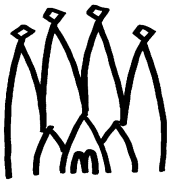

Gaudi

Data Processing Applications Framework

Developers Guide

Corresponding to Gaudi release v9

Version: 2
Issue: 0
Edition: 0
Status:
ID:
Date: 20 December 2001

European Laboratory for Particle Physics
Laboratoire Européen pour la Physique des Particules
CH-1211 Genève 23 - Suisse

Document Control Sheet

Document	**Title:**	Gaudi Developers Guide		
	Version:	2		
	Issue:	0		
	Edition:	0		
	ID:			
	Date:	20 December 2001		
	Keywords:			
Tools	**DTP System:**	Adobe FrameMaker	**Version:**	5.5
Authorship	**Coordinator:**	M.Cattaneo - marco.cattaneo@cern.ch		

Conditional text tags

The following table lists the conditional text tags defined for this document, and their recommended settings for different types of books

Tag / *default colour*	**Recommended setting**		
	Gaudi Developers Guide	**Athena Developers Guide**	**Gaudi Users Guide for LHCb**
Atlas / *Red*	not set	set / *Black*	not set
LHCb / *Blue*	not set	not set	set / *Black*
NotAtlas / *Black*	set	not set	set
NotLHCb / *Black*	set	set	not set

Document Status Sheet

Title:	Gaudi Developers Guide		
ID:			
Version	**Issue**	**Date**	**Reason for change**
1	0	18/Apr/2001	First version, based on LHCb Gaudi Users Guide version 6.3, including changes for Gaudi release 7
2	0	20/Dec/2001	Corresponding to Gaudi release 9

Table of Contents

Chapter 1
Introduction

1.1 Purpose of the document

This document is intended as a combination of user guide and tutorial for the use of the Gaudi application framework software. It is complemented principally by two other "documents": the Architecture Design Document (ADD) [1], and the online auto-generated reference documentation [2]. A third document [3] lists the User Requirements and Scenarios that were used as input and validation of the architecture design. All these documents and other information about Gaudi, including this user guide and source code documentation, are available via the Gaudi home page: *http://cern.ch/proj-gaudi*.

The ADD contains a discussion of the architecture of the framework, the major design choices taken in arriving at this architecture and some of the reasons for these choices. It should be of interest to anyone who wishes to write anything other than private analysis code.

As discussed in the ADD the application framework should be usable for implementing the full range of offline computing tasks: the generation of events, simulation of the detector, event reconstruction, testbeam data analysis, detector alignment, visualisation, etc. etc..

In this document we present the main components of the framework which are available in the current release of the software. It is intended to increment the functionality of the software at each release, so this document will also develop as the framework increases in functionality. Having said that, physicist users and developers actually see only a small fraction of the framework code in the form of a number of key interfaces. These interfaces should change very little if at all and the user of the framework cares very little about what goes on in the background.

The document is arranged as follows: Chapter 2 is a short resume of the framework architecture, presented from an "Algorithm-centric" point of view, and re-iterating only a part of what is presented in the ADD.

Chapter 3 contains a summary of the functionality which is present in the current release, and details of how to obtain and install the software.

Chapter 4 discusses in some detail an example which comes with the framework library. It covers the main program, some of the basic aspects of implementing algorithms, the specification of job options and takes a look at how the code is actually executed. The subject of coding algorithms is treated in more depth in Chapter 5.

Chapter 6 discusses the use of the framework data stores and event data. Chapter 7 is a placeholder for describing the experiment specific event data models. Chapters 8, 9, 10 discuss the other types of data accessible via these stores: detector description data (material and geometry), histogram data and n-tuples.

Chapter 11 deals with services available in the framework: job options, messages, particle properties, auditors, chrono & stat, random numbers, incidents, introspection. It also has a section on developing new services. Framework tools are discussed in Chapter 12, the use and implementation of converter classes in Chapter 13.

Chapter 14 discusses scripting as a means of configuring and controlling the application interactively. This is followed by a description in Chapter 15 of how visualisation facilities might be implemented inside Gaudi.

Chapter 16 describes the package structure of Gaudi and discusses the different types of libraries in the distribution.

Chapter 17 gives pointers to the documentation for class libraries which we are recommending to be used within Gaudi.

Appendix A contains a list of references. Appendix B lists the options which may be specified for the standard components available in the current release. Appendix C gives the details of the syntax and possible error messages of the job options compiler. Finally, Appendix D is a small guide to designing classes that are to be used in conjunction with the application framework.

1.2 Conventions

1.2.1 Units

We have decided to adopt the same system of units as CLHEP, as used also by GEANT4. This system is fully documented in the CLHEP web pages, at the URL: *http://wwwinfo.cern.ch/asd/lhc++/clhep/manual/UserGuide/Units/units.html*.

The list of basic units is reproduced in Table 1.1. Note that this differs from the convention used in GEANT 3, where the basic units of length, time and energy are, respectively, centimetre, GeV, second..

Users should not actually need to know what units are used in the internal representation of the data, as long as they are consistent throughout the Gaudi data stores. What they care about is that they can define and plot quantities with the correct units. In some specialised algorithms they may also wish to renormalise the data to a different set of units, if the default set would lead to numerical precision problems.

Table 1.1 CLHEP system of units

Quantity	Unit
Length	millimetre
Time	nanosecond
Energy	MeV
Electric charge	positron charge
Temperature	Kelvin
Amount of substance	mole
Plane angle	radian

We therefore propose the following rules, which are discussed more fully in reference [5].

1. All dimensioned quantities in the Gaudi data stores shall conform to the CLHEP system of units.
2. All definitions of dimensioned quantities shall be dimensioned by multiplying by the units defined in the `CLHEP/Units/SystemOfUnits.h` header file. For example:

```
const double my_height = 170*cm;
const double my_weight = 75*kg;
```

   Note that the user should not care about the numerical value of the numbers `my_height` and `my_weight`. Internally these numbers are represented as 1700. and 4.68e+26. respectively, but the user does not need to know.

3. All output of dimensioned quantities should be converted to the required units by dividing by the units defined in the `CLHEP/Units/SystemOfUnits.h` header file. For example:

```
my_hist = histoSvc()->book( "/stat/diet","corpulence (kg/m**2)",30,10.,40.);
double my_corpulence = my_weight / (my_height*my_height);
my_hist->fill( my_corpulence/(kg/m2), 1. );
```

   which, for a healthy person, should plot a number between 19. and 25....

4. Physical constants should not be defined in user code. They should be taken directly from the `CLHEP/Units/PhysicalConstants.h` header file. For example:

```
float my_rest_energy = my_weight * c_squared;
```

5. Users may wish to use a different set of units for specific purposes (e.g. when the default units may lead to precision problems). In this case algorithms can renormalise their private copy of the data (as shown in the last line of the rule 3 example) for internal use, but making sure that any data subsequently published to the public data stores is converted back to the CLHEP units.

1.2.2 Coding Conventions

The Gaudi software follows (or should follow!) the LHCb C++ coding conventions described in reference [6].

1.2.2.1 File extensions

One consequence of following the LHCb coding conventions is that the specification of the C++ classes is done in two parts: the header or ".h" file and the implementation or ".cpp" file.

We also define file extensions for Gaudi specific files. The recommended file extension for Job Options files is ".opts" (see Section 11.3.3 on page 86). For Python scripts, the extension ".py" is mandatory (see Chapter 14).

1.2.3 Naming Conventions

Histograms In order to avoid clashes in histogram identifiers, we suggest that histograms are placed in named subdirectories of the transient histogram store. The top level subdirectory should be the name of a sub-detector group (e.g. `VELO`). Below this, users are free to define their own structure. One possibility is to group all histograms produced by a given algorithm into a directory named after the algorithm.

1.2.4 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 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.

1.3 Reporting problems

Users of the Gaudi software are encouraged to report problems and requests for new features via the LHCb problem reporting system. This system is integrated in the CERN Problem Report Management System (CPRMS) provided by IT division, based on the Action Request System software from Remedy Corporation.

To report a new problem, go to the LHCb CPRMS home page *http://cern.ch/hep-service-prms/lhcb.html*, click on the **Submit** button, and fill in the form. This will add the report to the system and notify the developers by E-mail. You will receive E-mail notification of any changes in the status of your report.

To view the list of current problems, and their status, click the **Query** button on the LHCb CPRMS home page.

Active developers of the Gaudi software are encouraged to use the *gaudi-developers* mailing list for discussion of Gaudi features and future developments. This list is not, primarily, intended for bug reports. In order to send mail to *gaudi-developers@listbox.cern.ch*, you must first subscribe to the list, using the form at *https://wwwlistbox.cern.ch/admin-cgi/listbox-admin?operation=viewlist&mail=gaudi-developers@cern.ch*. You need a CERN mailserver account to be able to use this form...

The archive of the mailing list is publically accessible on the Web, at *http://cern.ch/~majordom/news/gaudi-developers/index.html*.

1.4 Editor’s note

This document is a snapshot of the Gaudi software at the time of the release of version v9. We have made every effort to ensure that the information it contains is correct, but in the event of any discrepancies between this document and information published on the Web, the latter should be regarded as correct, since it is maintained between releases and, in the case of code documentation, it is automatically generated from the code.

We encourage our readers to provide feedback about the structure, contents and correctness of this document and of other Gaudi documentation. Please send your comments to the editor, *Marco.Cattaneo@cern.ch*

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 references [1] and [4].

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.

The main components of the Gaudi software architecture can be seen in the object diagram shown in Figure 2.1. Object diagrams are very illustrative for explaining how a system is decomposed. They represent a hypothetical snapshot of the state of the system, showing the

objects (in our case component instances) and their relationships in terms of ownership and usage. They do not illustrate the structure, i.e. class hierarchy, of the software.

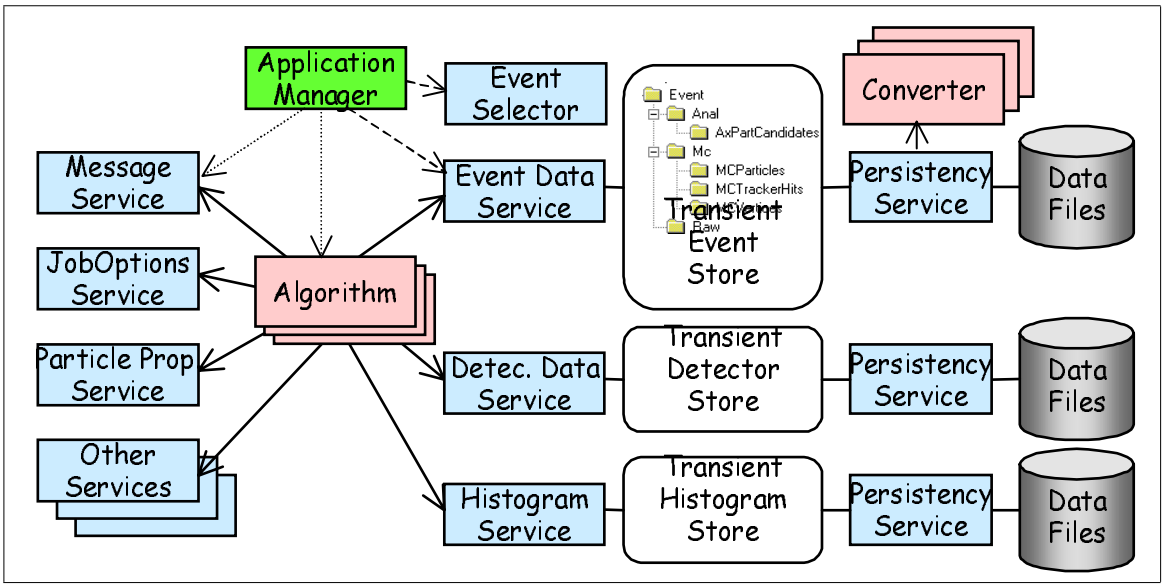

Figure 2.1 Gaudi Architecture Object Diagram

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 6. The third base class, Converter, exists more for technical necessity than anything else and will be discussed in Chapter 13. 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:

```
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:

```
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 2.2 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

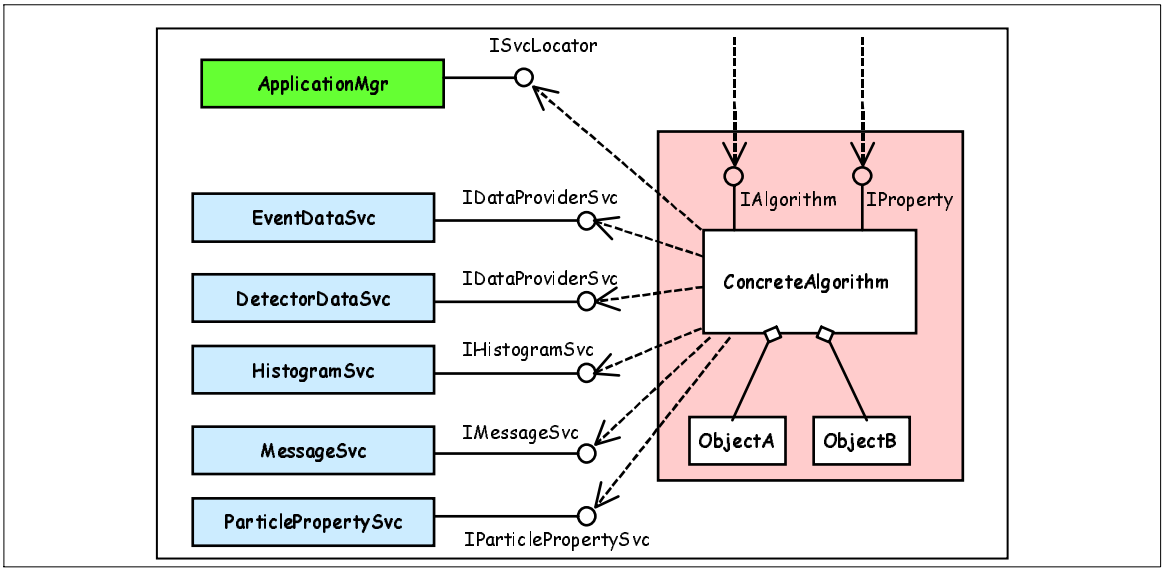

Figure 2.2 The main components of the framework as seen by an algorithm object

The particle property service is an example of additional services that are available to an algorithm. The job options service (see Chapter 11) 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: `IDataManagerSvc` 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 6 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 9. The n-tuple service is similar, with access via the `INtupleSvc` extension to the `IDataProviderSvc` interface, as discussed in Chapter 10.

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 11.

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:

```
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:

```
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`. Interface identifiers are discussed in detail in Chapter 16.

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 “`MessageService`”.

2.5 Controlling and Scheduling

2.5.1 Application Bootstrapping

The application is bootstrapped by creating an instance of the *ApplicationMgr* component. The *ApplicationMgr* is in charge of creating an initializing a minimal set of basic and essential services before control is given to specialized scheduling services. These services are shown in

Figure 2.3. The *EventLoopMgr* is in charge controlling the main physics event[1] loop and scheduling the top algorithms. There will be a number of more or less specialized implementations of EventLoopMgr which will perform the different actions depending on the running environment, and experiment specific policies (clearing stores, saving histograms, etc.). There exists the possibility to give the full control of the application to a component implementing the `IRunable` interface. This is needed for interactive applications such as event display, interactive analysis, etc. The *Runable* component can interact directly with the *EventLoopMgr* for triggering the processing of the next physics event.

The essential services that the *ApplicationMgr* need to instantiate and initialize are the *MessageSvc* and *JobOptionsSvc*.

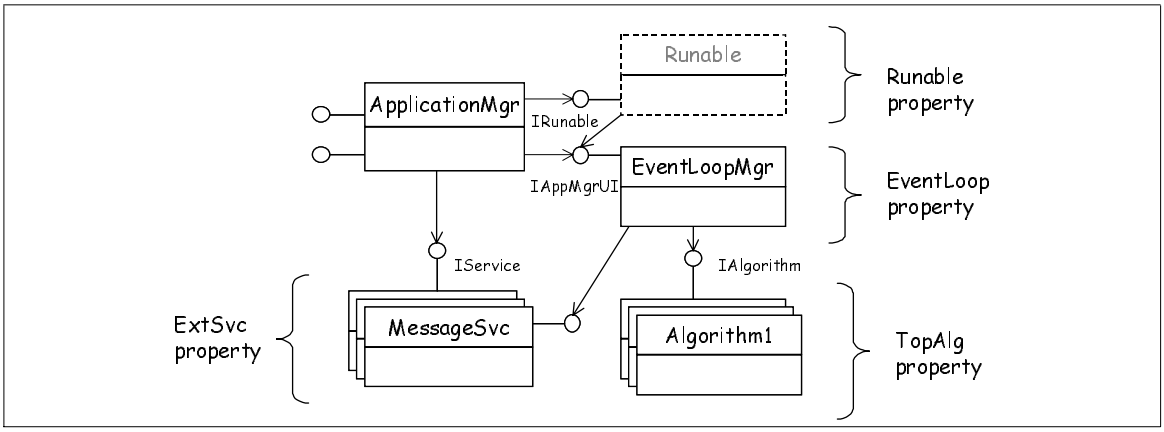

Figure 2.3 Control and Scheduling collaboration

2.5.2 Algorithm Scheduling

The Gaudi architecture foresees explicit invocation of algorithms by the framework or by other algorithms. This latter possibility allows for a hierarchical organization of algorithms, for example, a high level algorithm invoking a number of sub-algorithms.

