cpp
1: Rndm::Numbers gauss(randSvc(), Rndm::Gauss(0.5, 0.2));
2: if (gauss) {
3: IHistogram1D* his = histoSvc()->book("/stat/2","Gaussian",40,0.,3.);
4: for (long i = 0; i < 5000; i++)
5: his->fill(gauss(), 1.0);
6: }
```

One or several random numbers are required for the processing of every event. An example is shown in Listing 13.6.

**Listing 13.6** Example of the use of the random number generator within a standard Athena algorithm, for use at every event. The wrapper to the generator is part of the Algorithm itself and must be initialized before being used. Afterwards the usage is identical to the example described in Listing 13.5

```cpp
1: #include "GaudiKernel/RndmGenerators.h"
2:
3: // Constructor
4: class myAlgorithm : public Algorithm {
5: Rndm::Numbers m_gaussDist;
6: ...}
7: }
8:
9: // Initialisation
10: StatusCode myAlgorithm::initialize() {
11: ...
1: StatusCode sc=m_gaussDist.initialize(randSvc(), Rndm::Gauss(0.5, 0.2));
2: if (!status.isSuccess()) {
3: // put error handling code here...
4: }
5: ...}
6: }
```

There are a few points to be mentioned in order to ensure the reproducibility:

- Do not keep numbers across events. If you need a random number ask for it. Usually caching does more harm than good. If there is a performance penalty, it is better to find a more generic solution.
- Do not access the RndmEngine directly.
- Do not manipulate the engine. The random seeds should only be set by the framework on an event by event basis.
13.9 The Incident Service

The Incident service provides synchronization facilities to components in an Athena application. Incidents are named software events that are generated by software components and that are delivered to other components that have requested to be informed when that incident happens. The Athena components that want to use this service need to implement the IIncidentListener interface, which has only one method: handle(Incident&), and they need to add themselves as Listeners to the IncidentSvc. The following code fragment works inside Algorithms.

```cpp
class MyAlgorithm : public Algorithm, virtual public IIncidentListener {
  ...
};

MyAlgorithm::Initialize() {
  IIncidentSvc* incsvc;
  StatusCode sc = service("IncidentSvc", incsvc);
  int priority = 100;
  if( sc.isSuccess() ) {
    incsvc->addListener( this, "BeginEvent", priority);
    incsvc->addListener( this, "EndEvent");
  }
}

MyAlgorithm::handle(Incident& inc) {
  log << "Got informed of incident: " << inc.type()
    << " generated by: " << inc.source() << endreq;
}
```

The third argument in method addListener() is for specifying the priority by which the component will be informed of the incident in case several components are listeners of the same named incident. This parameter is used by the IncidentSvc to sort the listeners in order of priority.

13.9.1 Known Incidents

<table>
<thead>
<tr>
<th>Incident Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeginEvent</td>
<td>ApplicationMgr</td>
<td>The ApplicationMgr is starting processing of a new physics event. This incident can be use to clear caches of the previous event in Services and Tools.</td>
</tr>
<tr>
<td>EndEvent</td>
<td>ApplicationMgr</td>
<td>The ApplicationMgr has finished processing the physics event. The Event data store is not yet purged at this moment.</td>
</tr>
</tbody>
</table>
13.10 Developing new services

13.10.1 The Service base class

Within *Athena* we use the term "Service" to refer to a class whose job is to provide a set of facilities or utilities to be used by other components. In fact we mean more than this because a concrete service must derive from the `Service` base class and thus has a certain amount of predefined behaviour; for example it has `initialize()` and `finalize()` methods which are invoked by the application manager at well defined times.

Figure 13.1 shows the inheritance structure for an example service called `SpecificService`. The key idea is that a service should derive from the `Service` base class and additionally implement one or more pure abstract classes (interfaces) such as `IConcreteSvcType1` and `IConcreteSvcType2` in the figure.

![Diagram of Service Inheritance](image)

**Figure 13.1** Implementation of a concrete service class. Though not shown in the figure, both of the `IConcreteSvcType` interfaces are derived from `IInterface`.

As discussed above, it is necessary to derive from the Service base class so that the concrete service may be made accessible to other *Athena* components. The actual facilities provided by the service are available via the interfaces that it provides. For example the `ParticleProperties` service implements an interface which provides methods for retrieving, for example, the mass of a given particle. In figure 13.1 the service implements two interfaces each of two methods.
A component which wishes to make use of a service makes a request to the application manager. Services are requested by a combination of name, and interface type, i.e. an algorithm would request specifically either IConcreteSvcType1 or IConcreteSvcType2.

The identification of what interface types are implemented by a particular class is done via the queryInterface method of the IInterface interface. This method must be implemented in the concrete service class. In addition the initialize() and finalize() methods should be implemented. After initialization the service should be in a state where it may be used by other components.

The service base class offers a number of facilities itself which may be used by derived concrete service classes:

- Properties are provided for services just as for algorithms. Thus concrete services may be fine tuned by setting options in the job options file.
- A serviceLocator method is provided which allows a component to request the use of other services which it may need.
- A message service.

### 13.10.2 Implementation details

The following is essentially a checklist of the minimal code required for a service.

1. Define the interfaces
2. Derive the concrete service class from the Service base class.
3. Implement the queryInterface() method.
4. Implement the initialize() method. Within this method you should make a call to Service::initialize() as the first statement in the method and also make an explicit call to setProperties() in order to read the service’s properties from the job options (note that this is different from Algorithms, where the call to setProperties() is done in the base class).

#### Listing 13.7 An interface class

```cpp
#include "GaudiKernel/IInterface.h"

class IConcreteSvcType1 : virtual public IInterface {
public:
   void method1() = 0;
   int method2() = 0;
};
```
Listing 13.7 An interface class

```cpp
#include "IConcreteSvcType1.h"

const IID& IID_IConcreteSvcType1 = 143; // UNIQUE within LHCb !
```

Listing 13.8 A minimal service implementation

```cpp
#include "GaudiKernel/Service.h"
#include "IConcreteSvcType1.h"
#include "IConcreteSvcType2.h"

class SpecificService : public Service,
    virtual public IConcreteSvcType1,
    virtual public IConcreteSvcType2 {
public:
    // Constructor of this form required:
    SpecificService(const std::string& name, ISvcLocator* sl);  
    queryInterface(const IID& riid, void** ppvIF);
};
```
Listing 13.8 A minimal service implementation

```
// Factory for instantiation of service objects
static SvcFactory<SpecificService> s_factory;
const ISvcFactory& SpecificServiceFactory = s_factory;

// UNIQUE Interface identifiers defined elsewhere
extern const IID& IID_IConcreteSvcType1;
extern const IID& IID_IConcreteSvcType2;

// queryInterface
StatusCode SpecificService::queryInterface(const IID& riid, void** ppvIF) {
   if(IID_IConcreteSvcType1 == riid) {
      *ppvIF = dynamic_cast<IConcreteSvcType1*>(this);
      return StatusCode::SUCCESS;
   } else if(IID_IConcreteSvcType2 == riid) {
      *ppvIF = dynamic_cast<IConcreteSvcType2*>(this);
      return StatusCode::SUCCESS;
   } else {
      return Service::queryInterface(riid, ppvIF);
   }
}

StatusCode SpecificService::initialize() { ... }
StatusCode SpecificService::finalize() { ... }

// Implement the specifics ...
SpecificService::method1() { ... }
SpecificService::method2() { ... }
SpecificService::method3() { ... }
SpecificService::method4() { ... }
```
Chapter 14

Tools and ToolSvc

14.1 Overview

Tools are light weight objects whose purpose is to help other components perform their work. A framework service, the ToolSvc, is responsible for creating and managing Tools. An Algorithm requests the tools it needs to the ToolSvc, specifying if requesting a private instance by declaring itself as the parent. Since Tools are managed by the ToolSvc, any component1 can request a tool. Only Algorithms and Services can declare themselves as Tools parents.

In this chapter we first describe these objects and the difference between “private” and “shared” tools. We then look at the AlgTool base class and show how to write concrete Tools.

In section 14.3 we describe the ToolSvc and show how a component can retrieve Tools via the service.

Finally we describe Associators, common utility GaudiTools for which we provide the interface and base class.

14.2 Tools and Services

As mentioned elsewhere Algorithms make use of framework services to perform their work. In general the same instance of a service is used by many algorithms and Services are setup and initialized once at the beginning of the job by the framework. Algorithms also delegate some of their work to sub-algorithms. Creation and execution of sub-algorithms are the responsibilities of the parent

1. In this chapter we will use an Algorithm as example component requesting tools.
algorithm whereas the initialize() and finalize() methods are invoked automatically by the framework while initializing the parent algorithm. The properties of a sub-algorithm are automatically set by the framework but the parent algorithm can change them during execution. Sharing of data between nested algorithms is done via the Transient Event Store.

Both Services and Algorithms are created during the initialization stage of a job and live until the jobs ends.

Sometimes an encapsulated piece of code needs to be executed only for specific events, in which case it is desirable to create it only when necessary. On other occasions the same piece of code needs to be executed many times per event. Moreover it can be necessary to execute a sub-algorithm on specific contained objects that are selected by the parent algorithm or have the sub-algorithm produce new contained objects that may or may not be put in the Transient Store. Finally different algorithms may wish to configure the same piece of code slightly differently or share it as-is with other algorithms.

To provide this kind of functionality we have introduced a category of processing objects that encapsulate these “light” algorithms. We have called this category **Tools**.

Some examples of possible tools are single track fitters, association to Monte Carlo truth information, vertexing between particles, smearing of Monte Carlo quantities.

### 14.2.1 “Private” and “Shared” Tools

Algorithms can share instances of Tools with other Algorithms if the configuration of the tool is suitable. In some cases however an Algorithm will need to customize a tool in a specific way in order to use it. This is possible by requesting the ToolSvc to provide a “private” instance of a tool.

If an Algorithm passes a pointer to itself when it asks the ToolSvc to provide it with a tool, it is declaring itself as the parent and a “private” instance is supplied. Private instances can be configured according to the needs of each particular Algorithm via jobOptions.

As mentioned above many Algorithms can use a tool as-is, in which case only one instance of a Tool is created, configured and passed by the ToolSvc to the different algorithms. This is called a “shared” instance. The parent of “shared” tools is the ToolSvc.

### 14.2.2 The Tool classes

#### 14.2.2.1 The AlgTool base class

The main responsibilities of the AlgTool base class (see Listing 14.1) are the identification of the tools instances, the initialisation of certain internal pointers when the tool is created and the management of the tools properties. The AlgTool base class also offers some facilities to help in the implementation of derived tools.
Access to Services - A serviceLocator() method is provided to enable the derived tools to locate the services necessary to perform their jobs. Since concrete Tools are instantiated by the ToolSvc upon request, all Services created by the framework prior to the creation of a tool are available. In addition access to the message service is provided via the msgSvc() method. Both pointers are retrieved from the parent of the tool.

Listing 14.1 The definition of the AlgTool Base class

```cpp
class AlgTool : public virtual IAlgTool,
    public virtual IProperty {

public:
    // Standard Constructor.
    AlgTool( const std::string& type, const std::string& name,
              const IInterface* parent);

    virtual const std::string& name() const;
    virtual const std::string& type() const;
    virtual const IInterface* parent() const;

    virtual StatusCode setProperty(const Property& p);
    virtual StatusCode getProperty(Property* p) const;

    virtual const Property& getProperty( const std::string& name) const;
    virtual const std::vector<Property*>& getProperties( ) const;

    ISvcLocator* serviceLocator();
    IMessageSvc* msgSvc();
    IMessageSvc* msgSvc() const;

    StatusCode setProperties();