The *EventLoopMgr* component is in charge of initializing, finalizing and executing the set of Algorithms that have been declared with the `TopAlg` property. These *Algorithms* are executed unconditionally in the order they have been declared. This vary basic scheduling is insufficient for many use cases (event filtering, conditional execution, etc.). Therefore, a number of *Algorithms* have been introduced that perform more sophisticated scheduling and can be configured by some properties. Examples are: Sequencers, Prescalers, etc. and the list can be easily extended. See Section 5.5 for more details on these generic high level Algorithms.

1. We state physics event to differentiate from what is called generally an *event* in computing.

Chapter 3
Release notes and software installation

3.1 Release History

The Gaudi architecture and framework, which was initially developed by the LHCb collaboration, became a joint development project between several experiments, starting from release v6. At this time the package structure was modified, to split the experiment specific packages from the common packages. The following table reflects the version history since the re-packaging. For the history of earlier releases, please refer to previous versions of the (LHCb) Gaudi Users Guide.

Version	Date	Package List
v9	Dec 2001	GaudiPolicy[v5], GaudiExamples[v9], GaudiKernel[v11], GaudiSvc[v7], GaudiAud[v5], GaudiAlg[v5], GaudiTools[v5], GaudiNagC[v6], GaudiDb[v5], GauiRootDb[v5], GaudiODBCDb[v5], HbookCnv[v11], RootHistCnv[v5], GaudiPython[v2], GaudiObjDesc[v2], GaudiIntrospection[v2]
v8r1	July 2001	GaudiPolicy[v5], GaudiExamples[v8], GaudiKernel[v10], GaudiSvc[v6], GaudiAud[v4], GaudiAlg[v4], GaudiTools[v4], GaudiNagC[v5r3p2], GaudiDb[v4], GauiRootDb[v4], GaudiODBCDb[v3], HbookCnv[v10r1], RootHistCnv[v4], SIPython[v2]
v7	23/03/2001	GaudiPolicy[v4], GaudiExamples[v7], GaudiKernel[v9], GaudiSvc[v5], GaudiAud[v3], GaudiAlg[v3], GaudiTools[v3], GaudiNagC[v5r3p1], GaudiDb[v3], HbookCnv[v9], RootHistCnv[v3], SIPython[v1r1]

3.2 Current Functionality

We use an incremental and iterative approach for producing the Gaudi software. We plan to expand its capabilities release by release. The functionality list that follows is organized by categories.

Interfaces A set of interfaces that facilitates the interaction between the different components of the framework. Mainly these are interfaces to services.

Basic framework services This set of services offer the minimal functionality needed for constructing applications. They are described in detail in Chapter 11.
The *message service* is used to send and format messages generated in the code, with an associated severity that is used for filtering and dispatching them.
The *job options service* allows the configuration of the application by end users assigning values to properties defined within the code; properties can be basic types (`float, bool, int, string`), or extended with bounds checking, hierarchical lists, and immediate callback from string "commands".
The *Random Numbers service* makes available several random number distributions via a standard interface, and ensures that applications use a unique random number engine in a reproducible fashion.
The *Chrono service* offers the functionality for measuring elapsed time and job execution statistics.
Auditors and *AuditorSvc* provide monitoring of various characteristics of the execution of Algorithms. Auditors are called before and after invocation of any Algorithm method.
The *Incident service* provides a synchronization between objects within the Application by using named incidents that are communicated to *listener* clients.
The *Tools service*, which provides management of Tools, is discussed in Chapter 12. Tools are lightweight objects which can be requested and used many times by other components to perform well defined tasks. A base class for *associator* tools has been added in this release.
Data services provide the access to the transient data objects (event, detector and statistical data). The data services are described in chapters 6 to 10. The basic building blocks for the implementation of the experiment specific data models are also described in Chapter 6.

Event data persistent storage The current version provides a set of generic classes for implementing event data persistency (*GaudiDb* package) and a set of classes supporting persistent I/O to ROOT files (*GaudiRootDb* package). Details can be found in Chapter 13.

Histograms & N-tuples The framework provides facilities for creating histograms (1 and 2 dimensional) and n-tuples (row and column wise) from user algorithms. The histogram interface is the AIDA[10] common interface. Saving histograms and n-tuples is currently implemented using the HBOOK and ROOT format. The interface to histograms and n-tuples from the user code should not be affected if the persistency representation is changed later. Details of the histogram and n-tuple facilities can be found in Chapter 9 and 10 respectively.

Event tag collections The framework provides facilities for creating and using collections of event tags (based on an n-tuples implementation) for fast direct access to physics events. The user can specify an event tag collection as input data to an application and perform sophisticated selections using the facilities existing in the data storage technology. This is explained in Chapter 10.

Detector description and geometry The framework provides facilities for accessing detector description and geometry data. This is described in Chapter 8. A concrete implementation exists in LHCb, but is not distributed with Gaudi.

Analysis services A number of facilities and services are included in the current release to facilitate writing physics analysis code. The *GaudiAlg* package is a collection of general purpose algorithms, including a sequencer which uses the filtering capability of algorithms to manage the execution of algorithm sequences in a filtering application (see Section 5.5). The *Particle Properties service* (Section 11.5) provides the properties of all the elementary particles. Numerical utilities are available via the CLHEP and NAG C libraries (Chapter 17).

Visualization services The framework provides a mechanism for the visualisation of event and detector data. A prototype implementation exists in LHCb, but is not distributed with Gaudi. This is briefly described in Chapter 15.

Object Description and Object Introspection The framework provides object modelling and description using XML files. Two code generation back-ends are currently available: to generate the data object header files and to generate the object dictionaries for the object introspection. Refer to

Scripting services The framework provides a service for interfacing Python with a Gaudi application. The user can interact with a Gaudi application from the Python prompt. The current functionality allows the user to set and get properties from Algorithms and Services, interact with the data stores (event, detector and histogram) using the object introspection capability, and to schedule the execution of the application's algorithms. Refer to .

Dynamic loading of libraries The framework can be used to implement different data processing applications for different environments. It is important that the services and their concrete implementations are extendable and configurable dynamically at run time and not statically. The latter would imply linking with all the available libraries producing huge executables. And in certain applications and environments some of the libraries will surely never be used. The framework provides support for dynamic libraries for the Windows and Linux platforms.

3.3 Changes between releases

3.3.1 Changes between current release (v9) and previous release (v8)

- **Object description and object introspection**. Two new packages has been added that provide the object description based on XML files and run-time object introspection capability. Refer to Section 6.7 and Section 11.10 for more details.
- **Python service**. The scripting service based on Python has been re-implemented using the Boost library[1]. Its functionality has been extended. Refer to Chapter 14 for more details.
- **Algorithms.** Added toolSvc() accessor to Algorithm base class.
- **Algorithm Tools**. Added `initialize()` and `finalize()` methods in `IAlgTool` interface. The base class `AlgTool` implements them as dummy but allows an implementation of them on specific Algorithm Tools. Removed the need to implement a `queryInterface()` in specific Algorithm tools. Instead use the expression `declareInterface<Ixxxx>(this)` in the constructor.
- A number of small internal framework improvements:
    - ApplicationMgr. Re-organization to relocate the management of services to `ServiceManager` class. The interfaces `ISvcManager` and `ISvcLocator` have changed.

1. http://www.boost.org/libs/libraries.htm

- Introduced a new constructor for `InterfaceID` that uses a name (class name) instead of an interface number.
- JobOptions. Introduced new options `#pragma print on`, `#pragma print off` to switch the printing of job options on and off.
- Histograms. New job option `HistogramPersistencySvc.PrintHistos` to steer printing to standard output. Allow RZ directory names up to 16 characters rather than 8.

3.3.1.1 Incompatible changes

In this section we will list changes that users need to make to their code in order to upgrade to the current version of Gaudi from the previous version.

1. In the area of *Data Stores* many low level base classes (`DataObject`, `DataSvc`, `Converters`, `Registry`, `GenericAddress`, etc.) have changed together with some basic interfaces (`IConverter`, `IDataManagerSvc`, `IDataProviderSvc`, etc.). This implies that some packages, typically converters packages, will need deep changes in the code. Instructions on how to upgrade them can be found in *http://cern.ch/lhcb-comp/Frameworks/Gaudi/Gaudi_v9/Changes_cookbook.pdf*.
   End user algorithm packages should not be too affected by these changes.
2. Removed the list of default interfaces in `ApplicationMgr`. Services are late created if needed. This may cause problems if the order of creation played a role. The Algorithms and Tools that were accessing services using the call `serviceLocator()->service("name", interface)` may require to force the creation of a previously default service by adding a third argument with `true` to force such creation if not existing.
3. The constant `CLID_Event` has been removed from `ClassID.h`. It needs to be defined now in the `Event.h` header file.
4. Algorithm tools are required to implement an interface (pure abstract base class) using the facility provided for declaring it as mentioned above in the list of changes.

3.3.2 Changes between release v8 and release v7

3.3.2.1 Incompatible changes

In this section we will list changes that users need to make to their code in order to upgrade to version v8 of Gaudi from version v7.

1. Location of Histogram Interfaces. Gaudi version v8 uses the standard AIDA interfaces for histograms. These interfaces are located in the AIDA project area. The changes to the end-user code is that the include file should be prefixed with `AIDA/` instead of the current `GaudiKernel/`.
2. Persistent representation of N-tuples. N-tuples saved in HBOOK format no longer have type information in the first row. See the discussion in Section 10.2.3.2 for more details.

3. The output of N-tuples to ODBC () is no longer supported. N-Tuple preselections based on SQL or interpreted C++ are no longer available. If you rely on these features, please contact the Gaudi development team.
4. When saving data objects in a data store, all the sub-directory nodes in the path must already exist or should be explicitly created. In fact this is not a new feature, but a bug fix! Implicit creation of sub-directory nodes will be implemented in a future version.

3.3.3 Changes between release v7 and release v6

- The control of the "physics event" loop has been separated from the `ApplicationMgr` and has become a new component, the event loop manager. A number of subsequent specializations have been provided: `MininalEventLoopMgr`, `EventLoopMgr`, and `GaudiEventLoopMgr`. These changes have been made to allow the possibility to have other types of event loop processing. These changes are backward compatible.
- The first version of a scripting service based on Python has been released.
- A number of small internal framework improvements:
    - Elimination of the up to now required static libraries.
    - Added version number (major and minor) to the Interface ID to check for interface compatibility at run-time.
    - Re-shuffling of the System class and conversion to a namespace.
    - Handling empty vectors in JobOptions.

3.3.3.1 Incompatible changes

In this section we will list changes that users need to make to their code in order to upgrade to version v7 of Gaudi from version v6.

1. Histogram persistency. In previous versions, the HBOOK histogram persistency service was created by default. From this version there is no default histogram persistency: the ROOT or HBOOK persistency services have to be explicitly declared. See Section 9.4 for details
2. `EvtMax`. In previous versions it was possible to declare the number of events to be processed through either of the properties `ApplicationMgr.EvtMax` or `EventSelector.EvtMax`. In this release, only `ApplicationMgr.EvtMax` is supported, the default being all the events in the input file
3. The property `EventSelector.JobInput` has been removed. Use `EventSelector.Input` instead (note the change in format of the value string).

3.3.4 Deprecated features

We list here features of the framework which have become obsolete, because they have been superseded by more recent features. Users are discouraged from using the deprecated features, which may be removed in a future release.

Adding indexed items to N-tuples Use the function `addIndexedItem` instead of `addItem`.

Accessors names in Algorithm Use the service accessors with short names (e.g. `msgSvc()`) instead of the long equivalent ones (e.g. `messageService()`)

Access to extra services from Algorithms. Use the templated method `service()` instead of using the `serviceLocator()` together with the `queryInterface()` to get a reference to a service interface.

User Parameters in detector elements. The XML tag for user parameters in the detector description (detector elements, etc.) is now `<param/>` instead of `<userparameter/>`. The old name will be maintained for a while. The methods in `DetectorElement` and `Condition` classes will accordonly be changed to use the word `param` instead of `userParameter`.

3.4 Availability

The application framework is supported on the following platforms:

- Windows NT4 and Windows 2000, using the Developer Studio 6.0 SP2 Visual C++ environment
- RedHat Linux 6.1 (certified CERN Linux distribution with SUE and AFS) with egcs-2.91.66 and gcc-2.95.2.

The code, documentation and installation instructions are available from the Gaudi web site at: *http://cern.ch/proj-gaudi/*.

Framework sources and binaries are also available in the CERN AFS cell, at */afs/cern.ch/sw/Gaudi* .

3.5 Using the framework

3.5.1 CVS repository

The framework sources are stored in CVS and can be accessed using the CVS server. You have to specify the following option in your CVS command:

```
-d :pserver:cerncvs@lhcbcvs.cern.ch:/local/gaudicvs
```

You have to login to the CVS server first:

```
cvs -d :pserver:cerncvs@lhcbcvs.cern.ch:/local/gaudicvs login
```

The server will ask for a password, reply `CERNuser`. You can now send all the CVS commands that don't require write access. If you use a command like **commit**, you will get an error message. When you have finished, you can logout of the server.

3.5.2 CMT

The framework libraries have been built using the Configuration Management Tool (CMT) [7]. Therefore, using the CMT tool is the recommended way to modify existing packages or re-build the examples included in the release. If CMT is not available in your system, please follow the installation instructions in [7]. The following simple examples are for Unix, but similar commands exist also for Windows. They assume that the `CMTPATH` environment variable is set to `$HOME/mycmt:$GAUDIHOME`.

Getting a copy of a package: Suppose you want to build the latest released version of the `GaudiExamples` package:

```
  1:  cd mycmt
  2:  cmt checkout GaudiExamples -r v7
```

Building and running an example: Now that you have the code, suppose you want to modify the `AlgSequencer` example, then build it and run it:

```
  3:  cd GaudiExamples/v7/src/AlgSequencer
  4:  emacs HelloWorld.cpp
----- Make your modification, then
  5:  cd ../../cmt
  6:  source setup.csh
  7:  emacs requirements
----- Uncomment AlgSequencer and comment all the others
  8:  gmake
  9:  cd ../home
 10:  emacs AlgSequencer.txt
----- Make any modification if needed
 11:  ../$CMTCONFIG/AlgSequencer.exe
```

Modifying a library and rerunning the example: Suppose now you want to modify one of the Gaudi libraries, build it, then rerun the `AlgSequencer` example with it:

```
 12:  cd $HOME/mycmt
 13:  cmt checkout GaudiAlg v3
 14:  cd GaudiAlg/v3/src
 15:  emacs ...
---- Make your modification...
 16:  cd ../cmt
 17:  source setup.csh
 18:  gmake
 19:  cd $HOME/mycmt/GaudiExamples/v7/cmt
 20:  cmt show uses
 21:  ---- Verify the you are now using the GaudiAlg version from $HOME/mycmt
 22:  ---- There is no need to relink, since GaudiAlg is a component library
 23:  cd ../home
 24:  ../$CMTCONFIG/AlgSequencer.exe
```

3.5.3 Using the framework on Windows with Developer Studio or Nmake

The libraries for Windows are available for download from the web, and in AFS in the *Win32Debug* subdirectory of each package.

Instructions for installing the Gaudi environment on Windows and for customising MS Visual Studio will be made available here in due course. For now, please refer to the LHCb specific instructions at: *http://cern.ch/lhcb-comp/Support/html/DevStudio/default.htm*

The `requirements` files and CMT commands expect the following environment variables:

- **HOME** Needs to be set to the user's home directory. Typically this is in a network server and will have the form "\\server\username" or can also be a local directory like "C:\home". This environment variable is used to locate the .cmtrc file that contains the default CMTPATH.
- **PATH** Should be set up correctly to locate the Developer Studio executables (this is typically the case after installation).
- **TEMP** Location for temporary files. This is set correctly after the operating system installation.
- **SITEROOT** This is the root where software is installed. Typically it will point to some share in some server (\\server\siteroot) or to the locally mounted AFS drive (F:\cern.ch).
- **CMTPATH** The first location where CMT is looking for packages. This is typically the local directory C:/mycmt
- **CMTSITE** This is your site name. At CERN site and for the Windows platform we use CERN_WIN32.

3.5.4 Using the framework in Unix

The libraries for Linux are available for download from the web, and in AFS in the *i386_linux22* and *Linuxdbx* subdirectories of each package (for the optimised and debug versions respectively).

Instructions for installing the Gaudi environment on Linux will be made available here in due course. For now, please refer to the LHCb specific instructions at:
http://cern.ch/lhcb-comp/Support/html/start_gaudi.htm

3.6 Working with development releases

This User Guide corresponds to release v9 of the Gaudi software.

For Gaudi packages, before they are publicly released and frozen, the development versions are periodically rebuilt from the head revision of the CVS repository in the *development release area*. These versions are not guaranteed to work and may change without notice; they are intended for integration tests. They should be used with care, mainly if you wish to use new features of the software which have not yet been incorporated in a public release.

3.7 Installation of the framework outside CERN

3.7.1 Package installation

To use the Gaudi framework you also need to have access to installations of some external packages, listed below:

CMT, CLHEP, NAG C, HTL, Python, Xerces, qqlhcb, ROOT, BOOST and CERNLIB.

Up to date instructions for installation of these packages and setting of the environment (variables, path,..) needed to use the framework can be found on the Web at *http://cern.ch/lhcb-comp/Support/html/Install.htm*.

Chapter 4
Getting started

4.1 Overview

In this chapter we walk through one of the example applications (`RandomNumber`) which are distributed with the framework. We look briefly at the different files and go over the steps needed to compile and execute the code. We also outline where various subjects are covered in more detail in the remainder of the document. Finally we cover briefly the other example applications which are distributed and say a few words on what each one is intended to demonstrate.

4.2 Creating a job

Traditionally, a “job” is the running of a program on a specified set of input data to produce a set of output data, usually in batch.

For the example applications supplied this is essentially a two step process. First the executable must be produced, and secondly the necessary environment variables must be set and the required job options specified, as illustrated in Figure 4.1.

The example applications consist of a number of “source code” files which together allow you to generate an executable program. These are:

- The main program.
- Header and implementation files for each of the concrete algorithm classes.
- A CMT requirements file.
- The set of Gaudi libraries.

In order for the job to run as desired you must provide the correct configuration information for the executable. This is done via entries in the job options file.

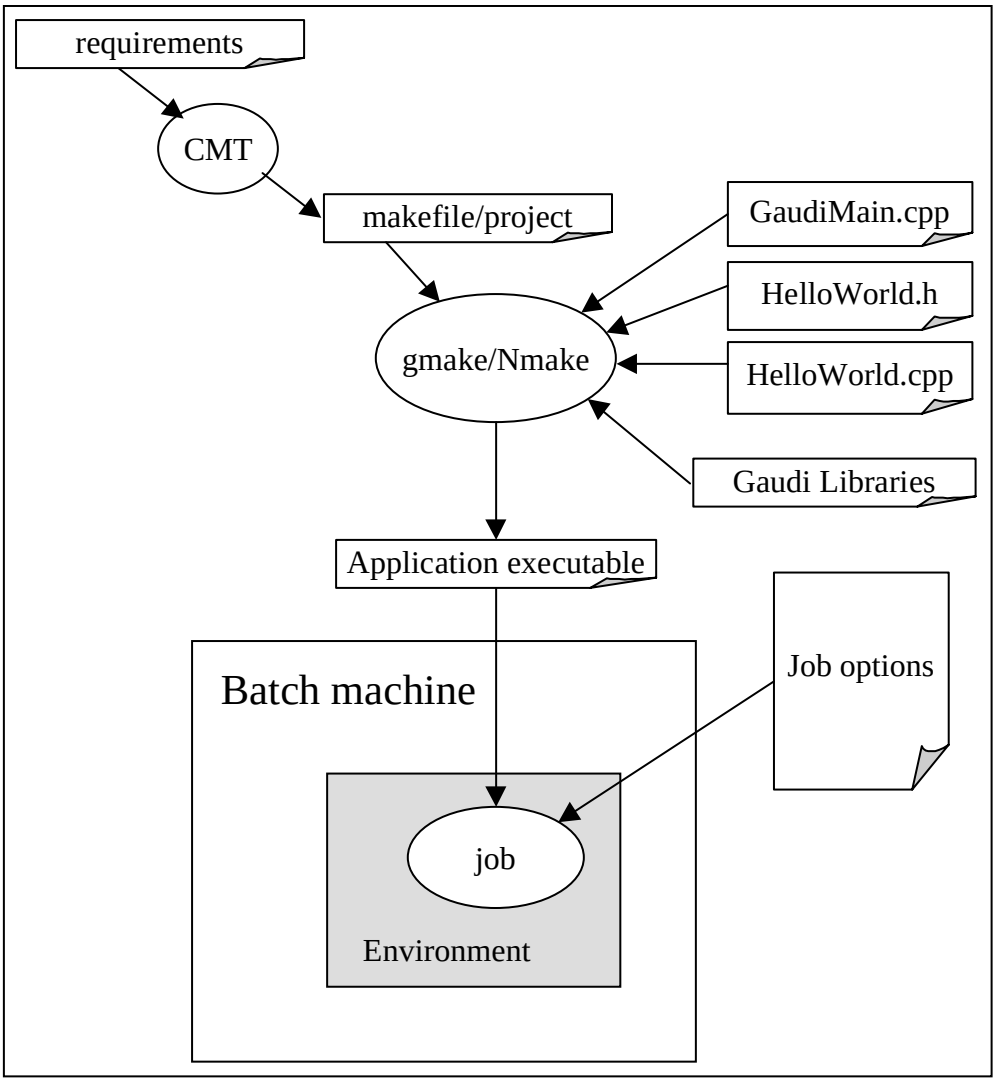

Figure 4.1 Creating a job from the `AlgSequencer` example application

4.3 The main program

The main program is needed to bootstrap the job. It can be completely general, and can be reused by all Gaudi applications. An example main program, from the package `GaudiExamples`, is shown in Listing 4.1. It is constructed as follows:

Include files These are needed for the creation of the application manager and Smart interface pointers.

Application Manager instantiation Line 12 instantiates an `ApplicationMgr` object. The application manager is essentially the job controller. It is responsible for creating and correctly initialising all of the services and algorithms required, for looping over the input data events and executing the algorithms specified in the job options file, and for terminating the job cleanly.

Listing 4.1 The example main program.

```
 1:  // Include files
 2:  #include "GaudiKernel/SmartIF.h"
 3:  #include "GaudiKernel/Bootstrap.h"
 4:  #include "GaudiKernel/IAppMgrUI.h"
 5:  #include "GaudiKernel/IProperty.h"
 6:  #include <iostream>
 7:
 8:  //--- Example main program
 9:  int main(int argc, char** argv) {
10:
11:  // Create an instance of an application manager
12:    IInterface* iface = Gaudi::createApplicationMgr();
13:
14:    SmartIF<IProperty>     propMgr ( IID_IProperty, iface );
15:    SmartIF<IAppMgrUI>     appMgr  ( IID_IAppMgrUI, iface );
16:
17:    if( !appMgr.isValid() || !propMgr.isValid() ) {
18:      std::cout << "Fatal error creating ApplicationMgr " << std::endl;
19:      return 1;
20:    }
21:
22:    // Get the input configuration file from arguments
23:    std:: string opts = (argc>1) ? argv[1] :  "../options/job.opts";
24:
25:    propMgr->setProperty( "JobOptionsPath", opts );
26:
27:    // Run the application manager and process events
28:    appMgr->run();
29:
30:    // All done - exit
31:    return 0;
32:  }
```

Retrieval of Interface pointers The code on lines 14 and 15 retrieves the pointers to the `IProperty` and `IAppMgrUI` interfaces of the application manager.

Setting the application manager's properties The only property which needs to be set explicitly in the main program is the name of the job options file which contains all of the other configuration information needed to run the job. In this example, the name is the first argument of the program and defaults to "`../options/job.opts`" (line 23); it is set on line 25.

Program execution All of the code before line 28 is essentially for setting up the job. Once this is done, a call to `appMgr::run()` is all that is needed to start the job proper! The steps that occur within this method are discussed briefly in section 4.6.

4.4 Configuring the job

The application framework makes use of a job options file for job configuration. Part of the job options file of an example application is shown in Listing 4.2.

Listing 4.2 Part of the job options file for the `RootIORead` example application.

```
 1:  // Include standard option files
 2:  #include "$STDOPTS/Common.opts"
 3:
 4:  // Private Application Configuration options
 5:  ApplicationMgr.DLLs       += { "GaudiDb", "GaudiRootDb" };
 6:  ApplicationMgr.ExtSvc     += { "DbEventCnvSvc/RootEvtCnvSvc" };
 7:  ApplicationMgr.TopAlg      = { "ReadAlg" };
 8:
 9:  // Set output level threshold (2=DEBUG,3=INFO,4=WARNING,5=ERROR,6=FATAL)
10:  MessageSvc.OutputLevel     = 4;
11:  EventSelector.OutputLevel  = 2;
12:
13:  // Input File
14:  EventSelector.Input = {"DATAFILE='RootDst.root' TYP='ROOT' OPT='READ'"};
15:  EventSelector.FirstEvent   = 1;
16:  ApplicationMgr.EvtMax      = 5;
17:
18:  // Persistency service setup:
19:  EventPersistencySvc.CnvServices += { "RootEvtCnvSvc" };
20:
21:  // Setup for ROOT I/O system
22:  RootEvtCnvSvc.DbType  = "ROOT";
```

The format of an options file is discussed fully in Chapter 11. Options may be set both for algorithms and services and the list of available options for standard components is given in Appendix B. Here we look briefly at a few of the more commonly used options.

4.4.1 Defining the algorithms to be executed

The option `ApplicationMgr.TopAlg` (line 7) is a list of algorithms that will be created and controlled directly by the application manager, the so-called top-level algorithms. The syntax is a list of the form:

```
ApplicationMgr.TopAlg = { "Type1/Name1", "Type2/Name2" };
```

The line above instructs the application manager to create two top level algorithms. One of type `Type1` called "`Name1`" and one of type `Type2` called "`Name2`".

In the case where the name of the algorithm is the same as the algorithm's type (i.e. class), only the class name is necessary. In the example, an instance of the class `"ReadAlg"` will be created with name `"ReadAlg"`.

4.4.2 Defining the job input

Event data input is controlled by an `EventSelector`. The `EventSelector` uses a storage technology dependent data persistency service to load the data into the transient event data store, with the help of *converters* which are able to convert the data from the technology

dependent persistent representation, to the technology independent representation in the transient data store.

In order to set up this mechanism, one needs a number of job options:

- Line 14 defines the input data file, and the persistency technology (ROOT I/O in this example).
- Line 6 tells the application manager to create a new event conversion service, to be called `RootEvtCnvSvc`. Note that this is just a name for our convenience, the service is of type `DbEventCnvSvc` and does not (yet) know that it will deal with ROOT technology. The configuration of `RootEvtCnvSvc` to use the ROOT I/O technology is done in line 22.
- Line 19 tells the event persistency service (`EventPersistencySvc` created by the application manager by default) to use the `RootEvtCnvSvc` to do the conversion between persistent and transient data representations.
- Line 5 tells the application manager which additional libraries to load in order to find the required conversion service. In this example, the `GaudiDb` library contains the `DbEventCnvSvc` class, the `GaudiRootDb` library contains the ROOT specific database drivers.
- Finally, the options on lines 15 and 16 tell the `EventSelector` to start reading sequentially from the first event in the file, for five events.

In the special case where no event input is required (e.g. for event generation), one can replace the above options by the two options:

```
ApplicationMgr.EvtMax = 20;   // events to be processed (default is 10)
ApplicationMgr.EvtSel = "NONE"; // do not use any event input
```

A discussion of event I/O can be found in Chapter 10. Converters and the conversion process are described in Chapter 13.

4.4.3 Defining job output

One can consider three types of job output: event data (including event collections and n-tuples), statistical data (histograms) and printout. Here we discuss only the simplest (printout); histograms are discussed in Chapter 9, event data in Section 6.10.1, event collections in Section 10.3.1.

Printout in Gaudi is handled by the message service (described in Chapter 11), which allows to control the amount of printout according to severity level. The global threshold for printout is set by the option on line 10 - in this example only messages of severity level `WARNING` or above will be printed. This can be over-ridden for individual algorithms or services, as in line 11, where the threshold for `EventSelector` is set to `DEBUG`.

4.5 Algorithms

The subject of specialising the Algorithm base class to do something useful will be covered in detail in Chapter 5. Here we will limit ourselves to looking at an example `HelloWorld` class.

4.5.1 The `HelloWorld.h` header file

The `HelloWorld` class definition is shown in Listing 4.3.

Listing 4.3 The header file of the class: `HelloWorld`.

```
 1: // Include files
 2: #include "GaudiKernel/Algorithm.h"  // Required for inheritance
 3: #include "GaudiKernel/Property.h"
 4: #include "GaudiKernel/MsgStream.h"
 5:
 6: class HelloWorld : public Algorithm {
 7: public:
 8:   /// Constructor of this form must be provided
 9:   HelloWorld(const std::string& name, ISvcLocator* pSvcLocator);
10:
11:   /// Three mandatory member functions of any algorithm
12:   StatusCode initialize();
13:   StatusCode execute();
14:   StatusCode finalize();
15: private:
16:   /// These data members are used in the execution of this algorithm
17:   /// and are set in the initialisation phase by the job options service
18:   int         m_int;
19:   double      m_double;
20:   std::string m_string;
21: };
```

Note the following:

- The class is derived from the Algorithm base class as must be all specialised algorithm classes. This implies that the `Algorithm.h` file must be included (line 6).
- All derived algorithm classes must provide a constructor with the parameters shown in line 9. The first parameter is the name of the algorithm and is used amongst other things to locate any options that may have been specified in the job options file.
- The `HistoAlgorithm` class has three (private) data members, defined in lines 18 to 20. These are properties that can be set via the job options file.
- The three methods on lines 12 to 14 must be implemented, since they are pure virtual in the base class.

4.5.2 The `HelloWorld` implementation file

The implementation file contains the actual code for the constructor and for the methods: `initialize()`, `execute()` and `finalize()`. It also contains two lines of code for the `HelloWorld` factory, which we will discuss in section 5.3.1

The constructor must call the base class constructor, passing on its two arguments. As usual, member variables should be initialised. Here we declare and initialise the member variables that we wish to be set by the job options service. This is done by calling the `declareProperty()` method.

```
1:  HelloWorld::HelloWorld(const std::string& name, ISvcLocator* ploc)
2:              : Algorithm(name, ploc) {
3:  //----------------------------------------------------------------------
4:    // Declare the algorithm's properties
5:    declareProperty( "Int",    m_int    = 100 );
6:    declareProperty( "Double", m_double = 100.);
7:    declareProperty( "String", m_string = std::string("one hundred"));
8:  }
```

Initialisation The application manager invokes the `sysInitialize()` method of the algorithm base class which, in turn, invokes the `initialize()` method of the base class, the `setProperties()` method, and finally the `initialize()` method of the concrete algorithm class. As a consequence all of an algorithm's properties will have been set before its `initialize()` method is invoked, and all of the standard services such as the message service are available. This is discussed in more detail in Chapter 5.

Looking at the code in the example (Listing 4.4) we see that we are now able to print out the values of the algorithm's properties, using the message service and the `MsgStream` utility class. A local `MsgStream` object is created (line 3), which uses the Algorithm's standard message service via the `msgSvc()` accessor, and the algorithm's name via the `name()` accessor. The use of these is discussed in more detail in Chapter 11.

Note that the job will stop if the `initialize()` method of any algorithm does not return `StatusCode::SUCCESS`. This is to avoid processing with a badly configured application.:

Listing 4.4 Example of initialize() method

```
 1:  StatusCode HelloWorld::initialize() {
 2:  //----------------------------------------------------------------------
 3:    MsgStream log(msgSvc(), name());
 4:    log << MSG::INFO << "initializing...." << endreq;
 5:    log << MSG::INFO << "Property Int    = " << m_int << endreq;
 6:    log << MSG::INFO << "Property Double = " << m_double << endreq;
 7:    log << MSG::INFO << "Property String = " << m_string << endreq;
 8:
 9:    m_initialized = true;
10:    return StatusCode::SUCCESS;
11:  }
```


execution The `execute()` method is called by the application manager once for every event. This is where most of the real action should take place. The trivial `HelloWorld` class just prints out a message... Note that the method must return `StatusCode::SUCCESS` on successful completion. If a particular algorithm returns a FAILURE status code more than a (configurable) maximum number of times, the application manager will decide that this algorithm is badly configured and jump to the finalisation stage before all events have been processed. .

```
1:  StatusCode HelloWorld::execute() {
2:  //--------------------------------------------------------------------
3:    MsgStream         log( msgSvc(), name() );
4:    log << MSG::INFO << "executing...." << endreq;
5:
6:    return StatusCode::SUCCESS;
7:  }
```

Finalisation The `finalize()` method is called at the end of the job. In this trivial example a message is printed. .

```
1:  StatusCode HelloWorld::finalize() {
2:  //--------------------------------------------------------------------
3:    MsgStream log(msgSvc(), name());
4:    log << MSG::INFO << "finalizing...." << endreq;
5:
6:    return StatusCode::SUCCESS;
7:  }
```

4.6 Job execution

From the main program and the CMT requirements file we can make an executable, as explained in section 3.5. This executable together with the file of job options form a job which may be submitted for batch or run interactively. Figure 4.2 shows a trace of an example program execution. The diagram is not intended to be complete, merely to illustrate a few of the points mentioned earlier in the chapter.

1. The application manager instantiates the required services and initialises them. The message service is done first to allow the other services to use it, and the job options service is second so that the other services may be configured at run time.
2. The algorithms which have been declared to the application manager within the job options (via the `TopAlg` option) are created. We denote these algorithms “top-level” as they are the only ones controlled directly by the application manager. For illustration purposes we instantiate an `EmptyAlgorithm` and a `HistoAlgorithm`.
3. The top-level algorithms are initialised. Their properties (if they have any) are set and they may make use of the message service. If any algorithm fails to initialise, the job is stopped.