    StatusCode declareProperty(const std::string& name, int& reference);
    StatusCode declareProperty(const std::string& name, double& reference);
    StatusCode declareProperty(const std::string& name, bool& reference);
    StatusCode declareProperty(const std::string& name,
                                 std::string& reference);
    StatusCode declareProperty(const std::string& name,
                                 std::vector<int>& reference);
    StatusCode declareProperty(const std::string& name,
                                 std::vector<double>& reference);
    StatusCode declareProperty(const std::string& name,
                                 std::vector<bool>& reference);
    StatusCode declareProperty(const std::string& name,
                                 std::vector<std::string>& reference);

protected:
    // Standard destructor.
    virtual ~AlgTool();
```
Declaring Properties - A set of methods for declaring properties similarly to Algorithms is provided. This allows tuning of data members used by the Tools via JobOptions files. The ToolSvc takes care of calling the `setProperties()` method of the AlgTool base class after having instantiated a tool. Properties need to be declared in the constructor of a Tool. The property `outputLevel` is declared in the base class and is identically set to that of the parent component. For details on Properties see section 13.3.1.

Constructor - The base class has a single constructor which takes three arguments. The first is the type (i.e. the class) of the Tool object being instantiated, the second is the full name of the object and the third is a pointer to the IInterface of the parent component. The name is used for the identification of the tool instance as described below. The parent interface is used by the tool to access for example the `outputLevel` of the parent.

IAlgTool Interface - It consists of three accessor methods for the identification and management of the tools: `type()`, `name()` and `parent()`. These methods are all implemented by the base class and should not be overridden.

14.2.2.2 Tools identification

A tool instance is identified by its full name. The name consist of the concatenation of the parent name, a dot, and a tool dependent part. The tool dependent part can be specified by the user, when not specified the tool type (i.e. the class) is automatically taken as the tool dependent part of the name.

Examples of tool names are `RecPrimaryVertex.VertexSmearer` (a private tool) and `ToolSvc.AddFourMom` (a shared tool). The full name of the tool has to be used in the jobOptions file to set its properties.

14.2.2.3 Concrete tools classes

Operational functionalities of tools must be provided in the derived tool classes. A concrete tool class must inherit directly or indirectly from the AlgTool base class to ensure that it has the predefined behaviour needed for management by the ToolSvc. The inheritance structure of derived tools is shown in Figure 14.1. ConcreteTool1 implements one additional abstract interface while
ConcreteTool3 and ConcreteTool4 derive from a base class SubTool that provides them with additional common functionality.

The idea is that concrete tools could implement additional interfaces, specific to the task a tool is designed to perform. Specialised tools intended to perform similar tasks can be derived from a common base class that will provide the common functionality and implement the common interface. Consider as example the vertexing of particles, where separate tools can implement different algorithms but the arguments passed are the same. If a specialized tool is only accessed via the additional interface, the interface itself must inherit from the IAlgTool interface in order for the tool to be correctly managed by the ToolSvc.

**14.2.2.4 Implementation of concrete tools**

An example minimal implementation of a concrete tool is shown in Listing 14.2 and Listing 14.3, taken from the LHCb ToolsAnalysis example application.

**Listing 14.2 Example of a concrete tool minimal implementation header file**

```cpp
1: #include "GaudiKernel/AlgTool.h"
2: class VertexSmearer : public AlgTool {
3:     public:
4:     // Constructor
5:     VertexSmearer( const std::string& type, const std::string& name,
6:                      const IInterface* parent);
7:     // Standard Destructor
8:     virtual ~VertexSmearer() {}
9:     // specific method of this tool
10:    StatusCode smear( MyAxVertex* pvertex );
```
The creation of concrete tools is similar to that of Algorithms, making use of a Factory Method. As for Algorithms, Tool factories enable their creator to instantiate new tools without having to include any of the concrete tools header files. A template factory is provided and a tool developer will only need to add the concrete factory in the implementation file as shown in lines 1 to 4 of Listing 14.3.

In addition a concrete tool class must specify a single constructor with the same parameter signatures as the constructor of the \texttt{AlgTool} base class as shown in line 5 of Listing 14.2.

Below is the minimal checklist of the code necessary when developing a Tool:

1. Derive the tool class from the \texttt{AlgTool} base class
2. Provide the constructor
3. Implement the factory adding the lines of code shown in Listing 14.3

In addition if the tool is implementing an additional interface you may need to:

1. Define the specific interface (inheriting from the \texttt{IAlgTool} interface).
2. Implement the \texttt{queryInterface()} method.
3. Implement the specific interface methods.
14.3 The ToolSvc

The ToolSvc manages Tools. It is its responsibility to create tools and make them available to Algorithms or Services.

The ToolSvc verifies if a tool type is available and creates the necessary instance after having verified if it doesn’t already exist. If a tool instance exists the ToolSvc will not create a new identical one but pass to the algorithm the existing instance. Tools are created on a “first request” basis: the first Algorithm requesting a tool will prompt its creation. The relationship between an algorithm, the ToolSvc and Tools is shown in Figure 14.1.

The ToolSvc will “hold” a tool until it is no longer used by any component or until the finalize() method of the tool service is called. Algorithms can inform the ToolSvc they are not going to use a tool previously requested via the releaseTool method of the IToolSvc interface\(^1\).

The ToolSvc is created by default by the ApplicationMgr and algorithms wishing to use the service can retrieve it using the service accessor method of the Algorithm base class as shown in the lines below.

```cpp
#include "GaudiKernel/IToolSvc.h"
...
IToolSvc* toolSvc=0;
StatusCode sc = service( "ToolSvc", toolSvc);
if ( sc.isFailure) {
  ...
```

\(^1\) The releaseTool method is not available in this release
14.3.1 Retrieval of tools via the IToolSvc interface

The IToolSvc interface is the ToolSvc specific interface providing methods to retrieve tools. The interface has two retrieve methods that differ in their parameters signature, as shown in Listing 14.4

**Listing 14.4** The IToolSvc interface methods

```cpp
1: virtual StatusCode retrieve(const std::string& type, 
   IAlgTool*& tool, 
   const IInterface* parent=0, 
   bool createIf=true ) = 0; 
2: virtual StatusCode retrieve(const std::string& type, 
   const std::string& name, 
   IAlgTool*& tool, 
   const IInterface* parent=0, 
   bool createIf=true ) = 0; 
```

The arguments of the method shown in Listing 14.4, line 1, are the tool type (i.e. the class) and the IAlgTool interface of the returned tool. In addition there are two arguments with default values: one is the IInterface of the component requesting the tool, the other a boolean creation flag. If the component requesting a tool passes a pointer to itself as the third argument, it declares to the ToolSvc that it is asking for a “private” instance of the tool. By default a “shared” instance is provided. In general if the requested instance of a Tool does not exist the ToolSvc will create it. This behaviour can be changed by setting to false the last argument of the method.

The method shown in Listing 14.4, line 2 differs from the one shown in line 1 by an extra argument, a string specifying the tool dependent part of the full tool name. This enables a component to request two separately configurable instances of the same tool.

To help the retrieval of concrete tools two template functions, as shown in Listing 14.5, are provided in the IToolSvc interface file.

**Listing 14.5** The IToolSvc template methods

```cpp
1: template <class T>
2: StatusCode retrieveTool( const std::string& type, 
   T*& tool, 
   const IInterface* parent=0, 
   bool createIf=true ) {...}
3: template <class T>
4: StatusCode retrieveTool( const std::string& type, 
   const std::string& name, 
   T*& tool, 
   const IInterface* parent=0, 
   bool createIf=true ) {...}
```

The two template methods correspond to the IToolSvc retrieve methods but have the tool returned as a template parameter. Using these methods the component retrieving a tool avoids explicit dynamic-casting to specific additional interfaces or to derived classes.
Listing 14.6 shows an example of retrieval of a shared and of a common tool.

**Listing 14.6** Example of retrieval of a shared tool in line 8: and of a private tool in line 14:

```cpp
1: IToolSvc* toolsvc = 0;
2: sc = service( "ToolSvc", toolsvc );
3: if( sc.isFailure() ) {
4:   log << MSG::FATAL << " Unable to locate Tool Service" << endreq;
5:   return sc;
6: }
7: // Example of tool belonging to the ToolSvc and shared between
8: sc = toolsvc->retrieveTool("AddFourMom", m_sum4p );
9: if( sc.isFailure() ) {
10:  log << MSG::FATAL << " Unable to create AddFourMom tool" << endreq;
11:  return sc;
12: }
13: // Example of private tool
14: sc = toolsvc->retrieveTool("ImpactPar", m_ip, this );
15: if( sc.isFailure() ) {
16:  log << MSG::FATAL << " Unable to create ImpactPar tool" << endreq;
17:  return sc;
18: }
```

### 14.4 GaudiTools

In general concrete tools are specific to applications or detectors’ code but there are some tools of common utility for which interfaces and base classes can be provided. The **Associators** described below and contained in the **GaudiTools** package are one of such tools.

#### 14.4.1 Associators

When working with Monte Carlo data it is often necessary to compare the results of reconstruction or physics analysis with the original corresponding Monte Carlo quantities on an event-by-event basis as well as on a statistical level.

Various approaches are possible to implement navigation from reconstructed simulated data back to the Monte Carlo truth information. Each of the approaches has its advantages and could be more suited for a given type of event data or data-sets. In addition the reconstruction and physics analysis code should treat simulated data in an identical way to real data.

In order to shield the code from the details of the navigation procedure, and to provide a uniform interface to the user code, a set of Gaudi Tools, called **Associators**, has been introduced. The user can navigate between any two arbitrary classes in the Event Model using the same interface as long as a corresponding associator has been implemented. Since an Associator retrieves existing navigational information, its actual implementation depends on the Event Model and how the navigational
information is stored. For some specific Associators, in addition, it can depend on some algorithmic choices: consider as an example a physics analysis particle and a possible originating Monte Carlo particle where the associating discriminant could be the fractional number of hits used in the reconstruction of the tracks. An advantage of this approach is that the implementation of the navigation can be modified without affecting the reconstruction and analysis algorithms because it would affect only the associators. In addition short-cuts or complete navigational information can be provided to the user in a transparent way. By limiting the use of such associators to dedicated monitoring algorithms where the comparison between raw/reconstructed data and MC truth is done, one could ensure that the reconstruction and analysis code treat simulated and real data in an identical way.

Associators must implement a common interface called IAssociator. An Associator base class providing at the same time common functionality and some facilities to help in the implementation of concrete Associators is provided. A first version of these classes is provided in the current release of Athena.

### 14.4.1.1 The IAssociator Interface

As already mentioned Associators must implement the IAssociator interface.

In order for Associators to be retrieved from the ToolSvc only via the IAssociator interface, the interface itself inherits from the IAlgTool interface. While the implementation of the IAlgTool interface is done in the AlgTool base class, the implementation of the IAssociator interface is the full responsibility of concrete associators.

The four methods of the IAssociator interface that a concrete Associator must implement are show in Listing 14.7

**Listing 14.7 Methods of the IAssociator Interface that must be implemented by concrete associators**

```c++
1: virtual StatusCode i_retrieveDirect( ContainedObject* objFrom,
           ContainedObject*& objTo,
           const CLID idFrom,
           const CLID idTo ) = 0;

2: virtual StatusCode i_retrieveDirect( ContainedObject* objFrom,
           std::vector<ContainedObject*>& vObjTo,
           const CLID idFrom,
           const CLID idTo ) = 0;

3: virtual StatusCode i_retrieveInverse( ContainedObject* objFrom,
           ContainedObject*& objTo,
           const CLID idFrom,
           const CLID idTo ) = 0;

4: virtual StatusCode i_retrieveInverse( ContainedObject* objFrom,
           std::vector<ContainedObject*>& vObjTo,
           const CLID idFrom,
           const CLID idTo ) = 0;
```

Two `i_retrieveDirect` methods must be implemented for retrieving associated classes following the same direction as the links in the data: for example from reconstructed particles to Monte Carlo particles. The first parameter is a pointer to the object for which the associated Monte Carlo
quantity(ies) is requested. The second parameter, the discriminating signature between the two methods, is one or a vector of pointers to the associated Monte Carlo objects of the type requested. Some reconstructed quantities will have only one possible Monte Carlo associated object of a certain type, some will have many, others will have many out of which a “best” associated object can be extracted. If one of the two methods is not valid for a concrete associator, such method must return a failure. The third and fourth parameters are the class IDs of the objects for which the association is requested. This allows to verify at run time if the objects’ types are those the concrete associator has been implemented for.

The two i_retrieveInverse methods are complementary and are for retrieving the association between the same two classes but in the opposite direction to that of the links in the data: from Monte Carlo particles to reconstructed particles. The different name is intended to alert the user that navigation in this direction may be a costly operation.

Four corresponding template methods are implemented in IAssociator to facilitate the use of Associators by Algorithms (see Listing 14.8). Using these methods the component retrieving a tool avoids some explicit dynamic-casting as well as the setting of class IDs. An example of how to use such methods is described in section 14.4.1.3.

**Listing 14.8** Template methods of the IAssociator interface