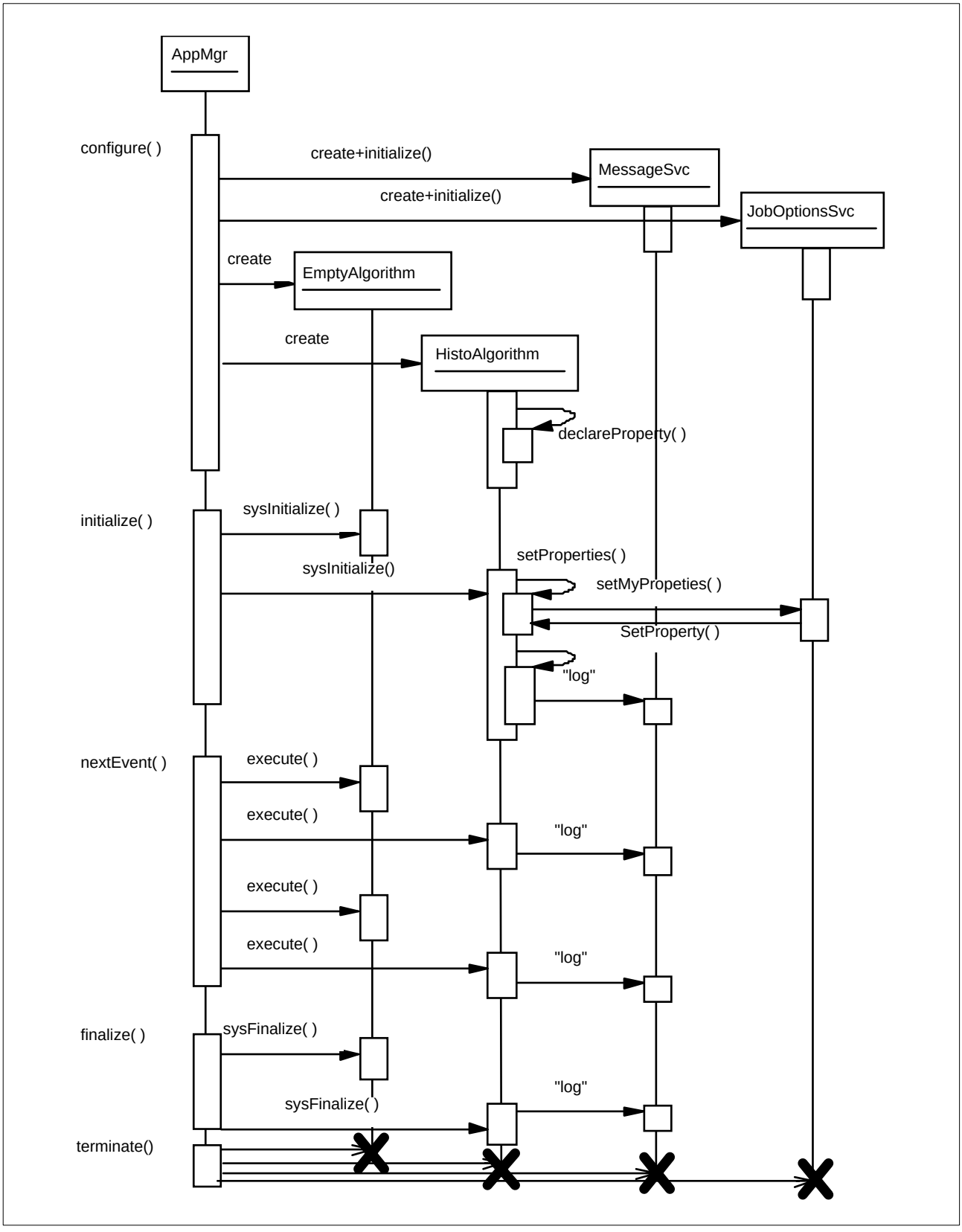

Figure 4.2 A sequence diagram showing a part of the execution of an example program.

4. The application manager now starts to loop over events. After each event is read, it executes each of the top level algorithms in order. The order of execution of the algorithms is the order in which they appear in the `TopAlg` option. This will continue until the required number of events has been processed, unless one or more of the algorithms return a FAILURE status code more than the maximum number of times, in which case the application manager will jump to the finalisation stage before all events have been processed.
5. After the required data sample has been read the application manager finalises each top level algorithm.
6. Services are finalised.
7. All objects are deleted and resources freed. The program terminates.

4.7 Examples distributed with Gaudi

A number of examples is included in the current release of the framework, in the `GaudiExamples` package. The package has some sub-directories in addition to the standard ones shown in Figure 16.2. The `options` sub-directory contains files of standard job options common to many examples. These files are included in the job options of the specific examples when necessary. The specific job options files can be found in the `home` sub-directory.

The code of the examples is in sub-directories of the `src` directory, one sub-directory per example. The intention is that each example demonstrates how to make use of some part of the functionality of the framework. The list of available examples is shown in Table 4.1.

Table 4.1 List of examples available in Gaudi release v9

Example Name	Target Functionality
AlgSequencer	Illustrating the use of the sequencer algorithm provided in the GaudiAlg package
AlgTool	Example implementation and use of a Gaudi Tool
Common	Actually not a complete example: contains the main program used in the examples
GPython	Exercise the Python scripting packages
Properties	Trivial algorithm showing how to set and retrieve Properties
RandomNumber	Example of use of the Random Number service
RootIO	Two examples, reading and writing persistent data with ROOT I/O

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 and protected methods of the `Algorithm` base class. The base class has no protected or public data members, so in fact, these are the only methods that are available. Most of these methods are 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; some 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.

```
 1: class Algorithm : virtual public IAlgorithm,
                      virtual public IProperty {
 2: public:
 3: // Constructor and destructor
 4: Algorithm( const std::string& name, ISvcLocator *svcloc );
 5: virtual ~Algorithm();
 6:
 7: // IAlgorithm interface only partially implemented
 8: StatusCode sysInitialize();
 9: StatusCode sysExecute();
10: StatusCode sysFinalize();
11: StatusCode beginRun();
12: StatusCode endRun();
13: const std::string&  name() const;
14:
15: virtual bool isExecuted() const;
16: virtual StatusCode setExecuted( bool state );
17: virtual StatusCode resetExecuted();
18: virtual bool isEnabled() const;
19: virtual bool filterPassed() const;
20: virtual StatusCode setFilterPassed( bool state );
21:
22: // Service accessors
23: template<class T> StatusCode service( const std::string& name, T*& svc,
    bool createIf = false );
24: void setOutputLevel( int level );
25: IMessageSvc*        msgSvc()         const;
26: IAuditorSvc*        auditorSvc()     const;
27: IDataProviderSvc*   eventSvc()       const;
28: IConversionSvc*     eventCnvSvc()    const;
29: IDataProviderSvc*   detSvc()         const;
30: IConversionSvc*     detCnvSvc()      const;
31: IHistogramSvc*      histoSvc()       const;
32: INtupleSvc*         ntupleSvc()      const;
33: IChronoStatSvc*     chronoSvc()      const;
34: IRndmGenSvc*        randSvc()        const;
35: IToolSvc*           toolSvc()        const;
36: ISvcLocator*        serviceLocator() const;
37:
38: StatusCode createSubAlgorithm( const std::string& type,
                     const std::string& name, Algorithm*& pSubAlg );
39: std::vector<Algorithm*>* subAlgorithms() const;
40:
41: // IProperty interface
42: virtual StatusCode setProperty( const Property& p);
43: virtual StatusCode setProperty( std::istream s& );
44: virtual StatusCode setProperty( const std::string& n,
                                  const std::string& v);
45: virtual StatusCode getProperty( Property* p ) const;
46: const   Property&  getProperty( const std::string& name) const;
47: virtual StatusCode getProperty( const std::string& n,
                                    std::string& v) const;
48: const std::vector<Property*>& getProperties() const;
```


Listing 5.1 The definition of the Algorithm base class.

```
49:  StatusCode setProperties();
50:  template <class T>
       StatusCode declareProperty(const std::string& name, T& property);
51:  StatusCode declareRemoteProperty(const std::string& name,
                IProperty* rsvc, const std::string& rname = "") const;
52:  /// Methods for IInterface
53:  unsigned long addRef();
54:  unsigned long release();
55:  StatusCode queryInterface(const IID& riid, void**);
56:
57:  protected:
58:    bool isInitialized( ) const;
59:    void setInitialized( );
60:    bool isFinalized( ) const;
61:    void setFinalized( );
62:  private:
63:  // Data members not shown
64:  Algorithm(const Algorithm& a);   // NO COPY ALLOWED
65:  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 The base class only partially implements this interface: the three pure virtual methods `initialize()`, `execute()` and `finalize()` must be implemented by a derived algorithm: these are where the algorithm does its useful work and are discussed in more detail in section 5.3. The base class provides default implementations of the methods `beginRun()` and `endRun()`, and the accessor `name()` which returns the algorithm's identifying name. The methods `sysInitialize()`, `sysFinalize()`, `sysExecute()` are used internally by the framework; they are not virtual and may not be overridden.

Service accessor methods Lines 25 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 Section 5.3.3). The services and interface types to which they point are self explanatory. Services may be located by name using the templated `service()` function in line 23 or by using the `serviceLocator()` accessor method on line 36, as described in Section 11.2. Line 24 declares a facility to modify the message output level from within the code (the message service is described in Section 11.4).

Creation of sub algorithms The methods on lines 38 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 A concrete algorithm must declare its properties to the framework using the templated `declareProperty` method (line 50), as discussed in Section 5.3.2 and Section 11.3.1. The Algorithm base class then uses the `setProperties()` method (line 49) to tell the framework to set these properties to the values defined in the job options file. The methods in lines 42 to 48 can later be used to access and modify the values of specific properties, as explained in Section 11.3.2.

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)

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 three lines into the implementation file, of the form:

```
#include "GaudiKernel/AlgFactory.h"
...
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). To declare them as properties it suffices to call the `declareProperty()` method. This method

Listing 5.2 Declaring member variables as properties.

```
 1: //------- In the header file -------------------------------------//
 2: class TriggerDecision : public Algorithm {
 3:
 4: private:
 5:   bool m_passAllMode;
 6:   int m_muonCandidateCut;
 7:   std::vector m_ECALEnergyCuts;
 8: }
 9: //------- In the implementation file ------------------------------//
10: static const AlgFactory<TriggerDecision>  s_factory;
11: const IAlgFactory& TriggerDecisionFactory = s_factory;
12:
13: TriggerDecision::TriggerDecision(std::string name, ISvcLocator *pSL) :
14:   Algorithm(name, pSL), m_passAllMode(false), m_muonCandidateCut(0) {
15:   m_ECALenergyCuts.push_back(0.0);
16:   m_ECALenergyCuts.push_back(0.6);
17:
18:   declareProperty("PassAllMode", m_passAllMode);
19:   declareProperty("MuonCandidateCut", m_muonCandidateCut);
20:   declareProperty("ECALEnergyCuts", m_ECALEnergyCuts);
21: }
22:
23: StatusCode TriggerDecision::initialize() {
24: }
```

is templated 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 Section 11.3 on page 83) 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.

5.3.3 Implementing IAlgorithm

Any concrete algorithm must implement the three pure virtual methods `initialize()`, `execute()` and `finalize()` of the `IAlgorithm` interface. 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 Figure 5.1 shows an example trace of the initialization phase.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 (using the mechanism described in Section 11.3.1 on page 83). At the end, `sysInitialize()` calls the `initialize()` method, which can be used to do such things as creating histograms, or creating sub-algorithms if

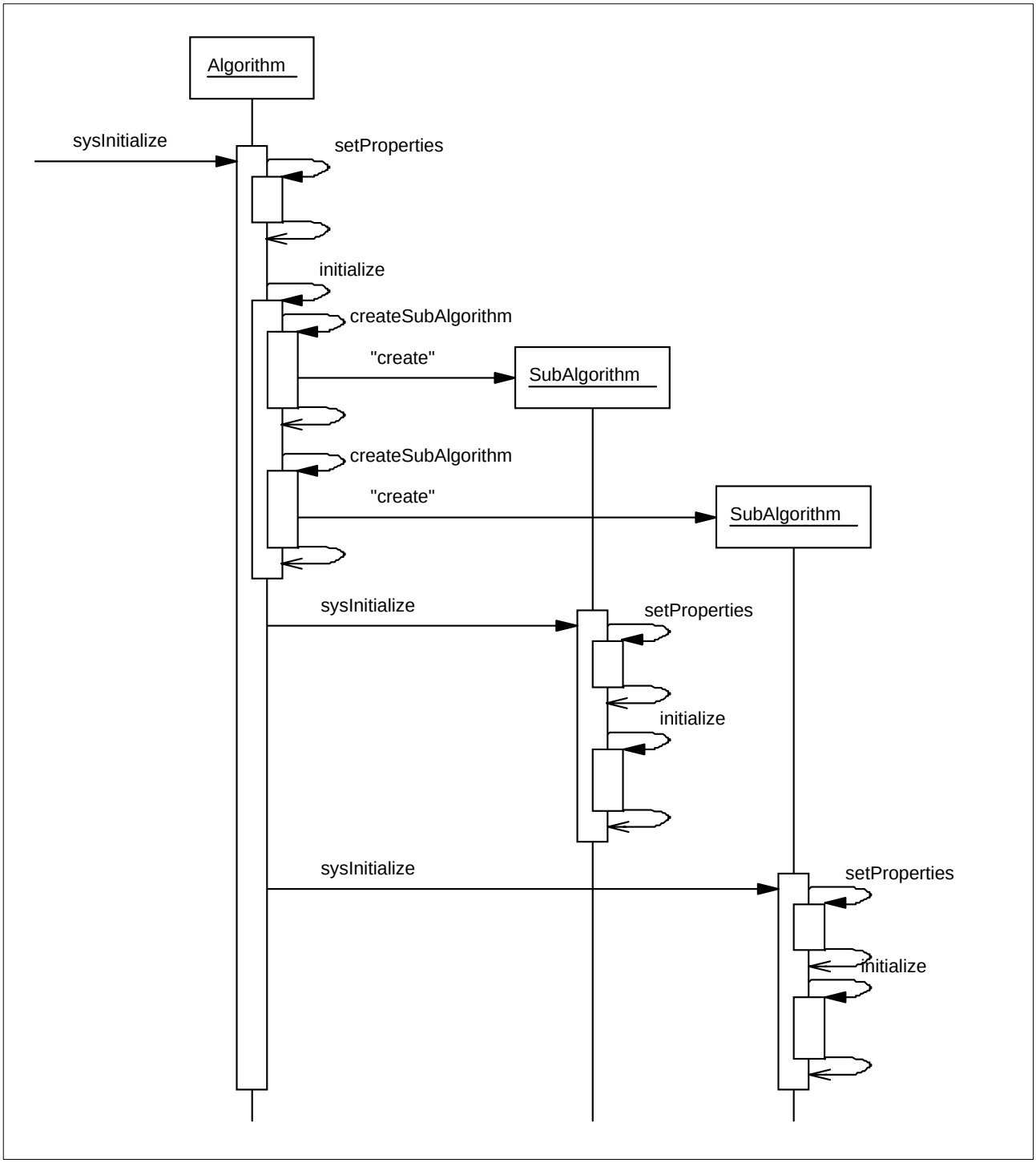

Figure 5.1 Algorithm initialization.

required (sub-algorithms are discussed in Section 5.4). If an algorithm fails to initialize it should return `StatusCode::FAILURE`. This will cause the job to terminate.

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 (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.

Optionally, the methods `beginRun()` and `endRun()` can also be implemented. These are called at the beginning and the end of the event loop respectively.

Monitoring of the execution (*e.g.* cpu usage) of each Algorithm instance is performed by *auditors* under control of the Auditor service (described in Section 11.7 on page 99). 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 Section 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 Gaudi 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
                              // Must be a member variable of the parent class
std::string type;             // Type of sub algorithm
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 may also change the properties of a sub-algorithm during the main event loop as explained in Section 11.3.2. 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 Gaudi 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 Gaudi.

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 a future release of Gaudi that will include an interactive scripting language.

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 5.3 is an extract of the job options file 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 5.3 Example job options using Sequencers demonstrating filtering

```
 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.;
```

5.5.2 Sequence branching

Listing 5.4 illustrates 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 5.4 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 5.5 illustrates the use of inverted logic. It achieves the same goal as the example in Listing 5.4 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 5.5 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"};
 6:  Seq2.Members = {"HelloWorld", "Prescaler:invert",
     "EventCounter/Counter2"};
 7:
 8:  Prescaler.PercentPass = 80.;
```


Chapter 6
Accessing data

6.1 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 `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 13.

6.2 Using the data stores

There are four data stores currently implemented within the Gaudi 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 8, 9 and 10 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 6.1 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.

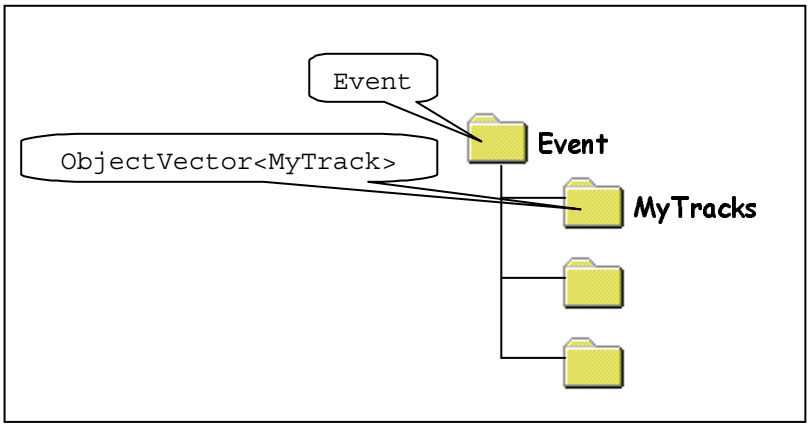

Figure 6.1 The structure the event data model of the RootIO example.

Interactions with the data stores are usually via the IDataProviderSvc interface, whose key methods are shown in Listing 6.1

Listing 6.1 Some of the key methods of the IDataProviderSvc interface.