```
1: template <class T1, class T2>
   StatusCode retrieveDirect( T1* from, T2*& to ) {...}
2: template <class T1>
   StatusCode retrieveDirect( T1* from,
               std::vector<ContainedObject*>& objVTo,
               const CLID idTo ) {...}
3: template <class T1, class T2>
   StatusCode retrieveInverse( T1* from, T2*& to ) {...}
4: template <class T1>
   StatusCode retrieveInverse( T1* from,
               std::vector<ContainedObject*>& objVTo,
               const CLID idTo ) {...}
```

### 14.4.1.2 The Associator base class

An associator is a type of AlgTool, so the Associator base class inherits from the AlgTool base class. Thus, Associators can be created and managed as AlgTools by the ToolSvc. Since all the methods of the AlgTool base class (as described in section 14.2.2.1) are available in the Associator base class, only the additional functionality is described here.

**Access to Event Data Service** - An eventSvc() method is provided to access the Event Data Service since most concrete associators will need to access data, in particular if accessing navigational short-cuts.

**Associator Properties** - Two properties are declared in the constructor and can be set in the jobOptions: “FollowLinks” and “DataLocation”. They are respectively a bool with initial value true and a std::string with initial value set to “”. The first is foreseen to be used by an associator when it is possible to either follow links between classes or retrieve navigational short cuts.
from the data. A user can choose to set either behaviour at run time. The second property contains the location in the data where the stored navigational information is located. Currently it must be set via the jobOptions when necessary, as shown in Listing 14.9 for a particular implementation provided in the Associator example. Two corresponding methods are provided for using the information from these properties: followLinks() and whichTable().

**Inverse Association** - Retrieving information in the direction opposite to that of the links in the data is in general a time consuming operation, that implies checking all the direct associations to access the inverse relation for a specified object. For this reason Associators should keep a local copy of the inverse associations after receiving the first request for an event. A few methods are provided to facilitate the work of Associators in this case. The methods inverseExist() and setInverseFlag(bool) help in keeping track of the status of the locally kept inverse information. The method buildInverse() has to be overridden by concrete associators since they choose in which form to keep the information and should be called by the associator when receiving the first request during the processing of an event.

**Locally kept information** - When a new event is processed, the associator needs to reset its status to the same conditions as those after having been created. In order to be notified of such an incident happening the Associator base class implements the IListener interface and, in the constructor, registers itself with the Incident Service (see section 13.9 for details of the Incident Service). The associator’s flushCache() method is called in the implementation of the IListener interface in the Associator base class. This method must be overridden by concrete associators wanting to do a meaningful reset of their initial status.

**14.4.1.3 A concrete example**

In this section we look at an example implementation of a specific associator. The code is taken from the LHCb Associator example, but the points illustrated should be clear even without a knowledge of the LHCb data model.

The AxPart2MCParticleAsct provides association between physics analysis particles (AxPartCandidate) and the corresponding Monte Carlo particles (MCParticle). The direct navigational information is stored in the persistent data as short-cuts, and is retrieved in the form of a SmartRefTable in the Transient Event Store. This choice is specific to AxPart2MCParticleAsct, any associator can use internally a different navigational mechanism. The location in the Event Store where the navigational information can be found is set in the job options via the “DataLocation” property, as shown in Listing 14.9.

**Listing 14.9** Example of setting properties for an associator via jobOptions

```cpp
ToolSvc.AxPart2MCParticleAsct.DataLocation = "/Event/Anal/AxPart2MCParticle";
```

In the current LHCb data model only a single MCParticle can be associated to one AxPartCandidate and vice-versa only one or no AxPartCandidate can be associated to one MCParticle. For this reason only the i_retrieveDirect and i_retrieveInverse methods providing one-to-one association are meaningful. Both methods verify that the objects passed are of the correct type before attempting to retrieve the information, as shown in Listing 14.10. When no association is found, a StatusCode::FAILURE is returned.
The `i_retrieveInverse` method providing the one-to-many association returns a failure, while a fake implementation of the one-to-many `i_retrieveDirect` method is implemented in the example, to show how an Algorithm can use such a method. In the `AxPart2MCParticleAsct` example the inverse table is kept locally and both the `buildInverse()` and `flushCache()` methods are overridden. In the example the choice has been made to implement an additional method `buildDirect()` to retrieve the direct navigational information on a first request per event basis.

Listing 14.11 shows how a monitoring Algorithm can get an associator from the `ToolSvc` and use it to retrieve associated objects through the template interfaces.

### Listing 14.10 Checking if objects to be associated are of the correct type

```cpp
1: if ( idFrom != AxPartCandidate::classID() ) {
2:   objTo = 0;
3:   return StatusCode::FAILURE;
4: }
5: if ( idTo != MCParticle::classID() ) {
6:   objTo = 0;
7:   return StatusCode::FAILURE;
8: }
```
Listing 14.11 Extracted code from the AsctExampleAlgorithm

```cpp
1: #include "GaudiTools/IAssociator.h"

2: // Example of retrieving an associator
3: IAssociator
4: StatusCode sc = toolsvc->retrieveTool("AxPart2MCParticleAsct", m_pAsct);
5: if( sc.isFailure() ) {
6:   log << MSG::FATAL << "Unable to create Associator tool" << endreq;
7:   return sc;
8: }

9: // Example of retrieving inverse one-to-one information from an
10: // associator
11: SmartDataPtr<MCParticleVector> vmcparts (evt,"/MC/MCParticles");
12: for( MCParticleVector::iterator itm = vmcparts->begin();
13:     vmcparts->end() != itm; itm++) {
14:   AxPartCandidate* mptry = 0;
15:   StatusCode sc = m_pAsct->retrieveInverse( *itm, mptry );
16:   if( sc.isSuccess() ) {...}
17:   else {...}
18: }

19: // Example of retrieving direct one-to-many information from an
20: // associator
21: SmartDataPtr<AxPartCandidateVector> candidates(evt,
22:     "/Anal/AxPartCandidates");
23: std::vector<ContainedObject*> pptry;
24: AxPartCandidate* itP = *(candidates->begin());
25: StatusCode sa =
26:     m_pAsct->retrieveDirect(itP, pptry, MCParticle::classID());
27: if( sa.isFailure() ) {...}
28: else {
29:   for (std::vector<ContainedObject*>::iterator it = pptry.begin();
30:        pptry.end() != it; it++ ) {
31:     MCParticle* imc = dynamic_cast<MCParticle*>( *it );
32:   }
33: }
```
Chapter 15
Converters

15.1 Overview

Consider a small piece of detector; a silicon wafer for example. This “object” will appear in many contexts: it may be drawn in an event display, it may be traversed by particles in a Geant4 simulation, its position and orientation may be stored in a database, the layout of its strips may be queried in an analysis program, etc. All of these uses or views of the silicon wafer will require code.

One of the key issues in the design of the framework was how to encompass the need for these different views within Athena. In this chapter we outline the design adopted for the framework and look at how the conversion process works. This is followed by sections which deal with the technicalities of writing converters for reading from and writing to ROOT files.

15.2 Persistency converters

Athena gives the possibility to read event data from, and to write data back to, ROOT files. The use of ODBC compliant databases is also possible, though this is not yet part of the Athena release. Other persistency technologies have been implemented for LHCb, in particular the reading of data from LHCb DSTs based on ZEBRA.

Figure 15.1 is a schematic illustrating how converters fit into the transient-persistent translation of event data. We will not discuss in detail how the transient data store (e.g. the event data service) or the persistency service work, but simply look at the flow of data in order to understand how converters are used.

One of the issues considered when designing the Athena framework was the capability for users to “create their own data types and save objects of those types along with references to already existing
objects”. A related issue was the possibility of having links between objects which reside in different stores (i.e. files and databases) and even between objects in different types of store.

Figure 15.1 shows that data may be read from an ODBC database and from ROOT files into the transient event data store and that data may be written out again to the same media. It is the job of the persistency service to orchestrate this transfer of data between memory and disk.

The figure shows two “slave” services: the ODBC conversion service and the ROOT I/O service. These services are responsible for managing the conversion of objects between their transient and persistent representations. Each one has a number of converter objects which are actually responsible for the conversion itself. As illustrated by the figure a particular converter object converts between the transient representation and one other form, here either MS Access or ROOT.

15.3 Collaborators in the conversion process

In general the conversion process occurs between the transient representation of an object and some other representation. In this chapter we will be using persistent forms, but it should be borne in mind that this could be any other “transient” form such as those required for visualisation or those which serve as input into other packages (e.g. Geant4).

Figure 15.1 shows the interfaces (classes whose name begins with "I") which must be implemented in order for the conversion process to function.

The conversion process is essentially a collaboration between the following types:

- IConversionSvc
For each persistent technology, or “non-transient” representation, a specific conversion service is required. This is illustrated in the figure by the class AConversionSvc which implements the IConversionSvc interface.
A given conversion service will have at its disposal a set of converters. These converters are both type and technology specific. In other words a converter knows how to convert a single transient type (e.g. MuonHit) into a single persistent type (e.g. RootMuonHit) and vice versa. Specific converters implement the IConverter interface, possibly by extending an existing converter base class.

A third collaborator in this process are the opaque address objects. A concrete opaque address class must implement the IOpaqueAddress interface. This interface allows the address to be passed around between the transient data service, the persistency service, and the conversion services without any of them being able to actually decode the address. Opaque address objects are also technology specific. The internals of an OdbcAddress object are different from those of a RootAddress object.

Only the converters themselves know how to decode an opaque address. In other words only converters are permitted to invoke those methods of an opaque address object which do not form a part of the IOpaqueAddress interface.

Converter objects must be “registered” with the conversion service in order to be usable. For the “standard” converters this will be done automatically. For user defined converters (for user defined types) this registration must be done at initialisation time (see Chapter 7).

### 15.4 The conversion process

As an example (see Figure 15.1) we consider a request from the event data service to the persistency service for an object to be loaded from a data file.

As we saw previously, the persistency service has one conversion service slave for each persistent technology in use. The persistency service receives the request in the form of an opaque address object. The svcType() method of the IOpaqueAddress interface is invoked to decide which conversion service the request should be passed onto. This returns a “technology identifier” which allows the persistency service to choose a conversion service.

The request to load an object (or objects) is then passed onto a specific conversion service. This service then invokes another method of the IOpaqueAddress interface, clID(), in order to decide which converter will actually perform the conversion. The opaque address is then passed onto the concrete converter who knows how to decode it and create the appropriate transient object.

The converter is specific to a specific type, thus it may immediately create an object of that type with the new operator. The converter must now “unpack” the opaque address, i.e. make use of accessor methods specific to the address type in order to get the necessary information from the persistent store.

For example, a ZEBRA converter might get the name of a bank from the address and use that to locate the required information in the ZEBRA common block. On the other hand a ROOT converter may extract a file name, the names of a ROOT TTree and an index from the address and use these to load an object from a ROOT file. The converter would then use the accessor methods of this “persistent” object in order to extract the information necessary to build the transient object.
We can see that the detailed steps performed within a converter depend very much on the nature of the non-transient data and (to a lesser extent) on the type of the object being built.

If all transient objects were independent, i.e. if there were no references between objects then the job would be finished. However in general objects in the transient store do contain references to other objects.

These references can be of two kinds:
i. “Macroscopic” references appear as separate “leaves” in the data store. They have to be registered with a separate opaque address structure in the data directory of the object being converted. This must be done after the object was registered in the data store in the method `fillObjRefs()`.

ii. Internal references must be handled differently. There are two possibilities for resolving internal references:

1. Load on demand: If the object the reference points to should only be loaded when accessed, the pointer must no longer be a raw C++ pointer, but rather a smart pointer object containing itself the information for later resolution of the reference. This is the preferred solution for references to objects within the same data store, e.g. references from the Monte-Carlo tracks to the Monte-Carlo vertices. Please see in the corresponding SICB converter implementations how to construct these smart pointer objects. Late loading is highly preferable compared to the second possibility.

2. Filling of raw C++ pointers: Here things are a little more complicated and introduces the need for a second step in the process. This is only necessary if the object points to an object in another store, e.g. the detector data store. To resolve the reference a converter has to retrieve the other object and set the raw pointer. These references should be set in the `fillObjRefs()` method. This of course is more complicated, because it must be ensured that both objects are present at the time the reference is accessed (i.e. when the pointer is actually used).

### 15.5 Converter implementation - general considerations

After covering the ground work in the preceding sections, let us look exactly what needs to be implemented in a specific converter class. The starting point is the `Converter` base class from which a user converter should be derived. For concreteness let us partially develop a converter for the `UDO` class of Chapter 7.

The converter shown in Listing 15.1 is responsible for the conversion of `UDO` type objects into objects that may be stored into an Objectivity database and vice-versa. The `UDOCnv` constructor calls the `Converter` base class constructor with two arguments which contain this information. These are the values `CLID_UDO`, defined in the `UDO` class, and `Objectivity_StorageType` which is also defined elsewhere. The first two `extern` statements simply state that these two identifiers are defined elsewhere.

All of the “book-keeping” can now be done by the `Converter` base class. It only remains to fill in the guts of the converter. If objects of type `UDO` have no links to other objects, then it suffices to implement the methods `createRep()` for conversion from the transient form (to Objectivity in this case) and `createObj()` for the conversion to the transient form.

If the object contains links to other objects then it is also necessary to implement the methods `fillRepRefs()` and `fillObjRefs()`.
15.6 Storing Data using the ROOT I/O Engine

One possibility for storing data is to use the ROOT I/O engine to write ROOT files. Although ROOT by itself is not an object oriented database, with modest effort a structure can be built on top to allow the Converters to emulate this behaviour. In particular, the issue of object linking had to be solved in order to resolve pointers in the transient world.

The concept of ROOT supporting paged tuples called trees and branches is adequate for storing bulk event data. Trees split into one or several branches containing individual leaves with data. The data structure within the Athena data store is tree like (as an example, part of the LHCb event data model is shown in Figure 15.1).

In the transient world Athena objects are sub-class instances of the “DataObject”. The DataObject offers some basic functionality like the implicit data directory which allows e.g. to browse a data store. This tree structure will be mapped to a flat structure in the ROOT file resulting in a separate tree representing each leaf of the data store. Each data tree contains a single branch containing objects of the same type. The Athena tree is split up into individual ROOT trees in order to give easy access to individual items represented in the transient model without the need of loading complete events from the root file i.e. to allow for selective data retrieval. The feature of ROOT supporting selective data reading using split trees did not seem too attractive since, generally, complete nodes in the transient store should be made available in one go.

However, ROOT expects “ROOT” objects, they must inherit from TObject. Therefore the objects from the transient store have to be converted to objects understandable by ROOT.

Listing 15.1 An example converter class