```
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).

Navigation through the tree stucture of the data store is possible via the IDataManagerSvc interface of the data service, as described for example in *http://cern.ch/lhcb-comp/Frameworks/Gaudi/Gaudi_v9/Changes_cookbook.pdf*.

6.3 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:

```
 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 6.8

The last two methods shown in Listing 6.1 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 6.2 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.

6.4 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 6.2. 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).

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.

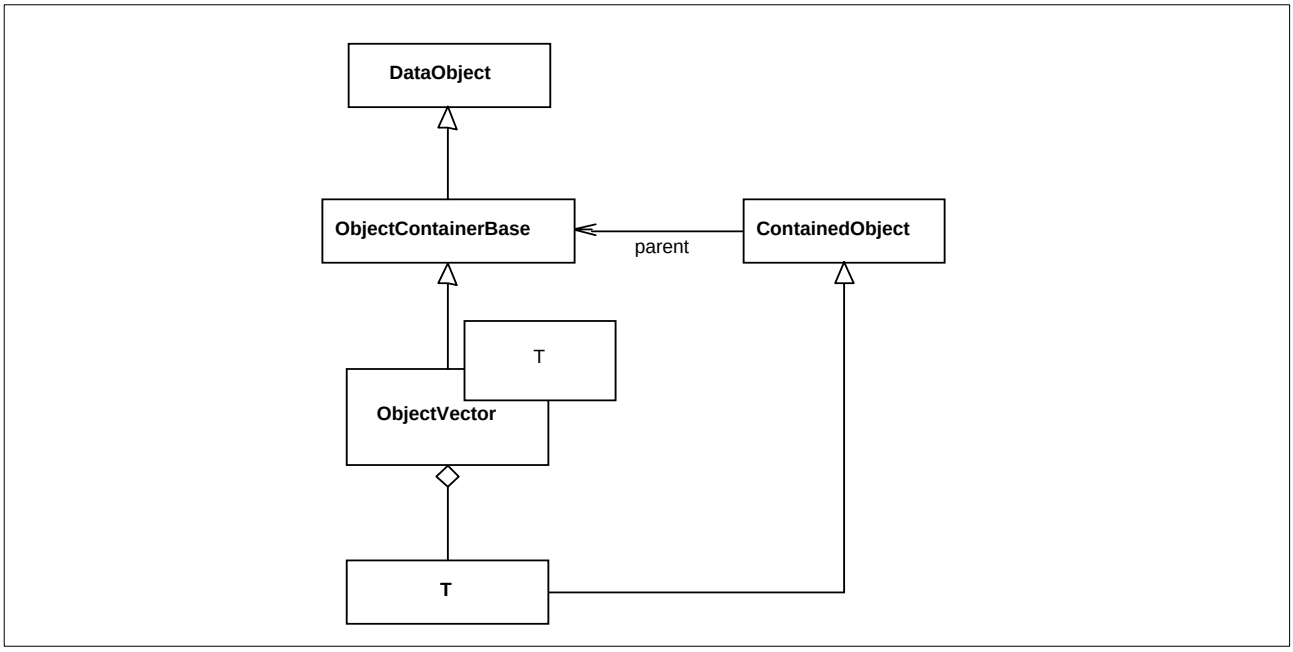

Figure 6.2 The relationship between the DataObject, ObjectVector and ContainedObject classes.

6.5 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.

```
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):

```
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:

```
trackContainer.release(h1);
```

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:

```
container1.push_back(h1); // Add to fist 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 6.3 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 6.3 Use of the ObjectVector templated class.

```
 1:  typedef ObjectVector<MyTrack> MyTrackVector;
 2:  MyTrackVector *tracks;
 3:  MyTrackVector::iterator it;
 4:
 5:  for( it = tracks->begin(); it != tracks->end(); it++ ) {
 6:    // Get the energy of the track and histogram it
 7:    double energy = (*it)->fourMomentum().e();
 8:    m_hEnergyDist->fill( energy, 1. );
 9:  }
```


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.

6.6 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.

6.7 Defining Data Objects

If you want to create a new data object in the transient or persistent stores of Gaudi, you will have to define the structure of this object. This structure will be defined by C++-classes. These classes in general look very similar to each other; mainly they define the members of the class, which are either data values or which point to another class (eg. via a Smart Reference - see Section 6.9). For each of these members there is usually a set- and a get-method and some more stuff for the Smart Reference Vectors.

The writing of these classes is a tedious task and having to write this redundant information many times, of course, also bears the risk of many unnecessary typos. To overcome this problem one may use XML in conjunction with the `GaudiObjDesc` package to describe the data-objects. There were two key issues which led to the development of this description language:

- The core information of a data-object lies in the members of the class, most of the rest is redundant information which can be produced automatically around the members.
- There is a lot of information which also must be provided, but which has a default-value in most of the cases.

The information provided in the XML files can be used to produce not only the object information in the classes but also reflection information about the objects (see Section 11.10, "The Gaudi Introspection Service"). Future releases may also produce, e.g., converters, or a description of the object in other languages.

As an example, the following XML code describes an `MCParticle` class:

Listing 6.4 Part of the XML description of the MCParticle class

```
1: <class name='MCParticle' author='Pavel Binko' id='210' desc='The Monte
   Carlo particle kinematics information'>
2:   <base name='ContainedObject'/>
3:   <attribute name='subEvtID' type='short' desc='Sub-event ID'/>
4:   <relation name='originMCVertex' multiplicity='1' type='MCVertex'
   desc='Pointer to origin vertex'/>
5:   <relation name='decayMCVertices' multiplicity='M' type='MCVertex'
   desc='Vector of pointers to decay vertices'/>
6: </class>
```

All of the elements in this listing (eg. `<class>`, `<attribute>`, `<relation>`) have several attributes with default values (eg. for relations and attributes `" setMeth='TRUE' "`), which don't have to be mentioned explicitly. If one doesn't want to use the default, the only thing that has to be done is to set the corresponding attribute to another value. There are also several hooks which can be applied eg. to define your own methods if they were not created automatically. The complete syntax of this description language can be found on the web at t(*http://cern.ch/lhcb-comp/Frameworks/DataDictionary/*).

Once a set of data-objects is defined, the XML file has to be saved to the `xml` directory of the package. The production of the C++ header files containing the object description can be automated by adding a line to the CMT requirements file of the package, as shown for example below:

```
document obj2doth LHCbEventObj2Doth ../xml/LHCbEvent.xml
```

Another possibility is to produce the information by hand with the tools of the `GaudiObjDesc` package (eg. `GODCppHeaderWriter.exe`) and then compile it.

6.7.1 The class ID

The class definition on line 1 of Listing 6.4 contains an '`id`' attribute. This class identifier is required if the objects of this class are to be made persistent. It is used by the data persistency services to make the translation between the transient and persistent representations of the object, ising the conversoin mechanism described in Chapter 13. For this mechanism to work, these identifiers must uniquely identify the class and no two classes may have the same identifier. The procedure for allocating unique class identifiers is, for the time being, experiment specific.

Types which are derived from `ContainedObject` must have a class ID 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.

6.8 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.

6.8.1 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 in Listing 6.5.

Listing 6.5 Use of a SmartDataPtr object.

```
 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.

6.9 Smart References and Smart Reference Vectors

It is foreseen that data objects in the transient data stores can reference other objects in the same data store. This relationship can be described in the XML data description using the 'relation' attribute of the class definition, as shown on line 4 of Listing 6.4

The current implementation of these relationships use 'Smart References' and 'Smart Reference Vectors'. These are similar to smart pointers, they provide safe data access and automate the loading on demand of referenced data, and are 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.

The XML code in Listing 6.4 will generate Smart Reference and Smart Reference Vector declarations as shown below:

```
#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:

```
#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].

6.10 Persistent storage of data

6.10.1 Saving event data to a persistent store

Suppose that you have defined your own data type as discussed in section 6.7. Suppose futhermore 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 13)
- Put some instructions (i.e. options) into the job option file (see Listing 6.6)
- Register your object in the store us usual, typically in the `execute()` method of your algorithm.

```
// 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 6.6 Job options for output to persistent storage

```
ApplicationMgr.OutStream  = { "DstWriter" };

DstWriter.ItemList          = { "/Event#1", "/Event/MyTracks#1"};
DstWriter.EvtDataSvc        = "EventDataSvc";
DstWriter.Output            = "DATAFILE='RootDst.root' TYP='ROOT'";

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

6.10.2 Reading event data from a persistent store

Suppose you want to read back the file written out in the previous section. To do this, your job options would look something like those described in Section 4.4.2 on page 32.

Chapter 7
Event Data

7.1 Overview

This chapter is a place holder for documenting the experiment specific event data models.

Chapter 8
Detector Description

8.1 Overview

This chapter is a place holder for documenting how to access the detector description data in the Gaudi 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 8.1. 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.

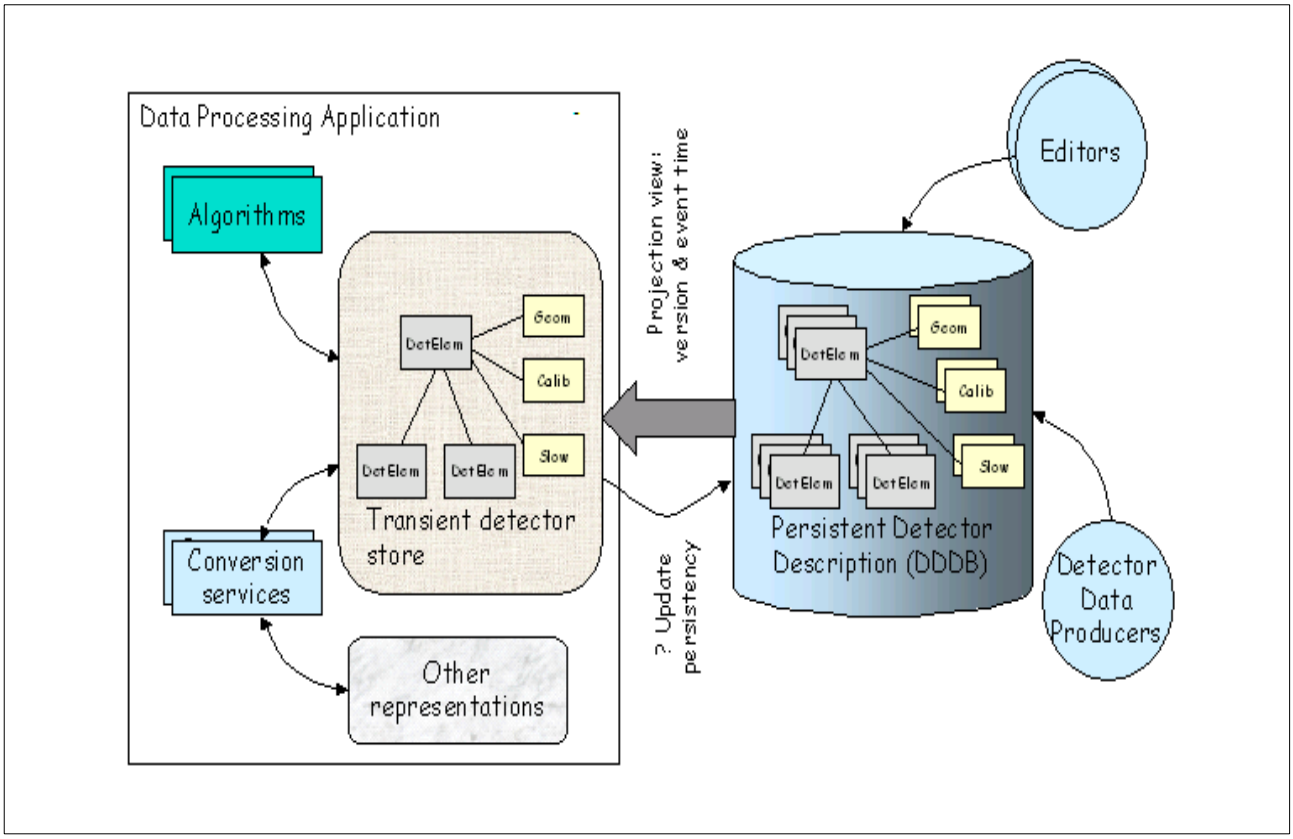

Figure 8.1 Overview of the Detector Description model.

Chapter 9
Histogram facilities

9.1 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 `IDataProviderSvc` interface discussed in Chapter 6. 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, using the methods defined in the `IHistogram1D` and `IHistogram2D` interfaces and documented on the AIDA (Abstract Interfaces for Data Analysis) project web pages: *http://wwwinfo.cern.ch/asd/lhc++/AIDA/*.

Internally, Gaudi uses the transient part of HTL (Histogram Template Library, *http://wwwinfo.cern.ch/asd/lhc++/HTL/*) to implement histograms.

9.2 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, always called “`/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. Note that the string "`/stat/`" can be omitted when referring to a histogram in the data store: "`/stat/simple`" is equivalent to "`simple`", **without** a leading "`/`".

As discussed in Section 5.2, the `Algorithm` base class defines a member function which returns a pointer to the `IHistogramSvc` interface of the standard histogram data service .

```
IHistogramSvc* histoSvc()
```

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 11.2.

9.3 Using histograms and the histogram service

The code fragment below shows how to book a 1D histogram and place it in a directory within the histogram data store, followed by a simple statement which fills the histogram.

```
#include "AIDA/IHistogram1d.h"
...
// Book 1D histogram in the histogram data store
IHistogram1d* m_hTrackCount= histoSvc()->
           book( "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) and must be unique within the RZ directory the histogram is assigned to. The name of the RZ directory is given by the directory and parent directories in the transient histogram store. Please note that HBOOK accepts only directory names, which are shorter than 16 characters and that HBOOK internally converts any directory name into upper case. 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 6.8.

```
SmartDataPtr<IHistogram1D> hist1D( histoSvc(), "simple/1" );
if( 0 != hist1D ) {
  // Print the found histogram
  histoSvc()->print( hist1D );
}
```

9.4 Persistent storage of histograms

By default, Gaudi does not produce a persistent histogram output. The options exist to write out histograms either in HBOOK or in ROOT format. The choice is made by giving the job option `ApplicationMgr.HistogramPersistency`, which can take the values "`NONE`" (no histograms saved, default), "`HBOOK`" or "`ROOT`". Depending on the choice, additional job options are needed, as described below.

9.4.1 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 Gaudi 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 v*
```

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.OuputFile = "histo.hbook";
```


Note that it is also possible to print the histograms to the standard output destination (HISTDO) by setting the following job option (default is `false`).

```
HistogramPersistencySvc.PrintHistos = true;
```

9.4.2 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 Gaudi 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 v*
```

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

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

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

```
HistogramPersistencySvc.OuputFile = "histo.rt";
```


Chapter 10
N-tuple and Event Collection facilities

10.1 Overview

In this chapter we describe facilities available in Gaudi 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.

10.2 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.

Gaudi 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 Gaudi 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 6).

10.2.1 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 always 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.). Note that the top level directory name "/NTUPLES/" can be omitted when referring to an N-tuple in the transient data store - in the example above the name could start with "FILE1" (without a leading "/").

10.2.2 Using the N-tuple Service.

This section explains the steps to be performed when defining an N-tuple:

- 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.

10.2.2.1 Defining N-tuple tags

When creating an N-tuple it is necessary to first define the tags to be filled in the N-tuple, as shown for example in Listing 10.1:

Listing 10.1 Definition of N-tuple tags.

```
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
```


Typically the tags belong to the filling algorithm and hence should be provided in the Algorithm's header file. Currently the supported data types are: `bool`, `long`, `float` and `double`. `double` types (Fortran REAL*8) are not recommened if using HBOOK for persistency: HBOOK will complain if the N-tuple structure is not defined in a way that aligns `double` types to 8 byte boundaries. In addition PAW cannot understand `double` types.

10.2.2.2 Booking and Declaring Tags to the N-tuple

Listing 10.2 shows how to book a column-wise N-Tuple. The first directory specifier (`FILE1` in the example) must correspond to an open output stream (see Section 10.2.3.2); lower directory levels are created automatically. After booking, the previously defined tags must be declared to the N-tuple; if not, they are invalid and will cause an access violation at run-time.

Listing 10.2 Creation of a column-wise N-tuple in a specified directory and file.