```c++
// Converter for class UDO.
extern const CLID& CLID_UDO;
extern unsigned char OBJY_StorageType;

static CnvFactory<UDOCnv> s_factory;
const ICnvFactory& UDOCnvFactory = s_factory;

class UDOCnv : public Converter {
public:
    UDOCnv(ISvcLocator* svcLoc) :
        Converter(Objectivity_StorageType, CLID_UDO, svcLoc) { }

    createRep(DataObject* pO, IOpaqueAddress*& a); // transient->persistent
    createObj(IOpaqueAddress* pa, DataObject*& pO); // persistent->transient

    fillObjRefs( ... ); // transient->persistent
    fillRepRefs( ... ); // persistent->transient
}
```
The following sections are an introduction to the machinery provided by the Athena framework to achieve the migration of transient objects to persistent objects. The ROOT specific aspects are not discussed here; the documentation of the ROOT I/O engine can be found at the ROOT web site http://root.cern.ch). Note that Athena only uses the I/O engine, not all ROOT classes are available.

Within Athena the ROOT I/O engine is implemented in the GaudiRootDb package.

### 15.7 The Conversion from Transient Objects to ROOT Objects

As for any conversion of data from one representation to another within the Athena framework, conversion to/from ROOT objects is based on Converters. The support of a “generic” Converter accesses pre-defined entry points in each object. The transient object converts itself to an abstract byte stream.

However, for specialized objects specific converters can be built by virtual overrides of the base class.

Whenever objects must change their representation within Athena, data converters are involved. For the ROOT case the converters must have some knowledge of ROOT internals and the service finally used to migrate ROOT objects (->TObject) to a file. In the same way the converter must be able to translate the functionality of the DataObject component to/from the Root storage. Within ROOT itself the object is stored as a Binary Large Object (BLOB).
The instantiation of the appropriate converter is done by a macro. The macro instantiates also the converter factory used to instantiate the requested converter. Hence, all other user code is shielded from the implementation and definitions of the ROOT specific code.

**Listing 15.2** Implementing a “generic” converter for the transient class *Event*.

```cpp
1: // Include files
2: #include "GaudiKernel/ObjectVector.h"
3: #include "GaudiKernel/ObjectList.h"
4: #include "GaudiDb/DbGenericConverter.h"
5: // Converter implementation for objects of class Event
6: #include "Event.h"
7: _ImplementConverter(Event)
```

The macro needs a few words of explanation: the instantiated converters are able to create transient objects of type *Event*. The corresponding persistent type is of a generic type, the data are stored as a machine independent byte stream. It is mandatory that the Event class implements a streamer method “serialize”. An example from the Event class of the *RootIO* example is shown in Listing 15.3.

The instantiated converter is of the type *DbGenericConverter* and the instance of the instantiating factory has the instance name *DbEventCnvFactory*.

**Listing 15.3** Serialisation of the class *Event*.

```cpp
1: /// Serialize the object for writing
2: virtual StreamBuffer& serialize( StreamBuffer& s ) const { 
3:     DataObject::serialize(s);
4:     return s
5:         << m_event
6:         << m_run
7:         << m_time;
8: }
9: /// Serialize the object for reading
10: virtual StreamBuffer& serialize( StreamBuffer& s ) { 
11:     DataObject::serialize(s);
12:     return s
13:         >> m_event
14:         >> m_run
15:         >> m_time;
16: }
```

### 15.7.1 Non Identifiable Objects

Non identifiable objects cannot directly be retrieved/stored from the data store. Usually they are small and in any case they are contained by a container object. Examples are particles, hits or vertices. These classes can be converted using a generic container converter. Container converters exist currently for lists and vectors. The containers rely on the serialize methods of the contained objects. The serialisation is able to understand smart references to other objects within the same data store. Listing 15.4 shows an example of the serialize methods of the *MyTrack* class of the *RootIO* example.
Please refer to the RootIO Gaudi example for further details how to store objects in ROOT files.

### 15.8 Storing Data using other I/O Engines

Once objects are stored as BLOBs, it is possible to adopt any storage technology supporting this datatype. This is the case not only for ROOT, but also for

- Objectivity/DB
- most relational databases, which support an ODBC interface like
  - Microsoft Access,
  - Microsoft SQL Server,
  - MySQL,
  - ORACLE and others.

Note that although storing objects using these technologies is possible, there is currently no implementation available in the Athena release. If you desperately want to use Objectivity or one of the ODBC databases, please contact Markus Frank (Markus.Frank@cern.ch).
Chapter 16

Visualization

16.1 Overview

This chapter is a place holder for documenting the experiment specific visualization information.
Chapter 17
Physical design issues

17.1 Overview

This chapter discusses several physical design issues. These include how to access the kernel GAUDI framework from within the ATLAS SRT software environment, and how to deal with the different types of libraries that are relevant.

17.2 Accessing the kernel GAUDI framework

Athena is based upon the kernel GAUDI framework, and therefore ATLAS packages that create components such as Algorithms or Services need access to components, header files and helper classes contained within that framework. The GAUDI framework is treated as an external package by the ATLAS computing environment. The required access is provided by the GaudiInterface ATLAS package.

17.2.1 The GaudiInterface Package

ATLAS packages that create Algorithms or Services should include the line shown in Listing 17.1 in their PACKAGE file:

Listing 17.1 Entry in PACKAGE file for a SRT package

```
use GaudiInterface 0 External
```
Notes:


While this should generally be sufficient for most packages, the GaudiInterface package also defines several linksets that can be used to access other aspects of the GAUDI installation itself. These linksets are the following:

- GaudiInterface[DbCnv] This linkset is obsolete, being replaced by the GaudiDb linkset. It is retained for backwards compatibility and will be removed in a future release.

- GaudiInterface[GaudiAlg] This allows developers to inherit from the Sequencer class and override its default behaviour.

- GaudiInterface[GaudiDb] This allows access to the StreamBuffer I/O support for DataObjects and ContainedObjects. It is a replacement for the DbCnv linkset which is deprecated.

These auxiliary linksets should be used in conjunction with the primary one as illustrated in Listing 17.2.

Listing 17.2 Linkset entry in PACKAGE file for a Package that depends upon GAUDI

| use GaudiInterface 0 External |
| use GaudiInterface[GaudiDb] 0 External |

17.3 Framework libraries

Three different sorts of library can be identified that are relevant to the framework. These are component libraries, linker libraries, and dual-purpose libraries. These libraries are used for different purposes and are built in different ways.

17.3.1 Component libraries

Component libraries are shared libraries that contain standard framework components which implement abstract interfaces. Such components are Algorithms, Auditors, Services, Tools or Converters. Component libraries are treated in a special manner by Athena, and should not be linked against. They do not export their symbols except for a special one which is used by the framework to discover what components are contained by the library. Thus component libraries are used purely at run-time, being loaded dynamically upon request, the configuration being specified by the job options file. Changes in the implementation of a component library do not require the application to be relinked.

Component libraries contain factories for their components, and it is important that the factory entries are declared and loaded correctly. The following sections describe how this is done.
When a component library is loaded, the framework attempts to locate a single entrypoint, called `getFactoryEntries()`. This is expected to declare and load the component factories from the library. Several macros are available to simplify the declaration and loading of the components via this mechanism.

Consider a simple package `MyComponents`, that declares and defines the `MyAlgorithm` class, being a subclass of `Algorithm`, and the `MyService` class, being a subclass of `Service`. Thus the package will contain the header and implementation files for these classes (`MyAlgorithm.h`, `MyAlgorithm.cxx`, `MyService.h` and `MyService.cxx`) in addition to whatever other files are necessary for the correct functioning of these components.

In order to satisfy the requirements of a component library, two additional files must also be present in the package. One is used to declare the components, the other to load them. Because of the technical limitations inherent in the use of shared libraries, it is important that these two files remain separate, and that no attempt is made to combine their contents into a single file.

The names of these files and their contents are described in the following sections.

### 17.3.1.1 Declaring Components

Components within the component library are declared in a file `MyComponents_entries.cxx`. By convention, the name of this file is the package name concatenated with `_entries`. The contents of this file are shown in Listing 17.3:

**Listing 17.3** The `MyComponents_entries.cxx` file

```cpp
#include "Gaudi/Kernel/DeclareFactoryEntries.h"

DECLARE_FACTORY_ENTRIES( MyComponents ) {  
  DECLARE_ALGORITHM( MyAlgorithm );  
  DECLARE_SERVICE ( MyService );
}
```

Notes:

1. The argument to the `DECLARE_FACTORY_ENTRIES` statement is the name of the component library.
2. Each component within the library should be declared using one of the `DECLARE_XXX` statements discussed in detail in the next Section.
17.3.1.1 Component declaration statements

The complete set of statements that are available for declaring components is shown in Listing 17.4. They include those that support C++ classes in different namespaces, as well as for DataObjects or ContainedObjects using the generic converters.

**Listing 17.4** The available component declaration statements

```
DECLARE_ALGORITHM(X)
DECLARE_ALGTOOL(X)
DECLARE_AUDITOR(X)
DECLARE_CONVERTER(X)
DECLARE_GENERIC_CONVERTER(X) [1]
DECLARE_OBJECT(X)
DECLARE_SERVICE(X)
DECLARE_TOOL(X) [2]
DECLARE_NAMESPACE_ALGORITHM(N,X) [3]
DECLARE_NAMESPACE_ALGTOOL(N,X)
DECLARE_NAMESPACE_AUDITOR(N,X)
DECLARE_NAMESPACE_CONVERTER(N,X)
DECLARE_NAMESPACE_GENERIC_CONVERTER(N,X)
DECLARE_NAMESPACE_OBJECT(N,X)
DECLARE_NAMESPACE_SERVICE(N,X)
DECLARE_NAMESPACE_TOOL(N,X)
```

Notes:

1. Declarations of the form `DECLARE_GENERIC_CONVERTER(X)` are used to declare the generic converters for DataObject and ContainedObject classes. For DataObject classes, the argument should be the class name itself (e.g. EventHeader), whereas for ContainedObject classes, the argument should be the class name concatenated with either List or Vector (e.g. CellVector) depending on whether the objects are associated with an ObjectList or ObjectVector.

2. The `DECLARE_ALGTOOL(X)` and `DECLARE_TOOL(X)` declarations are synonyms of each other.

3. Declarations of this form are used to declare components from explicit C++ namespaces. The first argument is the namespace (e.g. Atlfast), and the second is the class name (e.g. CellMaker).
17.3.1.2 Loading Components

Components within the component library are loaded in a file `MyComponents_load.cxx`. By convention, the name of this file is the package name concatenated with `_load`. The contents of this file are shown in Listing 17.5:

Listing 17.5 The `MyComponents_load.cxx` file
```
#include "Gaudi/Kernel/LoadFactoryEntries.h"

LOAD_FACTORY_ENTRIES(MyComponents)  [1]
```

Notes:

1. The argument to the `LOAD_FACTORY_ENTRIES` statement is the name of the component library.

17.3.1.3 Specifying component libraries at run-time

The fragments of the job options file or Python script that specifies the component library at run-time are shown in Listing 17.6a and Listing 17.6b.

Listing 17.6a Job options file fragment for specifying a component library at run-time
```
ApplicationMgr.Dlls += { "MyComponents" };  [1]
```

Listing 17.6b Python script fragment for specifying a component library at run-time
```
theApp.Dlls = [ "MyComponents" ]  [1]
```

Notes:

1. This is a list property, allowing multiple such libraries to be specified in a single line, separated by commas.

2. It is important to use the “+=” syntax in a job options file in order to append the new component library to any that might already have been configured. This is not necessary for a Python script.

17.3.2 Linker Libraries

These are libraries containing implementation classes. For example, libraries containing code of a number of base classes or concrete classes without abstract interfaces, etc. These libraries, in contrast to component libraries, export all their symbols and are needed during the linking phase in application building. These libraries can be linked to the application either statically or dynamically through the use of the corresponding type of library. In the first case the code is added physically to the executable file.
In this case, changes in these libraries require the application to be re-linked. In the second case, the linker only adds to the executable the minimal information required for loading the library and resolving the symbols at run time. Locating and loading the proper shared library at run-time is done using the \texttt{LD_LIBRARY_PATH} environment variable. Changes to the linker library will only require the application to be relinked if there is an interface change.

### 17.3.3 Dual purpose libraries and library strategy

It is also possible to have dual purpose libraries - ones which are simultaneously component and linker libraries. In general such libraries will contain DataObjects and ContainedObjects, together with their converters and associated factories. They are linker libraries since clients of these classes will need access to the header files and code, but they are component libraries since they have factories associated with the converters.

It is recommended that such dual purpose libraries be separated from single purpose component or linker libraries. Consider the case where several Algorithms share the use of several DataObjects (e.g. where one Algorithm creates and registers them with the transient event store, and another downstream Algorithm locates them), and also share the use of some helper classes in order to decode and manipulate the contents of the DataObjects. It is recommended that three different packages be used for this; one pure component package for the Algorithms, one dual-purpose for the DataObjects, and one pure linker package for the helper classes. Obviously the package handling the Algorithms will in general depend on the other packages. However, no other package should in general depend on the package handling the component library.

### 17.3.4 Building the different types of libraries

Using ATLAS SRT, component and linker libraries are differentiated by different options that are passed through to the linker. Specifically, component and dual-purpose libraries require lines to be added to the package \texttt{GNUmakefile.in} file, as illustrated in Listing 17.7.

#### Listing 17.7 Linker options for component library in GNUmakefile.in

```
libMyComponents.so_LDFLAGS = -Wl,Bsymbolic
```

Notes:

1. This is Lunix-specific. The equivalent instructions for other platforms still need to be defined.

### 17.3.5 Linking FORTRAN code

Any library containing FORTRAN code (more specifically, code that references COMMON blocks) must be linked statically. This is because COMMON blocks are, by definition, static entities. When mixing C++ code with FORTRAN, it is recommended that separate libraries for the C++ and
FORTRAN are built, and the code is written in such a way that communication between the C++ and FORTRAN worlds is done exclusively via wrappers. In this way it is possible to build shareable libraries for the C++ code, even if it calls FORTRAN code internally.
Chapter 18

Framework packages, interfaces and libraries

18.1 Overview

It is clearly important to decompose large software systems into hierarchies of smaller and more manageable entities. This decomposition can have important consequences for implementation related issues, such as compile-time and link dependencies, configuration management, etc. A package is the grouping of related components into a cohesive physical entity. A package is also the minimal unit of software release.

In this chapter we describe the Gaudi package structure, and how these packages are implemented in libraries.

18.2 Gaudi Package Structure

The Gaudi software is decomposed into the packages shown in Figure 18.1..

At the lower level we find GaudiKernel, which is the framework itself, and whose only dependency is on the GaudiPolicy package, which contains the various flags defining the CMT [6] configuration management environment needed to build the Gaudi software. At the next level are the packages containing standard framework components (GaudiSvc, GaudiDb, GaudiTools, GaudiAlg, GaudiAud), which depend on the framework and on widely available foundation libraries such as CLHEP and HTL. These external libraries are accessed via CMT interface packages which use environment variables defined in the ExternalLibs package, which should be tailored to
the software installation at a given site. All the above packages are grouped into the GaudiSys set of packages which are the minimal set required for a complete Gaudi installation.

The remaining packages are optional packages which can be used according to the specific technology choices for a given application. In this distribution, there are two specific implementations of the histogram persistency service, based on HBOOK (HbookCnv) and ROOT (RootHistCnv) and one implementation of the event data persistency service (GaudiRootDb) which understands ROOT databases. There is also a prototype scripting service (SIPython) depending on the Python scripting language. Finally, at the top level we find the applications (GaudiExamples) which depend on GaudiSys and the scripting and persistency services.

18.2.1 Gaudi Package Layout

Figure 18.1 shows the layout for Gaudi packages. Note that the binaries directories are not in CVS, they are created by CMT when building a package.
18.2.2 Packaging Guidelines

Packaging is an important architectural issue for the Gaudi framework, but also for the experiment specific software packages based on Gaudi. Typically, experiment packages consist of:

- Specific event model
- Specific detector description
- Sets of algorithms (digitisation, reconstruction, etc.)

The packaging should be such as to minimise the dependencies between packages, and must absolutely avoid cyclic dependencies. The granularity should not be too small or too big. Care should be taken to identify the external interfaces of packages: if the same interfaces are shared by many packages, they should be promoted to a more basic package that the others would then depend on. It is a good idea to discuss your packaging with the librarian and/or architect.
### 18.3 Interfaces in Gaudi

One of the main design choices at the architecture level in Gaudi was to favour abstract interfaces when building collaborations of various classes. This is the way we best decouple the client of a class from its real implementation.

An abstract interface in C++ is a class where all the methods are pure virtual. We have defined some practical guidelines for defining interfaces. An example is shown in Listing 18.1:

**Listing 18.1 Example of an abstract interface (IService)**

```cpp
1: // $Header: $
2: ifndef GAUDIKERNEL_ISERVICE_H
3: #define GAUDIKERNEL_ISERVICE_H
4:
5: // Include files
6: #include "GaudiKernel/IInterface.h"
7: #include <string>
8:
9: // Declaration of the interface ID. (id, major, minor)
10: static const InterfaceID IID_IService(2, 1, 0);
11:
12: /** @class IService IService.h GaudiKernel/IService.h
13: *
14: General service interface definition
15: *
16: @author Pere Mato
17: */
18: class IService : virtual public IInterface {
19: public:
20: /// Retrieve name of the service
21: virtual const std::string& name() const = 0;
23: virtual const IID& type() const = 0;
24: /// Initialize Service
25: virtual StatusCode initialize() = 0;
26: /// Finalize Service
27: virtual StatusCode finalize() = 0;
28: /// Retrieve interface ID
29: static const InterfaceID& interfaceID() { return IID_IService; }
30: 
31: #endif // GAUDIKERNEL_ISERVICE_H
32:
```

From this example we can make the following observations:

- **Interface Naming.** The name of the class has to start with capital “I” to denote that it is an interface.
Derived from IInterface. We follow the convention that all interfaces should be derived from a basic interface IInterface. This interface defined 3 methods: addRef(), release() and queryInterface(). This methods allow the framework to manage the reference counting of the framework components and the possibility to obtain a different interface of a component using any interface (see later).

Pure Abstract Methods. All the methods should be pure abstract (virtual ReturnType method(...) = 0;) With the exception of the static method interfaceID() (see later) and some inline templated methods to facilitate the use of the interface by the end-user.

Interface ID. Each interface should have a unique identification (see later) used by the query interface mechanism.

### 18.3.1 Interface ID

We needed to introduce an interface ID for identifying interfaces for the queryInterface functionality. The interface ID is made of a numerical identifier that needs to be allocated off-line, and a major and minor version numbers. The version number is used to decide if the interface the service provider is returning is compatible with the interface the client is expecting. The rules for deciding if the interface request is compatible are:

- The interface identifier is the same
- The major version is the same
- The minor version of the client is less than or equal to the one of the service provider. This allows the service provider to add functionality (incrementing minor version number) keeping old clients still compatible.

The interface ID is defined in the same header file as the rest of the interface. Care should be taken of globally allocating the interface identifier, and of modifying the version whenever a change of the interface is required, according to the rules. Of course changes to interfaces should be minimized.

```cpp
static const InterfaceID IID_Ixxx(2 /*id*/, 1 /*major*/, 0 /*minor*/);

class Ixxx : public IInterface {
    ...
    static const InterfaceID& interfaceID() { return IID_Ixxx; }
};
```

The static method Ixxx::interfaceID() is useful for the implementation of templated methods and classes using an interface as template parameter. The construct T::interfaceID() returns the interface ID of interface T.
18.3.2 Query Interface

The method queryInterface() is used to request a reference to an interface implemented by a component within the Gaudi framework. This method is implemented by each component class of the framework. Typically, this is not very visible since it is done in the base class from which you inherit. A typical implementation looks like this:

Listing 18.2 Example implementation of queryInterface()

```cpp
1: StatusCode DataSvc::queryInterface(const InterfaceID& riid,
2:                     void** ppvInterface) {
3:     if ( IID_IDataProviderSvc.versionMatch(riid) ) {
4:         *ppvInterface = (IDataProviderSvc*)this;
5:     }
6:     else if ( IID_IDataManagerSvc.versionMatch(riid) ) {
7:         *ppvInterface = (IDataManagerSvc*)this;
8:     }
9:     else {
10:    return Service::queryInterface(riid, ppvInterface);
11: }
12:    addRef();
13:    return SUCCESS;
14: }
```

The implementation returns the corresponding interface pointer if there is a match between the received InterfaceID and the implemented one. The method versionMatch() takes into account the rules mentioned in Section 18.3.1.

If the requested interface is not recognized at this level (line 9), the call can be forwarded to the inherited base class or possible sub-components of this component.

18.4 Libraries in Gaudi

Two different sorts of library can be identified that are relevant to the framework. These are component libraries, and linker libraries. These libraries are used for different purposes and are built in different ways.

18.4.1 Component libraries

Component libraries are shared libraries that contain standard framework components which implement abstract interfaces. Such components are Algorithms, Auditors, Services, Tools or Converters. These libraries do not export their symbols apart from one which is used by the framework to discover what components are contained by the library. Thus component libraries should not be linked against, they are used purely at run-time, being loaded dynamically upon request, the configuration being specified...
by the job options file. Changes in the implementation of a component library do not require the application to be relinked.

Component libraries contain factories for their components, and it is important that the factory entries are declared and loaded correctly. The following sections describe how this is done.

When a component library is loaded, the framework attempts to locate a single entrypoint, called `getFactoryEntries()`. This is expected to declare and load the component factories from the library. Several macros are available to simplify the declaration and loading of the components via this function.

Consider a simple package `MyComponents`, that declares and defines the `MyAlgorithm` class, being a subclass of `Algorithm`, and the `MyService` class, being a subclass of `Service`. Thus the package will contain the header and implementation files for these classes (`MyAlgorithm.h`, `MyAlgorithm.cpp`, `MyService.h` and `MyService.cpp`) in addition to whatever other files are necessary for the correct functioning of these components.

In order to satisfy the requirements of a component library, two additional files must also be present in the package. One is used to declare the components, the other to load them. Because of the technical limitations inherent in the use of shared libraries, it is important that these two files remain separate, and that no attempt is made to combine their contents into a single file.

The names of these files and their contents are described in the following sections.

### 18.4.1.1 Declaring Components

Components within the component library are declared in a file `MyComponents_entries.cpp`. By convention, the name of this file is the package name concatenated with `_entries`. The contents of this file are shown below:

**Listing 18.3 The `MyComponents_entries.cpp` file**

```cpp
#include "GaudiKernel/DeclareFactoryEntries.h"

DECLARE_FACTORY_ENTRIES( MyComponents ) { [1]
    DECLARE_ALGORITHM( MyAlgorithm ); [2]
    DECLARE_SERVICE ( MyService );
}
```

Notes:

1. The macros described in this section are currently only implemented in the Atlas extensions to Gaudi. They are documented here because they are expected to be included in a future release of Gaudi. The current release of Gaudi uses a similar mechanism but with slightly different naming conventions.
1. The argument to the DECLARE_FACTORY_ENTRIES statement is the name of the component library.

2. Each component within the library should be declared using one of the DECLARE_XXX statements discussed in detail in the next Section.

### 18.4.1.2 Component declaration statements

The complete set of statements that are available for declaring components is given below. They include those that support C++ classes in different namespaces, as well as for DataObjects or ContainedObjects using the generic converters.

**Listing 18.4 The available component declaration statements**