```
 1:  #include "GaudiKernel/NTuple.h"
 2:  ..
 3:  NTuplePtr nt1(ntupleSvc(), "FILE1/MC/1");
 4:  if ( !nt1 )    {    // Check if already booked
 5:   nt1=ntupleSvc()->book("FILE1/MC/1",CLID_ColumnWiseTuple,"Hello World");
 6:   if ( 0 != nt1 )    {
 7:   // Add an index column
 8:     status = nt1->addItem ("Ntrack", m_ntrk, 0, 5000 );
 9:   // Add a variable size column, type float (length=length of index col)
10:     status = nt1->addIndexedItem ("px", m_ntrk, m_px);
11:     status = nt1->addIndexedItem ("py", m_ntrk, m_py);
12:     status = nt1->addIndexedItem ("pz", m_ntrk, m_pz);
13:   // Another one, but this time of type bool
14:     status = nt1->addIndexedItem ("flg",m_ntrk, m_flag);
15:   // Another one, type integer, numbers must be within [0, 5000]
16:     status = nt1->addIndexedItem ("idx",m_ntrk, m_index, 0, 5000 );
17:   // Add 2-dim column: [0:m_ntrk][0:2]; numerical numbers within [0, 8]
18:     status = nt1->addIndexedItem ("hit",m_ntrk, 2, m_hits, 0, 8 );
19:   }
20:   else    {   // did not manage to book the N tuple....
21:     return StatusCode::FAILURE;
22:   }
23:  }
```

In previous versions of Gaudi (up to v8), indexed items were added with the `addItem` function, causing confusion for users. For this reason the calls to add indexed arrays and matrices were changed, these should now be added using the member function `addIndexedItem`. Please consult the doxygen code documentation for further details. The old calls still exist, however they are deprecated.

Row wise N-tuples are booked in the same way, but giving the type `CLID_RowWiseTuple`. However, only individual items (class `NTuple::Item`) can be filled, no arrays and no matrices. Clearly this excludes the usage of indexed items. For row-wise N-tuples to be saved in HBOOK format, it is recommended to use only `float` type, for the reasons explained in Section 10.2.3.3.

When using HBOOK for persistency, the N-tuple identifier ("1" in this example) must be a number and must be unique in a given directory. This is a limitation imposed by HBOOK RZ

directories. 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.

10.2.2.3 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]`.

Implicit bounds checking is not 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 7N-tuple must be written.

Listing 10.3 Filling an N-tuple.

```
 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. See Listing 10.2 for initialisation of m_ntuple
16:  status = m_ntuple->write();
17:  if ( !status.isSuccess() ) {
18:    log << MSG::ERROR << "Cannot fill id 1" << endreq;
19:  }
```

10.2.2.4 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 10.4.

Listing 10.4 Reading an N-tuple.

```
 1:  NTuplePtr nt(ntupleSvc(), "FILE1/ROW_WISE/2");
 2:  if ( nt )    {
 3:    long count = 0;
 4:    NTuple::Item<float> px, py, pz;
 5:    status = nt->item("px", px);
 6:    status = nt->item("py", py);
 7:    status = nt->item("pz", pz);
 8:    // Access the N tuple row by row and print the first 10 tracks
 9:    while ( nt->read().isSuccess() )    {
10:      log << MSG::INFO  << " Entry [" << count++ << "]:";
11:      log << " Px=" << px << " Py=" << py << " Pz=" << pz << endreq;
12:    }
13:  }
```

10.2.3 N-tuple Persistency

10.2.3.1 Choice of persistency technology

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.

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.

ROOT This data format is used by the interactive ROOT program.

In the current implementation, N-tuples must use the same persistency technology as histograms. The choice of technology is therefore made in the same way as for histograms, as described in Section 9.4. Obviously the options have to be given only once and are valid for both histograms and N-tuples. The only difference is that histograms are saved to a different output file (defined by the job option `HistogramPersistencySvc.OuputFile`), a different output file (or set of output files) must be defined for the N-tuples.

10.2.3.2 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' 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, which must be the same as the data store directory name described in Section 10.2.1. Several files are possible, corresponding to different data store directories whose name can be chosen at will.

The detailed syntax of the options is as follows. In each case only the three leading characters are significant: DATAFILE=<...>, DATABASE=<...> or simply DATA=<...> would lead to the same result.

- **DAT**AFILE**='<file-specs>'**
  Specifies the datafile (file name) of the output stream.
- **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.

A similar option NTupleSvc.Input exists for N-tuple input.

10.2.3.3 Saving row wise N-tuples in HBOOK

Since the persistent representation of row wise N-tuples in HBOOK is done by floats only, a convention is needed to access the proper data type. By default the float type is assumed, i.e. all data members are of float type. This is the recommended format.

It is possible to define row wise N-tuples in Gaudi containing data types other than float. This was the default in Gaudi versions previous to v8, where the first row of the N-tuple contained the type information. This possibility can be switched on by using the option

```
HistogramPersistencySvc.RowWiseNTuplePolicy = "USE_DATA_TYPES";
```

which also provides backwards compatibility for reading back old N-tuples produced with old Gaudi versions. Remember however that when using PAW to read N-tuples produced using this option, you must skip the first row and start with the second event.

10.3 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 10.3.2.3.

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 10.3.2.3. Gaudi algorithms are called only when this pre-selector also accepts the event, and normal event processing can start.

10.3.1 Writing Event Collections

Event collections are written to the data file using a Gaudi 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.

10.3.1.1 Defining the Address Tag

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

```
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 10.2.2.1. When booking the N-tuple, the address tag must be declared like any other tag, as shown in Listing 10.5. It is recommended to use the name "`Address`" for this tag.

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.

Listing 10.5 Connecting an address tag to an event collection N-tuple.

```
1:    StatusCode status = service("EvtTupleSvc", m_evtTupleSvc);
2:    if ( status.isSuccess() )   {
3:      NTuplePtr nt(m_evtTupleSvc, "/NTUPLES/EvtColl/Collection");
4:     ... Book N-tuple
5:     // Add an event address column
6:     status = nt->addItem ("Address", m_evtAddress);
```

Please note that the event tuple service necessary for writing event collections is not instantiated by default and hence must be specified in the job options file:

Listing 10.6 Adding the event tag collection service to the job options.

```
1:  ApplicationMgr.ExtSvc += { "TagCollectionSvc/EvtTupleSvc" };
```

It is up to the user to locally remember within the algorithm writing the event collection tuple the reference to the corresponding service. Although the `TagCollectionSvc` looks like an N-tuple service, the implementation is different.

10.3.1.2 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 10.7 also demonstrates the setting of a filter to steer whether the event is written out to the event collection.

Listing 10.7 Fill the address tag of an N-tuple at execution time:

```
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:    }
```

10.3.2 Event Collection Persistency

10.3.2.1 Output File Specification

Conversion services exist to convert event collection 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:

```
EvtTupleSvc.Output = {"FILE1 DATAFILE='coll.root' TYP='ROOT' OPT='NEW'",
                      "FILE2 ...",
                       ...
                      "FILEN ..."};
```


where `<coll.root>` should be replaced by the name of the file to which you wish to write the event collection. FILE1 is the logical name of the output file - it could be any other string.

The options are the same as for N-tuples (see Section 10.2.3.2) with the following additions:

- **TYP='<typ-spec>'**
  Specifies the type of the output stream. Currently supported types are:
  - ROOT: Write as a ROOT tree.
  - MS Access: Write as a Microsoft Access database.
  - There is also weak support for the following database types1:
    - 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.

  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 (e.g. GaudiRootDb) implementations of the database access drivers:

  ApplicationMgr.DLLs += { "GaudiDb", "GaudiRootDb" };
- **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.
- **AUT**HENTICATION**='<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. Additional options are availible when accessing events using an event tag collection.

10.3.2.2 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 10.8 (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 10.7), and of the `EvtCollectionStream`.

Listing 10.8 Job options for writing out an event collection

```
1:  ApplicationMgr.OutStream   = { "Sequencer/EvtCollection" };
2:  ApplicationMgr.ExtSvc += { "TagCollectionSvc/EvtTupleSvc" };
3:  EvtCollection.Members = { "EvtCollectionWrite/Selector",
    "EvtCollectionStream/Writer"};
4:  Writer.ItemList   = { "/NTUPLES/EvtColl/Collection" };
5:  Writer.EvtDataSvc = "EvtTupleSvc";
6:  EvtTupleSvc.Output = { "EvtColl DATAFILE='MyEvtCollection.root'
                            OPT='NEW' TYP='ROOT'" };
```

10.3.2.3 Reading Events using Event Collections

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

Listing 10.9 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:  };
```

The tags that were not already introduced earlier are:

- **COL**LECTION
  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".
- **ADD**RESS **(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**ECTION **(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**CTION **(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 10.10. Note that the factory declaration on line 16 is mandatory in order to allow Gaudi 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 10.2.3.2.

Listing 10.10 Example of a function object for second stage pre-selections.

```
 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
```

10.3.3 Interactive Analysis using Event Tag Collections

Event tag collections are very similar to N-tuples and hence allow within limits some interactive analysis.

10.3.3.1 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");
root [2] TFile* f = new TFile("tuple.root");
root [3] TTree* t = (TTree*)f->Get("<local>_MC_ROW_WISE_2");
root [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 10.11):

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

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

Listing 10.11 Interactive analysis of ROOT N-tuples: interactive.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, NUMSYMB;
 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("NUMSYMB",&NUMSYMB);
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:  }
```

10.4 Known Problems

Nothing is perfect and there are always things to be sorted out....

- When building the `GaudiRootDb` package on Linux using CMT you must first set up the ROOT environment, by sourcing the `setup.csh` file

Chapter 11
Framework services

11.1 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 Gaudi framework. The *Job Options* service, the *Message* service, the *Particle Properties* service, the *Chrono & Stat* service, the *Auditor* service, the *Random Numbers* service, the *Incident* service and the *Introspection* service are available in this release. The *Tools* service is described in Chapter 12.

We also describe how to implement new services for use within the Gaudi 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.

11.2 Requesting and accessing services

The Application manager only creates by default the `JobOptionsSvc` and `MessageSvc`. Other services are created on demand the first time they are accessed, provided the corresponding DLL has been loaded. The services in the GaudiSvc package are accessible in this way by default - these are the default data store services (`EventDataSvc`, `DetectorDataSvc`, `HistogramDataSvc`, `NTupleSvc`) and many of the framework services described in this chapter and in Chapter 12 (`ToolSvc`, `ParticlePropertySvc`, `ChronoStatSvc`, `AuditorSvc`, `RndmGenSvc`, `IncidentSvc`).

Additional services can be made accessible by loading the appropriate DLL, using the property `ApplicationMgr.DLLs` in the job options file, as shown for example in Listing 6.6 on page 59.

Sometimes it may be necessary to force the Application Manager to create a service at initialisation (for example if the order of creation is important). This can be done using the property `ApplicationMgr.ExtSvc`. In the example below this option is used to create a specific type of persistency service.:

Listing 11.1 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 36 of Listing 5.1 on page 40. 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: The third argument is optional: when set to `true`,

Listing 11.2 Code to access the IParticlePropertySvc interface from an Algorithm

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

the service will be created if it does not already exist; if it is missing, or set to `false`, the service will not be created if it is not found, and an error is returned.

In components other than Algorithms and Services (e.g. Tools, Converters), which do not provide the `service` function, you can locate a service using the `serviceLocator` function:

```
#include "GaudiKernel/IParticlePropertySvc.h"
...
IParticlePropertySvc* m_ppSvc;
IService* theSvc;

StatusCode sc=serviceLocator()->getService("ParticlePropertySvc",theSvc,true);
if ( sc.isSuccess() ) {
  sc = theSvc->queryInterface(IID_IParticlePropertySvc, (void**)&m_ppSvc);
}
if ( sc.isFailure) {
...
```


11.3 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.

11.3.1 Algorithm, Tool and Service Properties

In general a concrete Algorithm, Service or Tool will have several data members which are used to control execution. These data members (*properties*) can be of a basic data type (`int`, `float`, etc.) or class (`Property`) encapsulating some common behaviour and higher level of functionality. Each concrete Algorithm, Service, Tool declares its properties to the framework using the `declareProperty` templated method as shown for example on line 12 of Listing 11.4 (see also Section 5.3.2 on page 42). The method `setProperties()` is called by the framework in the initialization phase; this causes the job options service to make repeated calls to the `setProperty()` method of the Algorithm, Service or Tool (once for each property in the job options file), which actually assigns values to the data members.

11.3.1.1 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 (Listings 11.3 and 11.4).

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

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


Listing 11.3 EventCounter.h

```
 1: #include "GaudiKernel/Algorithm.h"
 2: #include "GaudiKernel/Property.h"
 3: class EventCounter : public Algorithm {
 4: public:
 5:     EventCounter( const std::string& name, ISvcLocator* pSvcLocator );
 6:     ~EventCounter( );
 7:     StatusCode initialize();
 8:     StatusCode execute();
 9:     StatusCode finalize();
10: private:
11:     IntegerProperty m_frequency;
12:     int m_skip;
13:     int m_total;
14: };
```

Listing 11.4 EventCounter.cpp

```
 1: #include "GaudiAlg/EventCounter.h"
 2: #include "GaudiKernel/MsgStream.h"
 3: #include "GaudiKernel/AlgFactory.h"
 4:
 5: static const AlgFactory<EventCounter>    Factory;
 6: const IAlgFactory& EventCounterFactory = Factory;
 7:
 8: EventCounter::EventCounter(const std::string& name, ISvcLocator*
 9:                            pSvcLocator) :
10:                            Algorithm(name, pSvcLocator),
11:                            m_skip ( 0 ), m_total( 0 ) {
12:   declareProperty( "Frequency", m_frequency=1 );  // [1]
13:   m_frequency.setBounds( 0, 1000 );               // [2]
14: }
15:
16: StatusCode EventCounter::initialize() {
17:   MsgStream log(msgSvc(), name());
18:   log << MSG::INFO << "Frequency: " << m_frequency << endreq;    // [3]
19:   return StatusCode::SUCCESS;
20: }
```

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.

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

```
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

```
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:

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

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

11.3.1.2 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 Gaudi 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:

```
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.

11.3.2 Accessing and modifiying properties

Properties are private data which are initialised by the framework using the default values given when they are declared in constructors, or the values read from the job options file. On occasions it may be necessary for components to access (or even modify) the values of properties of other components. This can be done by using the `getProperty()` and `setProperty()` methods of the `IProperty` interface. In the example below,,an algorithm stores the default value of a cut of its sub-algorithm, then executes the sub-algorithm with a different cut, before resetting the cut back to its default value. Note that in the example we begin with a pointer to an `Algorithm` object, not an `IAlgorithm` interface. This means that we have access to the methods of both the `IAlgorithm` and `IProperty` interfaces and can therefore call the methods of the `IProperty` interface. In the general one may need to navigate to the `IProperty` interface first, as explaned in Section 16.3.2.

```
Algorithm* myAlg;
...
std:string dfltCut;
StatusCode sc = myAlg->getProperty( "TheCut", dfltCut );
if ( sc.isSuccess() ) {
  msgAlg->setProperty( "TheCut", "0.8" );
  StatusCode sc1 = myAlg->execute();
  ...
}
if( scl.isSuccess() ) msgProp->setProperty( "The Cut", dfltCut );
```

11.3.3 Job options file format

An example of a job options file was shown in Listing 4.2 on page 32. 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

11.3.3.1 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:

```
<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.: `12e2;`

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. `{};`

11.3.3.2 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 be 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 exist then the append operation "+=" behaves as assignment operation "=".

11.3.3.3 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.opts"
```

The "`filename`" can also contain the path where this file is located. By convention we use "`.opts`" as the file extension for job options. The include directive can be placed anywhere in the job option file, usually at the top (as in C++). Note that the value of a property defined earlier in the file may be over-ridden by assigning a new value to the same property: the last value assigned is the valid value! This makes it possible to over-ride the value of a property defined in a previously included file without changing the include file.

It is possible to use environment variables in the `#include` statement, either standalone or as part of a string. Both Unix style ("`$environmentvariable`") and Windows style ("`%environmentvariable%`") are understood (on both platforms!). For example, in line 2: of Listing 4.2 the logical name `$STDOPTS`, which is defined in the `GaudiExamples` package, points to a directory containing a number of standard job options include files that can be used by applications.

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 has already been included, so there is no need for `#ifndef` statements as in C or C++ to check for multiple inclusion.

11.3.3.4 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.:

```
#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.

11.3.3.5 Switching on/off printing

By default, the Job Options Service prints out the contents of the Job Options files to the standard output destination. The possibility exists to switch off this printing, and to toggle between the two states, as shown below:

```
1: // Switch off printing
2: #pragma print off
3: ..(some job options)
4: //Switch printing back on
5: #pragma print on
```

In the example above, all lines between line 2 and line 5 will not be printed.

11.4 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
//(1=VERBOSE, 2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL, 7=ALWAYS)
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.

11.4.1 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 11.5 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 11.5 Use of a MsgStream object.

```
 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:

```
enum MSG::Level { VERBOSE, DEBUG, INFO, WARNING, ERROR, FATAL, ALWAYS };
```

Thus the code in Listing 11.5 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& operator <<(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 char, unsigned char, int, float, etc. are supported

`MsgStream& operator <<(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 `level`.

Accepted Stream Manipulators

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

endl Inserts a newline sequence. Opposite to the ostream behaviour this manipulator does not flush the buffer. Full name: `MsgStream& endl(MsgStream& s)`

ends Inserts a null character to terminate a string. Full name: `MsgStream& ends(MsgStream& s)`

flush Flushes the stream's buffer but does not produce any output! Full name: `MsgStream& flush(MsgStream& s)`

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 `std::cout`. Full name: `MsgStream& endreq(MsgStream& s)`

endmsg Same as `endreq`

11.5 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!

11.5.1 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 11.2 on page 82 shows how to access this service from within an algorithm.

11.5.2 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`[1]

11.5.3 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. Particles that are known to Geant but not to stdHep, such as Deuteron, have an artificial stdHep ID using unreserved (>7) digits. 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 Gaudi

The `IParticlePropertySvc` interface provides the following functions:

Listing 11.6 The IParticlePropertySvc interface.

```
// 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:

```
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;
```

11.5.4 Examples

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

Listing 11.7 Code fragment to find particle properties by particle name.

```
// 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 11.8 Code fragment showing how to use the map iterators to access particle properties.

```
// 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;
}
```


11.6 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.20.

11.6.1 Code profiling

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

```
/// ...
IChronoStatSvc* 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.:

```
 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 11.7.

11.6.2 Statistical monitoring

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

```
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 *Flag*s for the given tag, and the "accumulated" *weight*, which is the product of all *Weight*s 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.

11.6.3 `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.

11.6.3.1 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:

```
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

11.6.3.2 Stat

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

```
 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.

11.6.4 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:

```
 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:  }
13: Stat   stat  ( svc , "Good Stat", a); /// OK
14: } ///  end of the scope!
15: /// ...
```


11.7 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.

11.7.1 Enabling the Auditor Service and specifying the enabled Auditors

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

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

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

```
// Enable the NameAuditor and ChronoAuditor
AuditorSvc.Auditors = { "NameAuditor", "ChronoAuditor" };
```


11.7.2 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:

```
// Enable initialize and finalize auditing & disable execute auditing
// for the myAlgorithm Algorithm
myAlgorithm.AuditInitialize = true;
myAlgorithm.AuditExecute    = false;
myAlgorithm.AuditFinalize   = true;
```

11.7.3 Implementing new Auditors

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

```
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:

```
#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);
};
```


11.8 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, Gaudi 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 Gaudi 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 11.1 shows the general architecture of the Gaudi 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

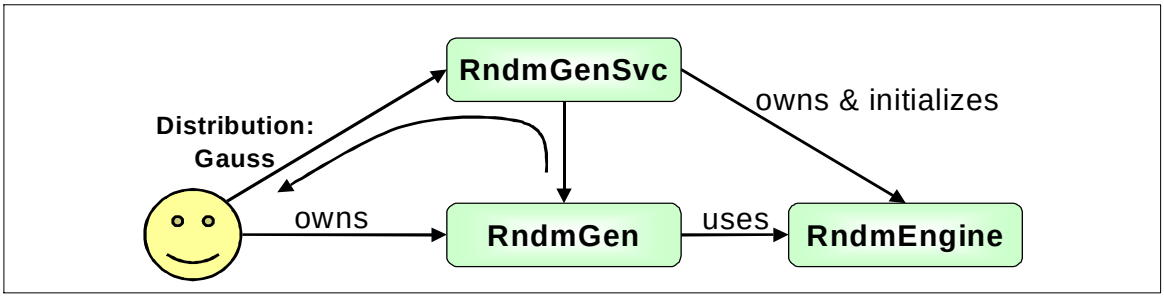

Figure 11.1 The architecture of the random number service. The client requests from the service a random number generator satisfying certain criteria

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 `Rndm::Bit()`
- Generate a flat distribution with boundaries [`min, max`] with parameters: `Rndm::Flat(double min, double max)`
- Generate a gaussian distribution with parameters: `Rndm::Gauss(double mean, double sigma)`
- Generate a poissonian distribution with parameters: `Rndm::Poisson(double mean)`
- Generate a binomial distribution according to `n` tests with a probability `p` with parameters: `Rndm::Binomial(long n, double p)`
- Generate an exponential distribution with parameters: `Rndm::Exponential(double mean)`
- Generate a Chi**2 distribution with `n_dof` degrees of freedom with parameters: `Rndm::Chi2(long n_dof)`
- Generate a Breit-Wigner distribution with parameters: `Rndm::BreitWigner(double mean, double gamma)`
- Generate a Breit-Wigner distribution with a cut-off with parameters: `Rndm::BreitWignerCutOff (mean, gamma, cut-off)`
- Generate a Landau distribution with parameters: `Rndm::Landau(double mean, double sigma)`
- Generate a user defined distribution. The probability density function is given by a set of descrete points passed as a vector of doubles: `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 11.9 .

Listing 11.9 Example of the use of the random number generator to fill a histogram with a Gaussian distribution within a standard Gaudi algorithm

```
 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 11.10.

Listing 11.10 Example of the use of the random number generator within a standard Gaudi 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 11.9

```
 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.

11.9 The Incident Service

The Incident service provides synchronization facilities to components in a Gaudi 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 Gaudi 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*.

```
#include "GaudiKernel/IIncidentListener.h"
#include "GaudiKernel/IIncidentSvc.h"

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.

11.9.1 Known Incidents

Table 11.1 Table of known named incidents

Incident Type	Source	Description
BeginEvent	ApplicationMgr	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.
EndEvent	ApplicationMgr	The ApplicationMgr has finished processing the physics event. The Event data store is not yet purged at this moment.

11.10 The Gaudi Introspection Service

Introspection is the ability of a programming language to interact with objects from a meta-level. The Gaudi Introspection package defines a meta-model which gives the layout of this meta-information.

The data to fill this meta-information (i.e. the *dictionary*) can be generated by the Gaudi Object Description package (described in Section 6.7 on page 55) by adding a few lines to the CMT requirements file, as shown for example in Listing 11.11.

Listing 11.11 CMT requirements for generation of data dictionary of the LHCbEvent package

```
#---- dictionary
document obj2dict LHCbEventObj2Dict -group=dict ../xml/LHCbEvent.xml
library LHCbEventDict -group=dict ../dict/*.cpp
macro LHCbEventDict_shlibflags  "$(use_linkopts) $(libraryshr_linkopts)"
```

The C++-code generated in this way is compiled into a dll and loaded into the Gaudi Introspection Model at runtime.

To get a reference to information about a real object, clients have to use the Gaudi Introspection Service (`IntrospectionSvc`). The service can also be used to load the meta-information into the model. The Gaudi Introspection Service is already used in several places in the framework (e.g. Interface to Python, Data Store Browser).

Further information about this service is available at *http://cern.ch/lhcb-comp/Frameworks/DataDictionary/default.htm*.

11.11 Developing new services

11.11.1 The Service base class

Within Gaudi 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 11.2 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.

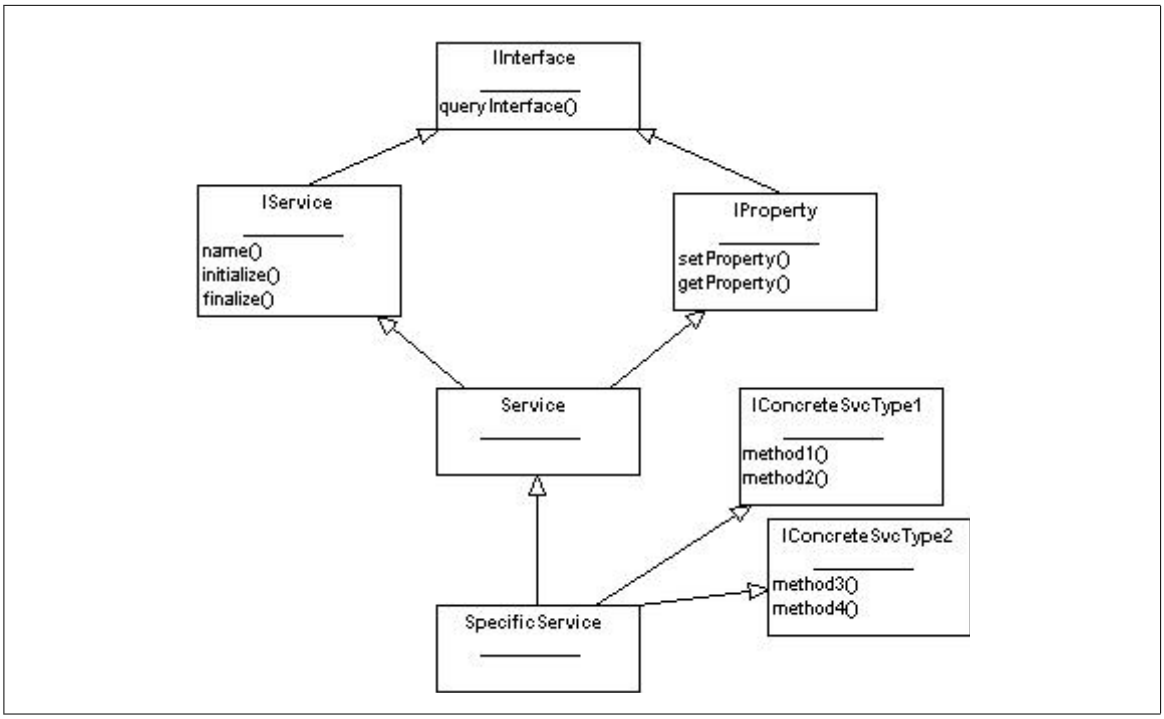

Figure 11.2 Implementation of a concrete service class. Though not shown in the figure, both of the IConcreteSvcType interfaces are derived from IInterface.

As discussed above, it is necessary to derive from the Service base class so that the concrete service may be made accessible to other Gaudi 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 11.2 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.

11.11.2 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 11.12 An interface class

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

class IConcreteSvcType1 : virtual public IInterface {
public:
  void method1() = 0;
  int method2() = 0;
}


#include "IConcreteSvcType1.h"

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


Listing 11.13 A minimal service implementation

```
#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(constIID& riid, void** ppvIF);
};


// 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 12
Tools and ToolSvc

12.1 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 component[1] can request a tool. Algorithms, Services and other Tools 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 how to write concrete Tools.

In section 12.3 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.

12.2 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 algorithm whereas the `initialize()` and `finalize()` methods are invoked automatically by the framework while initializing the parent algorithm.

1. In this chapter we will use an Algorithm as example component requesting tools.

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.

12.2.1 "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`.

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`.

12.2.2 The Tool classes

12.2.2.1 The AlgTool base class

The main responsibilities of the `AlgTool` base class (see Listing 12.1) 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 and management of the additional tools interfaces..

Listing 12.1 The definition of the AlgTool Base class. Highlighted in bold are methods relevant for the implementation of concrete tools.

```
 1: class AlgTool : public virtual IAlgTool,
 2:                 public virtual IProperty {
 3:
 4: public:
 5: // Standard Constructor.
 6: AlgTool( const std::string& type, const std::string& name,
            const IInterface* parent);
 7:
 8: ISvcLocator* serviceLocator() const;
 9: IMessageSvc* msgSvc() const;
10:
11: virtual StatusCode setProperty( const Property& p );
12: virtual StatusCode setProperty( std::istream& s );
13: virtual StatusCode setProperty( const std::string& n,
                                   const std::string& v );
14: virtual StatusCode getProperty(Property* p) const;
15: virtual const Property& getProperty( const std::string& name ) const;
16: virtual StatusCode getProperty( const std::string& n,std::string& v )
                     const;
17: virtual const std::vector<Property*>& getProperties( ) const;
18:
19: StatusCode setProperties();
20:
21: template <class T>
22:   StatusCode declareProperty(const std::string& name, T& property) const
23:
24: virtual const std::string& name() const;
25: virtual const std::string& type() const;
26: virtual const IInterface*  parent() const;
27:
28: virtual StatusCode initialize();
29: virtual StatusCode finalize();
30:
31: virtual StatusCode queryInterface(const IID& riid, void** ppvUnknown);
32: void declInterface( const IID&, void*);
33: template <class I> class declareInterface {
      public:
        template <class T> declareInterface(T* tool)
    }
34:
35: protected:
36:   // Standard destructor.
37:   virtual ~AlgTool();
```

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.

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.

Properties - A template method 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, unless specified otherwise in the JobOptions. For details on Properties see section 11.3.1.

`IAlgTool` Interface - It consists of three accessor methods for the identification and managment of the tools: `type()`, `name()` and `parent()`. These methods are all implemented by the base class and should not be overridden. Two additional methods, `initialize()` and `finalize()`, allow concrete tools to be configured after creation and orderly terminated before deletion. An empty implementation is provided by the `AlgTool` base class and concrete tools need to implement these methods only when relevant for their purpose. The `ToolSvc` is responsible for calling these methods at the appropriate time.

Tools Interfaces - Concrete tools must implement additional interfaces that will inherit from `IAlgTool`. When a component implements more that one interface it is necessary to "recognize" the various interfaces. This is taken care of by the `AlgTool` base class once the additional interface is declared by a concrete tool (or tools' base class). The declaration of the additional interface must be done in the constructor of a concrete tool and is done via the template method `declareInterface`.

12.2.2.2 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.

12.2.2.3 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`.

Concrete tools must 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. The `ToolSvc` interacts with specialized

tools only through the additional tools interface, therefore the interface itself must inherit from the `IAlgTool` interface in order for the tool to be correctly managed by the `ToolSvc`.

The inheritance structure of derived tools is shown in Figure 12.1. `ConcreteTool1` implements one additional abstract interface while `ConcreteTool2` and `ConcreteTool3` derive from a base class `SubTool` that provides them with additional common functionality.

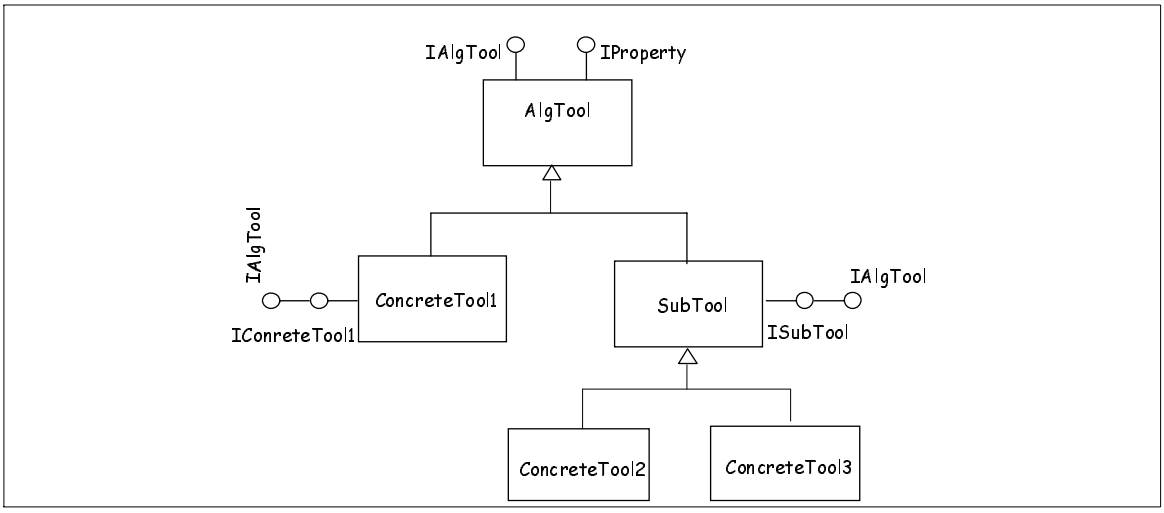

Figure 12.1 Tools class hierarchy

12.2.2.4 Implementation of concrete tools

An example minimal implementation of a concrete tool is shown in Listings 12.2, 12.3 and 12.4, taken from the LHCb `ToolsAnalysis` example application ..

Listing 12.2 Example of a concrete tool additional interface

```
1:  static const InterfaceID IID_IVertexSmearer("IVertexSmearer", 1 , 0);
2:
3:  class IVertexSmearer : virtual public IAlgTool {
4:  public:
5:    /// Retrieve interface ID
6:    static const InterfaceID& interfaceID() { return IID_IVertexSmearer; }
7:    // Actual operator function
8:    virtual StatusCode smear( MyAxVertex* ) = 0;
9:  };
```

Listing 12.3 Example of a concrete tool minimal implementation header file

```
1:  #include "GaudiKernel/AlgTool.h"
2:  class VertexSmearer : public AlgTool, virtual public IVertexSmearer {
3:  public:
4:    // Constructor
5:    VertexSmearer( const std::string& type, const std::string& name,
                     const IInterface* parent);
6:    // Standard Destructor
7:    virtual ~VertexSmearer() { }
8:    // specific method of this tool
9:    StatusCode smear( MyAxVertex* pvertex );
```


Listing 12.4 Example of a concrete tool minimal implementation file

```
 1:  #include "GaudiKernel/ToolFactory.h"
 2:  // Static factory for instantiation of algtool objects
 3:  static ToolFactory<VertexSmearer> s_factory;
 4:  const IToolFactory& VertexSmearerFactory = s_factory;
 5:
 6:  // Standard Constructor
 7:  VertexSmearer::VertexSmearer(const std::string& type,
                                  const std::string& name,
                                  const IInterface* parent)
                           : AlgTool( type, name, parent ) {
 8:
 9:    // Locate service needed by the specific tool
10:    m_randSvc = 0;
11:    if( serviceLocator() ) {
12:      StatusCode sc=StatusCode::FAILURE;
13:      sc = serviceLocator()->service( "RndmGenSvc", m_randSvc, true );
14:    }
15:    // Declare additional interface
16:    declareInterface<IVertexSmearer>(this);
17:
18:    // Declare properties of the specific tool
19:    declareProperty("dxVtx", m_dxVtx = 9 * micrometer);
20:    declareProperty("dyVtx", m_dyVtx = 9 * micrometer);
21:    declareProperty("dzVtx", m_dzVtx = 38 * micrometer);
22:  }
23:  // Implement the specific method ....
24:  StatusCode VertexSmearer::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 12.4

In addition a concrete tool class must specify a single constructor with the same parameter signatures as the constructor of the `AlgTool` base class as shown in line 5 of Listing 12.3.