```
DECLARE_ALGORITHM(X)
DECLARE_AUDITOR(X)
DECLARE_CONVERTER(X)
DECLARE_GENERIC_CONVERTER(X) [1]
DECLARE_OBJECT(X)
DECLARE_SERVICE(X)

DECLARE_NAMESPACE_ALGORITHM(N,X) [2]
DECLARE_NAMESPACE_AUDITOR(N,X)
DECLARE_NAMESPACE_CONVERTER(N,X)
DECLARE_NAMESPACE_GENERIC_CONVERTER(N,X)
DECLARE_NAMESPACE_OBJECT(N,X)
DECLARE_NAMESPACE_SERVICE(N,X)
```

**Notes:**

1. Declarations of the form DECLARE_GENERIC_CONVERTER(X) are used to declare the generic converters for DataObject and ContainedObject classes. For DataObject classes, the argument should be the class name itself (e.g. EventHeader), whereas for ContainedObject classes, the argument should be the class name concatenated with either List or Vector (e.g. CellVector) depending on whether the objects are associated with an ObjectList or ObjectVector.

2. Declarations of this form are used to declare components from explicit C++ namespaces. The first argument is the namespace (e.g. Atlfast), the second is the class name (e.g. CellMaker).
18.4.1.3 Loading Components

Components within the component library are loaded in a file `MyComponents_load.cpp`. By convention, the name of this file is the package name concatenated with `_load`. The contents of this file are shown below:

**Listing 18.5 The MyComponents_load.cpp file**

```cpp
#include "GaudiKernel/LoadFactoryEntries.h"

LOAD_FACTORY_ENTRIES( MyComponents ) [1]
```

Notes:
1. The argument of `LOAD_FACTORY_ENTRIES` is the name of the component library.

18.4.1.4 Specifying component libraries at run-time

The fragment of the job options file that specifies the component library at run-time is shown below.

**Listing 18.6 Selecting and running the desired tutorial example**

```cpp
ApplicationMgr.DLLs += { "MyComponents" }; [1]
```

Notes:
1. This is a list property, allowing multiple such libraries to be specified in a single line.
2. It is important to use the “+=” syntax to append the new component library or libraries to any that might already have been configured.

The convention in Gaudi is that component libraries have the same name as the package they belong to (prefixed by "lib" on Linux). When trying to load a component library, the framework will look for it in various places following this sequence:

- Look for an environment variable with the name of the package, suffixed by "Shr" (e.g. `${MyComponentsShr}`). If it exists, it should translate to the full name of the library, without the file type suffix (e.g. `${MyComponentsShr} = "$MYSOFT/MyComponents/v1/i386_linux22/libMyComponents"`).
- Try to locate the file `libMyComponents.so` using the `LD_LIBRARY_PATH` (on Linux), or `MyComponents.dll` using the `PATH` (on Windows).

18.4.2 Linker libraries

These are libraries containing implementation classes. For example, libraries containing code of a number of base classes or specific classes without abstract interfaces, etc. These libraries, contrary to
the component libraries, export all the symbols and are needed during the linking phase in the application building. These libraries can be linked to the application "statically" or "dynamically", requiring a different file format. In the first case the code is added physically to the executable file. In this case, changes in these libraries require the application to be re-linked, even if these changes do not affect the interfaces. In the second case, the linker only adds into the executable minimal information required for loading the library and resolving the symbols at run time. Locating and loading the proper shareable library at run time is done exclusively using the LD_LIBRARY_PATH for Linux and PATH for Windows. The convention in Gaudi is that linker libraries have the same name as the package, suffixed by "Lib" (and prefixed by "lib" on Linux, e.g. libMyComponentsLib.so).

18.4.3 Library strategy and dual purpose libraries

Because component libraries are not designed to be linked against, it is important to separate the functionalities of these libraries from linker libraries. For example, consider the case of a DataProvider service that provides DataObjects for clients. It is important that the declarations and definitions of the DataObjects be handled by a different shared library than that handling the service itself. This implies the presence of two different packages - one for the component library, the other for the DataObjects. Clients should only depend on the second of these packages. Obviously the package handling the component library will in general also depend on the second package.

It is possible to have dual purpose libraries - ones which are simultaneously component and linker libraries. In general such libraries will contain DataObjects and ContainedObjects, together with their converters and associated factories. It is recommended that such dual purpose libraries be separated from single purpose component or linker libraries. Consider the case where several Algorithms share the use of several DataObjects (e.g. where one Algorithm creates them and registers them with the transient event store, and another Algorithm locates them), and also share the use of some helper classes in order to decode and manipulate the contents of the DataObjects. It is recommended that three different packages be used for this - one pure component package for the Algorithms, one dual-purpose for the DataObjects, and one pure linker package for the helper classes.

18.4.4 Building and linking with the libraries

Gaudi libraries and applications are built using CMT, but may be used also by experiments using other configuration management tools, such as ATLAS SRT.

18.4.4.1 Building and linking to the Gaudi libraries with CMT

Gaudi libraries and applications are built using CMT taking advantage of the CMT macros defined in the GaudiPolicy package. As an example, the CMT requirements file of the GaudiTools package is shown in Listing 18.7. The linker and component libraries are defined on lines 23 and 26 respectively - the linker library is defined first because it must be built ahead of the component library. Lines 28 and 34 set up the generic linker options and flags for the linker library, which are suffixed by the package specific flags set up by line 35. Line 31 tells CMT to generate the symbols needed for the
component library, while line 33 sets up the corresponding linker flags for the component library. Finally, line 30 updates LD_LIBRARY_PATH (or PATH on Windows) for this package. In packages with only a component library and no linker library, line 30 could be replaced by "apply_pattern packageShr", which would create the logical name required to access the component library by the first of the two methods described in Section 18.4.1.4.

Listing 18.7 CMT requirements file for the GaudiTools package

```
15: package GaudiTools
16:   version v1
17:
18: branches GaudiTools cmt doc src
19:   use GaudiKernel v8*
20:   include_dirs "$(GAUDITOOLSROOT)"
21:
22: #linker library
23:   library GaudiToolsLib ../src/Associator.cpp ../src/IInterface.cpp
24:
25: #component library
26:   library GaudiTools ../src/GaudiTools_load.cpp ../src/GaudiTools_dll.cpp
27:
28:   apply_pattern package_Llinkopts
29:
30:   apply_pattern ld_library_path
31:   macro_append GaudiTools_stamps "$(GaudiToolsDir)/GaudiToolsLib.stamp"
32:
33:   apply_pattern package_Cshlibflags
34:   apply_pattern package_Lshlibflags
35:   macro_append GaudiToolsLib_shlibflags "$(GaudiKernel_linkopts)"
```

18.4.4.2 Linking to the Gaudi libraries with Atlas SRT

Using ATLAS SRT, component and linker libraries are differentiated by different options that are passed through to the linker. Specifically, component and dual-purpose libraries require lines to be added to the package GNUmakefile.in file, as illustrated in Listing 18.8.

Listing 18.8 Linker options for component library in GNUmakefile.in

```
libMyComponents.so_LDFLAGS = -Wl,Bsymbolic
libMyComponents.so_LIBS = -lGaudiBase [1]
```

Notes:
1. This line is only strictly necessary when other dual-purpose libraries are linked to as a result of the package dependencies. However, since this dependency can come from other packages than those that the current package directly depends on, it is recommended that this line always be present. It is not harmful if used when it isn’t necessary.

18.4.5 Linking FORTRAN code

Any library containing FORTRAN code (more specifically, code that references COMMON blocks) must be linked statically. This is because COMMON blocks are, by definition, static entities. When mixing C++ code with FORTRAN, it is recommended to build separate libraries for the C++ and FORTRAN, and to write the code in such a way that communication between the C++ and FORTRAN worlds is done exclusively via wrappers. This makes it possible to build shareable libraries for the C++ code, even if it calls FORTRAN code internally.
Chapter 19
Analysis utilities

19.1 Overview

In this chapter we give pointers to some of the third party software libraries that we use within Athena or recommend for use by algorithms implemented in Athena.

19.2 CLHEP

CLHEP ("Class Library for High Energy Physics") is a set of HEP-specific foundation and utility classes such as random generators, physics vectors, geometry and linear algebra. It is structured in a set of packages independent of any external package. The documentation for CLHEP can be found on WWW at http://wwwinfo.cern.ch/asd/lhc++/clhep/index.html

CLHEP is used extensively inside Athena, in the GaudiSvc and GaudiDbHcbEvent packages.

19.3 HTL

HTL ("Histogram Template Library") is used internally in Athena (GaudiSvc package) to provide histogramming functionality. It is accessed through its abstract AIDA compliant interfaces. Athena uses only the transient part of HTL. Histogram persistency is available with ROOT or HBOOK.

19.4 NAG C

The NAG C library is a commercial mathematical library providing a similar functionality to the FORTRAN mathlib (part of CERNLIB). It is organised into chapters, each chapter devoted to a branch of numerical or statistical computation. A full list of the functions is available at http://wwwinfo.cern.ch/asd/lhc+/+Nag_C/html/doc.html

NAG C is not explicitly used in the Athena framework, but developers are encouraged to use it for mathematical computations. Instructions for linking NAG C with Athena can be found at http://cern.ch/lhcb-comp/Components/html/nagC.html

Some NAG C functions print error messages to stdout by default, without any information about the calling algorithm and without filtering on severity level. A facility is provided by Athena to redirect these messages to the Athena MessageSvc. This is documented at http://cern.ch/lhcb-comp/Components/html/GaudiNagC.html

19.5 ROOT

ROOT is used by Athena for I/O and as a persistency solution for event data, histograms and n-tuples. In addition, it can be used for interactive analysis, as discussed in Chapter 12. Information about ROOT can be found at http://root.cern.ch/
Appendix A

References

[3] HepMC Reference
[4] Python Reference
Appendix B
Options for standard components

The following is a list of options that may be set for the standard components: e.g. data files for input, print-out level for the message service, etc. The options are listed in tabular form for each component along with the default value and a short explanation. The component name is given in the table caption thus: [ComponentName].

Table B.1 Standard Options for the Application manager [ApplicationMgr]

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EvtSel⁵</td>
<td>&quot;NONE&quot;; (if no event input) ⁶</td>
<td></td>
</tr>
<tr>
<td>EvtMax</td>
<td>-1</td>
<td>Maximum number of events to process. The default is -1 (infinite) unless EvtSel = &quot;NONE&quot;; in which case it is 10.</td>
</tr>
</tbody>
</table>
| TopAlg              | ()            | List of top level algorithms. Format: 
{<Type>/<Name>[, <Type2>/<Name2>,...]};                                          |
| ExtSvc              | ()            | List of external services names (not known to the ApplicationMgr, see section 13.2). Format: 
{<Type>/<Name>[, <Type2>/<Name2>,...]};                                          |
| OutStream           | ()            | Declares an output stream object for writing data to a persistent store, e.g. {"DstWriter"}; See also Table B.10 |
| DLLs                | ()            | Search list of libraries for dynamic loading. Format: 
{<dll1>[,<dll2>,...]};                                                              |
| HistogramPersistency| "NONE"        | Histogram persistency mechanism. Available options are "HBOOK", "ROOT", "NONE" |
| Runable             | "AppMgrRunnable" | Type of runnable object to be created by Application manager |
EventLoop "GaudiEventLoopMgr" Type of event loop:
"GaudiEventLoopMgr" is standard event loop
"MinimalEventLoop" executes algorithms but does not read events

The last two options define the source of the job options file and so they cannot be defined in the job options file itself. There are two possibilities to set these options, the first one is using an environment variable called JOBOPTPATH or setting the option to the application manager directly from the main program\footnote{The coded option takes precedence.}

Table B.1 Standard Options for the Application manager [ApplicationMgr]

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| EventLoop     | "GaudiEventLoopMgr" | Type of event loop:  
"GaudiEventLoopMgr" is standard event loop  
"MinimalEventLoop" executes algorithms but does not read events |

Table B.2 Standard Options for the message service [MessageSvc]

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| OutputLevel | 0             | Verbose
ess threshold level:  
0=NIL, 1=VERBOSE, 2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL |
| Format      | "% F%18W%S%7W%R%T %0W%M" | Format string. |

Table B.3 Standard Options for all algorithms [<myAlgorithm>]

Any algorithm derived from the Algorithm base class can override the global Algorithm options thus:

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| OutputLevel     | 0             | Message Service Verbose
ess threshold level:  
0=NIL, 1=VERBOSE, 2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL |
| Enable          | true          | If false, application manager skips execution of this algorithm         |
| ErrorMax        | 1             | Job stops when this number of errors is reached                         |
| ErrorCount      | 0             | Current error count                                                     |
| AuditInitialize | false         | Enable/Disable auditing of Algorithm initialisation                     |
| AuditExecute    | true          | Enable/Disable auditing of Algorithm execution                          |
| AuditFinalize   | false         | Enable/Disable auditing of Algorithm finalisation                       |
Table B.4 Standard Options for all services [<myService>]

Any service derived from the Service base class can override the global MessageSvc.OutputLevel thus:

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputLevel</td>
<td>0</td>
<td>Message Service Verboseness threshold level: 0=NIL, 1=VERBOSE, 2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL</td>
</tr>
</tbody>
</table>

Table B.5 Standard Options for all Tools [<myTool>]

Any tool derived from the AlgTool base class can override the global MessageSvc.OutputLevel thus:

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputLevel</td>
<td>0</td>
<td>Message Service Verboseness threshold level: 0=NIL, 1=VERBOSE, 2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL</td>
</tr>
</tbody>
</table>

Table B.6 Standard Options for all Associators [<myAssociator>]

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FollowLinks</td>
<td>true</td>
<td>Instruct the associator to follow the links instead of using cached information</td>
</tr>
<tr>
<td>DataLocation</td>
<td>&quot;&quot;</td>
<td>Location where to get association information in the data store</td>
</tr>
</tbody>
</table>

Table B.7 Standard Options for Auditor service [AuditorSvc]

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auditors</td>
<td>[]</td>
<td>List of Auditors to be loaded and to be used. See section 13.7 for list of possible auditors</td>
</tr>
</tbody>
</table>

Table B.8 Standard Options for all Auditors [<myAuditor>]

Any Auditor derived from the Auditor base class can override the global Auditor options thus:

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputLevel</td>
<td>0</td>
<td>Message Service Verboseness threshold level: 0=NIL, 1=VERBOSE, 2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL</td>
</tr>
<tr>
<td>Enable</td>
<td>true</td>
<td>If false, application manager skips execution of the auditor</td>
</tr>
</tbody>
</table>
### Table B.9 Options of Algorithms in GaudiAlg package

<table>
<thead>
<tr>
<th>Algorithm name</th>
<th>Option Name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventCounter</td>
<td>Frequency</td>
<td>1;</td>
<td>Frequency with which number of events should be reported</td>
</tr>
<tr>
<td>Prescaler</td>
<td>PercentPass</td>
<td>100.0;</td>
<td>Percentage of events that should be passed</td>
</tr>
<tr>
<td>Sequencer</td>
<td>Members</td>
<td></td>
<td>Names of members of the sequence</td>
</tr>
<tr>
<td>Sequencer</td>
<td>StopOverride</td>
<td>false;</td>
<td>If true, do not stop sequence if a filter fails</td>
</tr>
</tbody>
</table>

### Table B.10 Options available for output streams (e.g. DstWriter)

Output stream objects are used for writing user created data into data files or databases. They are created and named by setting the option `ApplicationMgr.OutStream`. For each output stream the following options are available.

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| ItemList       | {}            | The list of data objects to be written to this stream, e.g. 
|                |               | {"/Event#1","Event/MyTracks/#1"}; |
| EvtDataSvc     | “EventDataSvc”| The service from which to retrieve objects. |
| Output         | {}            | Output data stream specification. Format: 
|                |               | {"DATAFILE='mydst.root' TYP='ROOT"}; |
| AcceptAlgs     | {}            | If any of these algorithms sets filterflag=true; the event is accepted |
| RequireAlgs    | {}            | If any of these algorithms is not executed, the event is rejected |
| VetoAlgs       | {}            | If any of these algorithms does not set filterflag = true; the event is rejected |

### Table B.11 Standard Options for persistency services (e.g. EventPersistencySvc)

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CnvServices</td>
<td>{}</td>
<td>Conversion services to be used by the service to load or store persistent data (e.g. &quot;RootEvtCnvSvc&quot;)</td>
</tr>
</tbody>
</table>

### Table B.12 Standard Options for conversion services (e.g. RootEvtCnvSvc)

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbType</td>
<td>&quot;&quot;</td>
<td>Persistency technology (e.g. &quot;ROOT&quot;)</td>
</tr>
</tbody>
</table>
### Table B.13 Standard Options for the histogram service [HistogramPersistencySvc]

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputFile</td>
<td>&quot;&quot;</td>
<td>Output file for histograms. No output if not defined</td>
</tr>
</tbody>
</table>

### Table B.14 Standard Options for the N-tuple service [NTupleSvc] (see section 12.2.3.1)

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>{}</td>
<td>Input file(s) for n-tuples. Format: <code>{&quot;FILE1 DATAFILE='tuple.hbook' OPT='OLD' TYP='HBOOK'&quot;</code>, <code>{&quot;FILE2 DATAFILE='tuple.root' OPT='OLD' TYP='ROOT'&quot;</code>...}]`</td>
</tr>
<tr>
<td>Output</td>
<td>{}</td>
<td>Output file of n-tuples. Format: <code>{&quot;FILE1 DATAFILE='tuple.hbook' OPT='NEW' TYP='HBOOK'&quot;</code>, <code>{&quot;FILE2 DATAFILE='tuple.root' OPT='NEW' TYP='ROOT'&quot;</code>...}]`</td>
</tr>
<tr>
<td>StoreName</td>
<td>&quot;/NTUPLES&quot;</td>
<td>Name of top level entry</td>
</tr>
</tbody>
</table>

### Table B.15 Standard Options for the standard event selector [EventSelector]

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>{}</td>
<td>Input data stream specification. Format: &quot;&lt;tagname&gt; = '&lt;tagvalue&gt;' &lt;opt&gt;&quot; Possible &lt;tagname&gt;: DATAFILE = 'filename', TYP = 'technology type' OPT = 'new'</td>
</tr>
<tr>
<td>FirstEvent</td>
<td>1</td>
<td>First event to process (allows skipping of preceding events)</td>
</tr>
<tr>
<td>EvtMax</td>
<td>All events on Application-Mgr.EvtSel input stream</td>
<td>Maximum number of events to process</td>
</tr>
<tr>
<td>PrintFreq</td>
<td>10</td>
<td>Frequency with which event number is reported</td>
</tr>
<tr>
<td>JobInput</td>
<td>&quot;&quot;</td>
<td>String of input files (same format as ApplicationMgr.EvtSel), used only for pileup event selector(s)</td>
</tr>
</tbody>
</table>
### Table B.16  Event Tag Collection Selector [EventCollectionSelector]

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CnvService</td>
<td>“EvtTupleSvc”</td>
<td>Conversion service to be used</td>
</tr>
<tr>
<td>Authentication</td>
<td>“”</td>
<td>Authentication to be used</td>
</tr>
<tr>
<td>Container</td>
<td>“B2PiPi”</td>
<td>Container name</td>
</tr>
<tr>
<td>Item</td>
<td>”Address”</td>
<td>Item name</td>
</tr>
<tr>
<td>Criteria</td>
<td>“”</td>
<td>Selection criteria</td>
</tr>
<tr>
<td>DB</td>
<td>“”</td>
<td>Database name</td>
</tr>
<tr>
<td>DbType</td>
<td>“”</td>
<td>Database type</td>
</tr>
<tr>
<td>Function</td>
<td>“NTuple::Selector”</td>
<td>Selection function</td>
</tr>
</tbody>
</table>

### Table B.17  Standard Options for Particle Property Service [ParticlePropertySvc]

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParticlePropertiesFile</td>
<td>“($LHCBDBASE)/cdf/particle.cdf”</td>
<td>Particle properties database location</td>
</tr>
</tbody>
</table>

### Table B.18  Standard Options for Random Numbers Generator Service [RndmGenSvc]

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine</td>
<td>“HepRndm::Engine&lt;RanluxEngine&gt;”</td>
<td>Random number generator engine</td>
</tr>
<tr>
<td>Seeds</td>
<td></td>
<td>Table of generator seeds</td>
</tr>
<tr>
<td>Column</td>
<td>0</td>
<td>Number of columns in seed table -1</td>
</tr>
<tr>
<td>Row</td>
<td>1</td>
<td>Number of rows in seed table -1</td>
</tr>
<tr>
<td>Luxury</td>
<td>3</td>
<td>Luxury value for the generator</td>
</tr>
<tr>
<td>UseTable</td>
<td>false</td>
<td>Switch to use seeds table</td>
</tr>
</tbody>
</table>

### Table B.19  Standard Options for Chrono and Stat Service [ChronoStatSvc]

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChronoPrintOutTable</td>
<td>true</td>
<td>Global switch for profiling printout</td>
</tr>
<tr>
<td>PrintUserTime</td>
<td>true</td>
<td>Switch to print User Time</td>
</tr>
<tr>
<td>PrintSystemTime</td>
<td>false</td>
<td>Switch to print System Time</td>
</tr>
<tr>
<td>PrintElapsedTime</td>
<td>false</td>
<td>Switch to print Elapsed time (Note typo in option name!)</td>
</tr>
<tr>
<td>ChronoDestinationCout</td>
<td>false</td>
<td>If true, printout goes to cout rather than MessageSvc</td>
</tr>
<tr>
<td>Option name</td>
<td>Default value</td>
<td>Meaning</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>ChronoPrintLevel</td>
<td>3</td>
<td>Print level for profiling (values as for MessageSvc)</td>
</tr>
<tr>
<td>ChronoTableToBeOrdered</td>
<td>true</td>
<td>Switch to order printed table</td>
</tr>
<tr>
<td>StatPrintOutTable</td>
<td>true</td>
<td>Global switch for statistics printout</td>
</tr>
<tr>
<td>StatDestinationCout</td>
<td>false</td>
<td>If true, printout goes to cout rather than MessageSvc</td>
</tr>
<tr>
<td>StatPrintLevel</td>
<td>3</td>
<td>Print level for profiling (values as for MessageSvc)</td>
</tr>
<tr>
<td>StatTableToBeOrdered</td>
<td>true</td>
<td>Switch to order printed table</td>
</tr>
</tbody>
</table>
Appendix C
Design considerations

C.1 Generalities

In this chapter we look at how you might actually go about designing and implementing a real physics algorithm. It includes points covering various aspects of software development process and in particular:

- The need for more “thinking before coding” when using an OO language like C++.
- Emphasis on the specification and analysis of an algorithm in mathematical and natural language, rather than trying to force it into (unnatural?) object orientated thinking.
- The use of OO in the design phase, i.e. how to map the concepts identified in the analysis phase into data objects and algorithm objects.
- The identification of classes which are of general use. These could be implemented by the computing group, thus saving you work!
- The structuring of your code by defining private utility methods within concrete classes.

When designing and implementing your code we suggest that your priorities should be as follows: (1) Correctness, (2) Clarity, (3) Efficiency and, very low in the scale, OOness

Tips about specific use of the C++ language can be found in the coding rules document [5] or specialized literature.
C.2 Designing within the Framework

A physicist designing a real physics algorithm does not start with a white sheet of paper. The fact that he or she is using a framework imposes some constraints on the possible or allowed designs. The framework defines some of the basic components of an application and their interfaces and therefore it also specifies the places where concrete physics algorithms and concrete data types will fit in with the rest of the program. The consequences of this are: on one hand, that the physicists designing the algorithms do not have complete freedom in the way algorithms may be implemented; but on the other hand, neither do they need worry about some of the basic functionalities, such as getting end-user options, reporting messages, accessing event and detector data independently of the underlying storage technology, etc. In other words, the framework imposes some constraints in terms of interfaces to basic services, and the interfaces the algorithm itself is implementing towards the rest of the application. The definition of these interfaces establishes the so called “master walls” of the data processing application in which the concrete physics code will be deployed. Besides some general services provided by the framework, this approach also guarantees that later integration will be possible of many small algorithms into a much larger program, for example a reconstruction program. In any case, there is still a lot of room for design creativity when developing physics code within the framework and this is what we want to illustrate in the next sections.

To design a physics algorithm within the framework you need to know very clearly what it should do (the requirements). In particular you need to know the following:

- What is the input data to the algorithm? What is the relationship of these data to other data (e.g. event or detector data)?
- What new data is going to be produced by the algorithm?
- What’s the purpose of the algorithm and how is it going function? Document this in terms of mathematical expressions and plain english.¹
- What does the algorithm need in terms of configuration parameters?
- How can the algorithm be partitioned (structured) into smaller “algorithm chunks” that make it easier to develop (design, code, test) and maintain?
- What data is passed between the different chunks? How do they communicate?
- How do these chunks collaborate together to produce the desired final behaviour? Is there a controlling object? Are they self-organizing? Are they triggered by the existence of some data?
- How is the execution of the algorithm and its performance monitored (messages, histograms, etc.)?
- Who takes the responsibility of bootstrapping the various algorithm chunks.

For didactic purposes we would like to illustrate some of these design considerations using a hypothetical example. Imagine that we would like to design a tracking algorithm based on a Kalman-filter algorithm.

¹ Catalan is also acceptable.
C.3 Analysis Phase

As mentioned before we need to understand in detail what the algorithm is supposed to do before we start designing it and of course before we start producing lines of C++ code. One old technique for that, is to think in terms of data flow diagrams, as illustrated in Figure A.1, where we have tried to decompose the tracking algorithm into various processes or steps.

![Data flow diagram](image)

**Figure A.1** Hypothetical decomposition of a tracking algorithm based on a Kalman filter using a Data flow Diagram

In the analysis phase we identify the data which is needed as input (event data, geometry data, configuration parameters, etc.) and the data which is produced as output. We also need to think about the intermediate data. Perhaps this data may need to be saved in the persistency store to allow us to run a part of the algorithm without starting always from the beginning.

We need to understand precisely what each of the steps of the algorithm is supposed to do. In case a step becomes too complex we need to sub-divide it into several ones. Writing in plain english and using
mathematics whenever possible is extremely useful. The more we understand about what the algorithm has to do the better we are prepared to implement it.

C.4 Design Phase

We now need to decompose our physics algorithm into one or more Algorithms (as framework components) and define the way in which they will collaborate. After that we need to specify the data types which will be needed by the various Algorithms and their relationships. Then, we need to understand if these new data types will be required to be stored in the persistency store and how they will map to the existing possibilities given by the object persistency technology. This is done by designing the appropriate set of Converters. Finally, we need to identify utility classes which will help to implement the various algorithm chunks.

C.4.1 Defining Algorithms

Most of the steps of the algorithm have been identified in the analysis phase. We need at this moment to see if those steps can be realized as framework Algorithms. Remember that an Algorithm from the viewpoint of the framework is basically a quite simple interface (initialize, execute, finalize) with a few facilities to access the basic services. In the case of our hypothetical algorithm we could decide to have a “master” Algorithm which will orchestrate the work of a number of sub-Algorithms. This master Algorithm will be also be in charge of bootstrapping them. Then, we could have an Algorithm in charge of finding the tracking seeds, plus a set of others, each one associated to a different tracking station in charge of propagating a proto-track to the next station and deciding whether the proto-track needs to be kept or not. Finally, we could introduce another Algorithm in charge of producing the final tracks from the surviving proto-tracks.

It is interesting perhaps in this type of algorithm to distribute parts of the calculations (extrapolations, etc.) to more sophisticated “hits” than just the unintelligent original ones. This could be done by instantiating new data types (clever hits) for each event having references to the original hits. For that, it would be required to have another Algorithm whose role is to prepare these new data objects, see Figure A.2.

The master Algorithm (TrackingAlg) is in charge of setting up the other algorithms and scheduling their execution. It is the only one that has a global view but it does not need to know the details of how the different parts of the algorithm have been implemented. The application manager of the framework only interacts with the master algorithm and does not need to know that in fact the tracking algorithm is implemented by a collaboration of Algorithms.

C.4.2 Defining Data Objects

The input, output and intermediate data objects need to be specified. Typically, the input and output are specified in a more general way (algorithm independent) and basically are pure data objects. This is
Figure A.2 Object diagram (a) and class diagram (b) showing how the complete example tracking algorithm could be decomposed into a set of specific algorithms that collaborate to perform the complete task.
because they can be used by a range of different algorithms. We could have various types of tracking algorithm all using the same data as input and producing similar data as output. On the contrary, the intermediate data types can be designed to be very algorithm dependent.

The way we have chosen to communicate between the different *Algorithms* which constitute our physics algorithm is by using the transient event data store. This allows us to have low coupling between them, but other ways could be envisaged. For instance, we could implement specific methods in the algorithms and allow other “friend” algorithms to use them directly.

Concerning the relationships between data objects, it is strongly discouraged to have links from the input data objects to the newly produced ones (i.e. links from hits to tracks). In the other direction this should not be a problem (i.e from tracks to constituent hits).

For data types that we would like to save permanently we need to implement a specific *Converter*. One converter is required for each type of data and each kind of persistency technology that we wish to use. This is not the case for the data types that are used as intermediate data, since these data are completely transient.

### C.4.3 Mathematics and other utilities

It is clear that to implement any algorithm we will need the help of a series of utility classes. Some of these classes are very generic and they can be found in common class libraries. For example the standard template library. Other utilities will be more high energy physics specific, especially in cases like fitting, error treatment, etc. We envisage making as much use of these kinds of utility classes as possible.

Some algorithms or algorithm-parts could be designed in a way that allows them to be reused in other similar physics algorithms. For example, perhaps fitting or clustering algorithms could be designed in a generic way such that they can be used in various concrete algorithms. During design is the moment to identify this kind of re-usable component or to identify existing ones that could be used instead and adapt the design to make possible their usage.
19.6 Overview

Installation of Athena at a remote site involves the following major components.

1. Installation of external packages. These should generally be very stable such that this procedure need not be repeated very frequently.

2. Installation of the GAUDI package. This is currently being modified frequently, and typically a new version is necessary for each Athena release. It is obviously hoped that eventually this will reach a level of functionality where installation of a new version will not be necessary for every Athena release.

3. Installation of the ATLAS release.

19.6.1 Acknowledgements

The following abbreviated guide is a summary of more detailed information that is available from the following sources:

1. Kristo Karr (Affiliation?) has put together a set of web pages that details installation instructions for various ATLAS releases accessible from the Trigger Web page at:

URL

Iwona Sajda (LBNL) has compiled a web page based on her experience installing ATLAS software at LBNL. The URL is:
http://www-rnc.lbl.gov/~sakrejda/ATLASREADME.txt

19.6.2 External Packages

Athena 1.3.2 depends upon the external packages and versions shown in Listing 19.1:

Listing 19.1 External packages and versions

<table>
<thead>
<tr>
<th>Package</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLHEP</td>
<td>1.6.0.0</td>
</tr>
<tr>
<td>HTL</td>
<td>1.3.0.1</td>
</tr>
<tr>
<td>CERNLIB</td>
<td>2000</td>
</tr>
<tr>
<td>ROOT</td>
<td>2.25</td>
</tr>
<tr>
<td>Qt</td>
<td>2.2.1</td>
</tr>
<tr>
<td>CMT</td>
<td>v1r7</td>
</tr>
<tr>
<td>GAUDI</td>
<td>0.7.0</td>
</tr>
</tbody>
</table>

Notes:

1. CLHEP and HTL are both part of the LHC++ set of packages.
2. The dependencies on ROOT come from the ROOT generic converters as described in Chapter 6, and from the ROOT histogram and ntuple persistency service available from GAUDI.
3. The Qt package is required by the ATLAS graphics environment.
4. CMT notes here.
5. This Gaudi version convention has been adopted by ATLAS. This version corresponds to the official GAUDI release v7, with some patches. Note that the other packages must be installed before attempting to build the GAUDI package.

19.6.3 Installing CLHEP

This section is in preparation.

19.6.4 Installing HTL

This section is in preparation.

19.6.5 Installing CERNLIB

This section is in preparation.
19.6.6 Installing ROOT

This section is in preparation.

19.6.7 Installing Qt

The section is in preparation.

19.6.8 Installing CMT

This section is in preparation.

19.6.9 Installing GAUDI

The GAUDI 0.7.0 release is located at:

/afs/cern.ch/atlas/project/Gaudi/install/0.7.0.tar.gz

This should be transferred to the desired location on the remote machine.

19.6.10 Installing the ATLAS release

This section is in preparation.
A
AIDA 169
  see Interfaces
Algorithm 14
  Base class 15, 23
  branches 30
  Concrete 23, 26
  Constructor 25, 26
  Declaring properties 25
  Execution 27
  Filters 30
  Finalisation 29
  Initialisation 25, 27, 29
  Nested 29
  sequences 30
  Setting properties 25
Algorithms
  EventCounter 31, 98
  Prescaler 31
  Sequencer 31
Application Manager 16
Architecture 13
Associators 131
  Example 134
B
Branches 30
C
Casting
  of DataObjects 45
Checklist
  for implementing algorithms 29
Class
  identifier (CLID) 51
CLHEP 169
CMT
  Building libraries with 166
Component 13, 162
  libraries 162, 165
Component libraries 149, 150
  component libraries 153
  components 150
ContainedObject 47
Converters 137
Data Store 43
    finding objects in 45, 51
    Histograms 77
    registering objects into 46
DataObject 15, 44, 45, 47
    ownership 46
DECLARE_ALGORITHM 151, 164
DECLARE_FACTORY_ENTRIES 151, 164
endreq, MsgStream manipulator 106
Event Collections 89
    Filling 90
    Reading Events with 91
    Writing 89
EventCounter algorithm. See Algorithms Examples
    Associator 134
Exception
    when casting 45
Factory
    for a concrete algorithm 25
Filters 30
FORTRAN 14, 154
    and shareable libraries 168
getFactoryEntries 151, 163
Guidelines
    for software packaging 159
H
HBOOK
    Constraints on histograms 79
    For histogram persistency 80
    Limitations on N-tuples 84, 89
Histograms
    HTL 169
    Persistency service 80
HTL 169
I
Inheritance 23
Intefaces
    IConversionSvc 138
Interactive Analysis
of N-tuples 92
Interface 13
   and multiple inheritance 17
   Identifier 17
   In C++ 17
Interface ID 161
Interfaces
   AIDA 77, 169
   IAlgorithm 17, 23, 25, 27
   IAlgTool 126
   IAssociator 132
   IAuditor 114
   IAxis 77
   IConverter 139
   IDataManager 16
   IDDataProviderSvc 16, 44, 83
   IDDataProvideSvc 77
   IHistogram 77
   IHistogram1D 77
   IHistogram2D 77
   IHistogramSvc 16, 44, 77
   IIncidentListener 118
   IMessageSvc 17
   in Gaudi 160
   INTupleSvc 44, 83
   INtupleSvc 16
   IOpaqueAddress 139
   IParticlePropertySvc 107
   IProperty 17, 23
   ISvcLocator 25
   IToolSvc 130
L
LD_LIBRARY_PATH 154
Libraries
   Building 166
   Building, with CMT 166
   Component 162, 165
   containing FORTRAN code 168
   Linker 165
Linker Libraries 153
LOAD_FACTORY_ENTRIES 153, 165
M
Message service 104
Monitoring
  of algorithm calls, with the Auditor service 113
  statistical, using the Chrono&Stat service 111
Monte Carlo truth
  navigation using Associators 131
N
NAG C 170
N-tuples 83
  Booking and declaring tags 85
  filling 85
  Interactive Analysis of 92
  Limitations imposed by HBOOK 84, 89
  persistency 87
  reading 86
  Service 83
O
Object Container 47
  and STL 47
ObjectList 47
ObjectVector 47
P
Package 157
  Internal layout 158
  structure of LHCb software 157
Packages
  Dependencies of Gaudi 157
  Guidelines 159
PAW
  for N-Tuple analysis 92
Persistency
  of histograms 80
  of N-tuples 87
Persistent store
  saving data to 53
Prescaler algorithm. See Algorithms
Profiling
  of execution time, using the Chrono&Stat service 110
  of execution time, with the Auditor service 113
  of memory usage, with the Auditor service 113
R
Random numbers
  generating 115
  Service 115
Retrieval 130
ROOT 170
   for histogram persistency 80
   for N-Tuple analysis 93
S
Saving data 53
Sequencer algorithm. See Algorithms
Sequences 30
Services 15
   Auditor Service 113
   Chrono&Stat service 110
   Histogram Persistency Services 80
   Incident service 118
   Job Options service 97
   Message Service 104
   N-tuples Service 83
   Particle Properties Service 107
   Random numbers service 115
   requesting and accessing 95
   ToolSvc 123, 129
   vs. Tools 123
SmartDataLocator 51
SmartDataPtr 51
SmartRef 52
StatusCode 27
T
Tools 123
   Associators 131
   provided in Gaudi 131
   vs. Services 123
ToolSvc, see Services