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

1. Define the specific interface (inheriting from the `IAlgTool` interface).
2. Derive the tool class from the AlgTool base class
3. Provide the constructor
4. Declare the additional interface in the constructor.
5. Implement the factory adding the lines of code shown in Listing 12.4
6. Implement the specific interface methods.

In addition if a tool requires special initialization and termination you can implement the initialize and finalize methods.

12.3 The ToolSvc

The `ToolSvc` manages `Tools`. It is its responsibility to create tools, configure them, make them available to `Algorithms` or `Services` and terminate them in an orderly fashion before deleting them.

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 12.2.

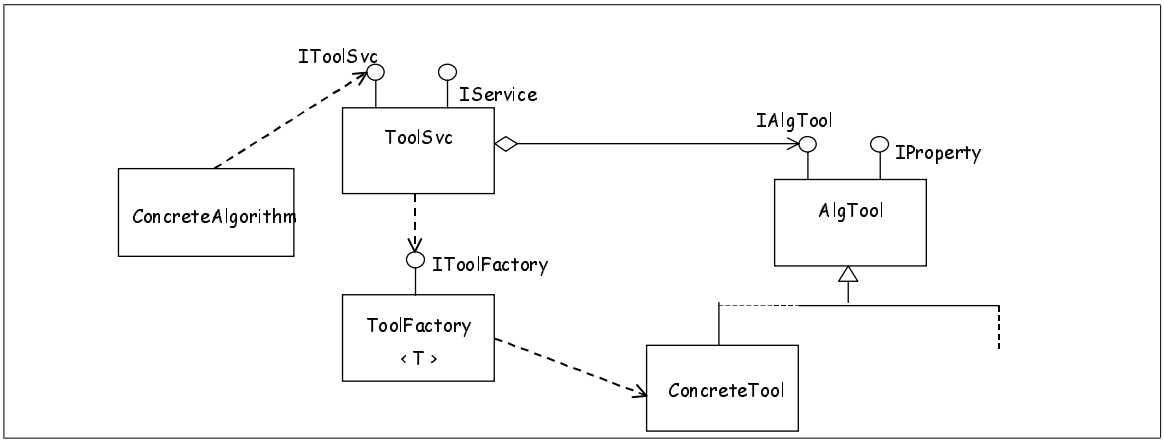

Figure 12.2 ToolSvc design diagram

Immediately after having created a tool, the `ToolSvc` will configure it by setting its properties and calling the tool `initialize`() method.

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. Before deleting the tools the `ToolSvc` will cleanly terminate them by calling their `finalize`() method.

The `ToolSvc` is created by default by the `ApplicationMgr` and algorithms wishing to use the service can do so via the algorithm `toolSvc()` accessor method. Services and AlgTools need to retrieve it using the `serviceLocator()` method of their respective base classes.

12.3.1 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 12.5

The arguments of the method shown in Listing 12.5, line 1, are the tool `type` (i.e. the class), the tool additional interface ID and the `IAlgTool` interface of the returned tool. In addition there are two arguments with default values: one is the `IInterface` of the component

Listing 12.5 The IToolSvc interface methods

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

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 12.5, 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.

When retriving concrete tools, it is recommended to use the two templated functions provided in the `IToolSvc` interface file which are shown in Listing 12.6.

Listing 12.6 The IToolSvc template methods

```
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 12.7 shows an example of retrieval of a shared and of a common tool.

Listing 12.7 Example of retrieval by an algortihm of a shared tool in line 4: and of a private tool in line 10:

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

12.4 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.

12.4.1 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 prototype version of these classes is provided in the current release of Gaudi.

12.4.1.1 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 12.8

Listing 12.8 Methods of the IAssociator Interface that must be implemented by concrete associators

```
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: for example 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 12.9). 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 12.4.1.3.

Listing 12.9 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 ) {...}
```

12.4.1.2 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 12.2.2.1) 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 12.10 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 11.9 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.

12.4.1.3 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 12.10.

Listing 12.10 Example of setting properties for an associator via jobOptions

```
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 12.11. When no association is found, a StatusCode::FAILURE is returned.

Listing 12.11 Checking if objects to be associated are of the correct type

```
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:  }
```

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

Listing 12.12 Extracted code from the AsctExampleAlgorithm

```
 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();
          vmcparts->end() != itm; itm++) {
13:       AxPartCandidate* mptry = 0;
14:       StatusCode sc = m_pAsct->retrieveInverse( *itm, mptry );
15:       if( sc.isSuccess() ) {...}
16:       else {...}
17:  }
18:  // Example of retrieving direct one-to-many information from an
19:  // associator
20:  SmartDataPtr<AxPartCandidateVector> candidates(evt,
                                          "/Anal/AxPartCandidates");
21:  std::vector<ContainedObject*> pptry;
22:  AxPartCandidate* itP = *(candidates->begin());
23:  StatusCode sa =
       m_pAsct->retrieveDirect(itP, pptry, MCParticle::classID());
24:  if( sa.isFailure() ) {...}
25:  else {
26:    for (std::vector<ContainedObject*>::iterator it = pptry.begin();
            pptry.end() != it; it++ ) {
27:         MCParticle* imc = dynamic_cast<MCParticle*>( *it );
28:    }
29:  }
```


Chapter 13
Converters

13.1 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 Gaudi. 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.

13.2 Persistency converters

Gaudi 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 Gaudi release. Other persistency technologies have been implemented for LHCb, in particular the reading of data from LHCb DSTs based on ZEBRA.

Figure 13.1 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. An introduction to the persistency mechanism of Gaudi can be found in reference [9].

One of the issues considered when designing the Gaudi 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

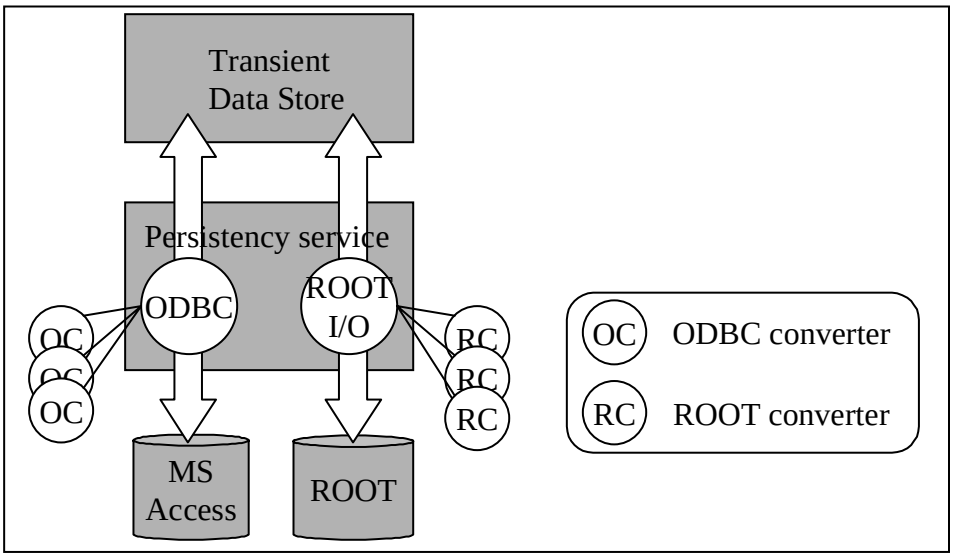

Figure 13.1 Persistency conversion services in Gaudi

which reside in different stores (i.e. files and databases) and even between objects in different types of store.

Figure 13.1 shows that data may be read from an ODBC database and/or 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.

13.3 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 13.2 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`
- `IConverter`
- `IOpaqueAddress`

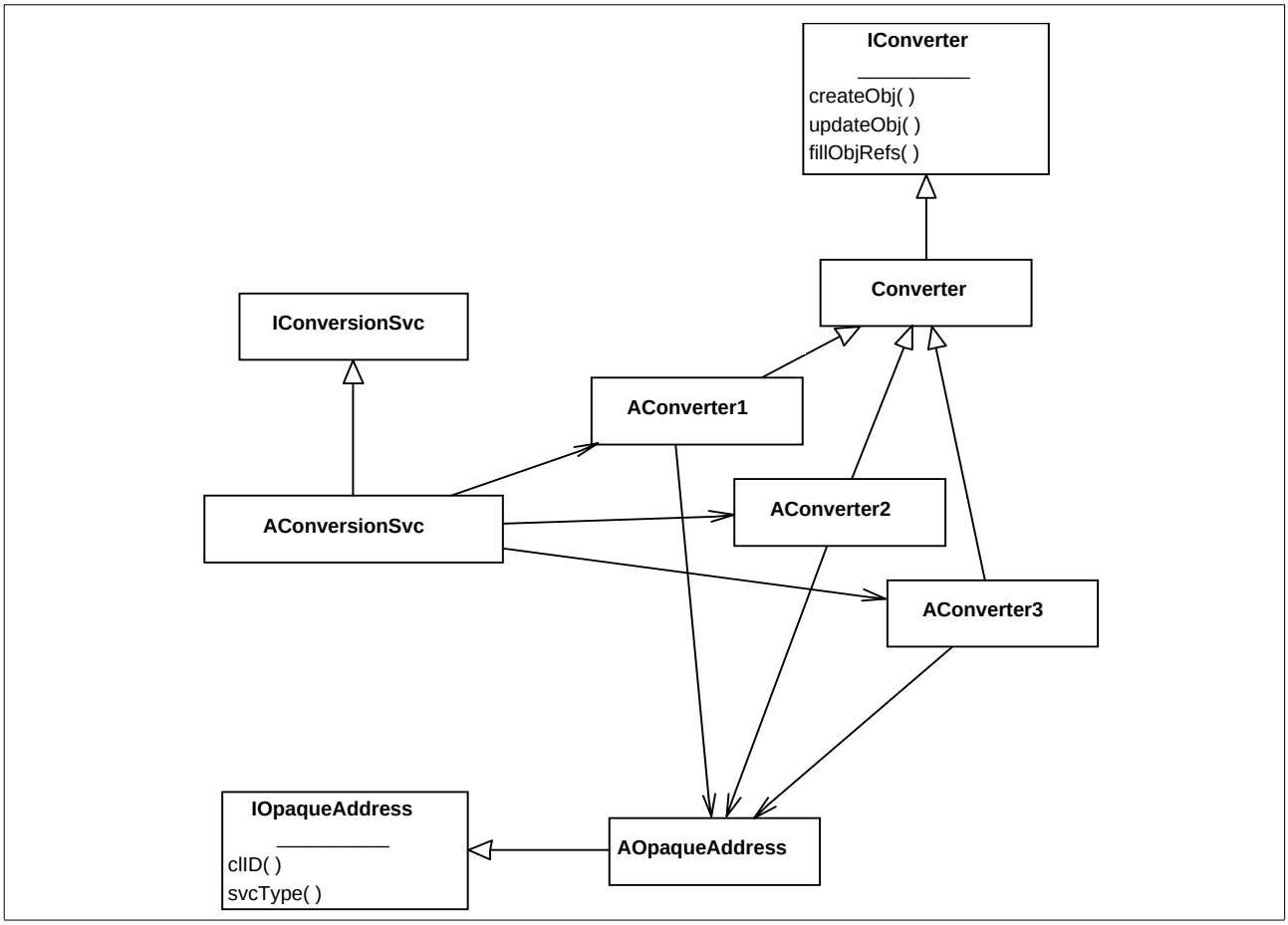

Figure 13.2 The classes (and interfaces) collaborating in the conversion process.

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 Section 6.10).

13.4 The conversion process

As an example (see Figure 13.3) we consider a request from the event data service to the persistency service for an object to be loaded from a data file.

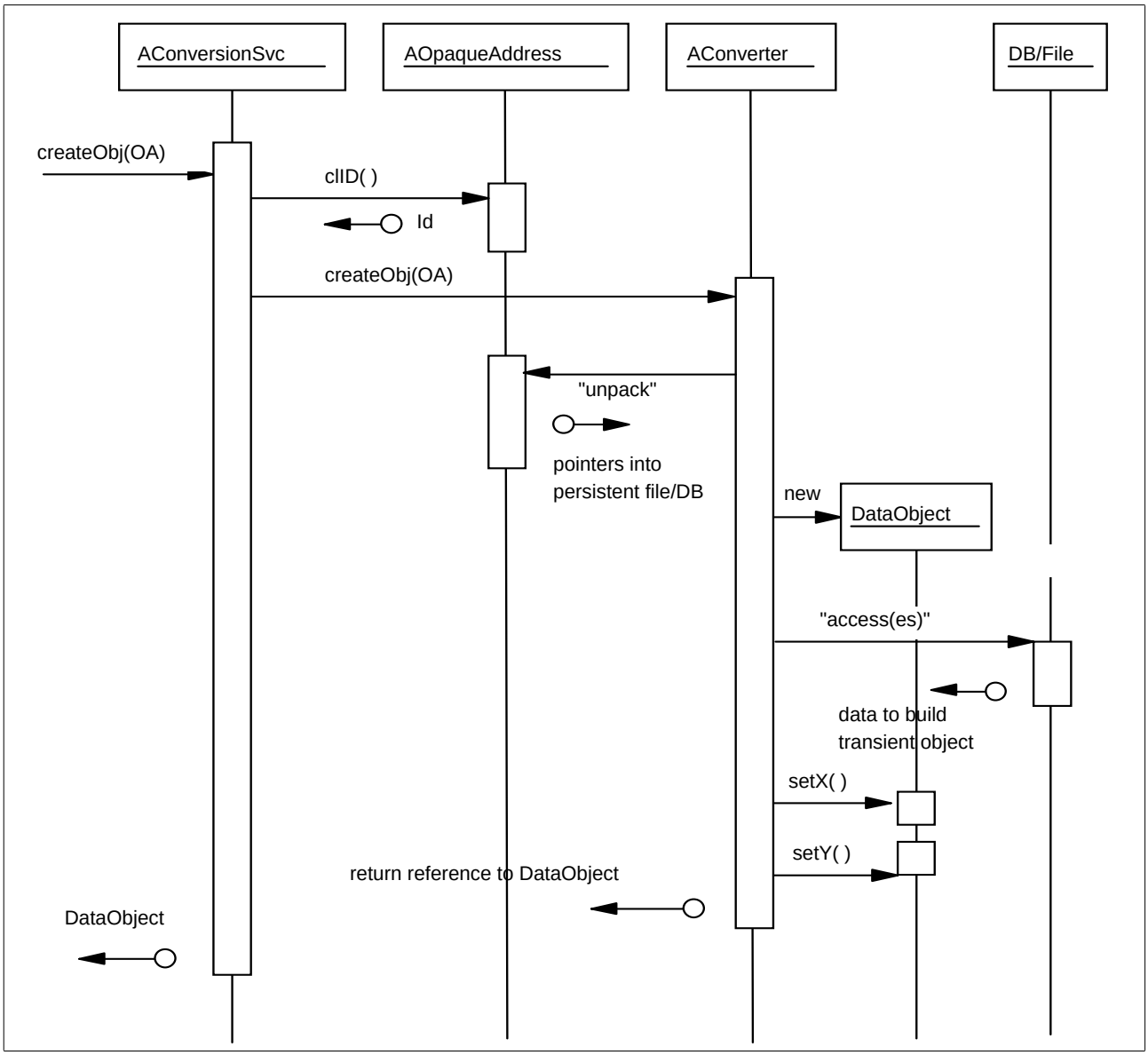

Figure 13.3 A trace of the creation of a new transient object.

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 Monte-Carlo tracks to Monte-Carlo vertices) and is generated by the Object Description Tools when a relation tag is found in the XML class description (see Section 6.9).

    2. Filling of raw C++ pointers. This is only necessary if the object points to an object in another store, e.g. the detector data store, and should be avoided in classes foreseen to be made persistent. 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).

13.5 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.

Listing 13.1 An example converter class

```
// 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 converter shown in Listing 13.1 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 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()`.

13.6 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 Gaudi data store is also tree like. In the transient world Gaudi 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 Gaudi 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.

The following sections are an introduction to the machinery provided by the Gaudi framework to achieve the migration of transient objects to persistent objects. The ROOT specific aspects are not discussed here; the ROOT I/O engine is documetned on the ROOT web site *http://root.cern.ch*). Note that Gaudi only uses the I/O engine, not all ROOT classes are available. Within Gaudi the ROOT I/O engine is implemented in the `GaudiRootDb` package.

13.7 The Conversion from Transient Objects to ROOT Objects

As for any conversion of data from one representation to another within the Gaudi 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 must be built.

Whenever objects must change their representation within Gaudi, data converters are involved. For the ROOT case, the converters must have some knowledge of ROOT internals and of the service finally used to migrate ROOT objects (->`TObject`) to a file. They 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 generic data conversion mechanism relies on two functionalities, which must be present:

- When writing or reading objects, the object’s data must be "serializable". The corresponding persistent type is of a generic type, the data are stored as a machine independent byte stream. This method is implemented automatically if the class is described using the Gaudi Object Description tools (described in Section 6.7 on page 55). When reading objects, an empty object must be created before any de-serialization can take place. The constructor must be called. This functionality does not imply any knowledge of the conversion mechanism itself and hence can be encapsulated into an object factory simply returning a DataObject. These data object factories are distinguished within Gaudi through the persistent data type

information, the class ID. For this reason the class ID of objects, which are written must only depend on the object type, i.e. every class needs it's own class ID. The instantiation of the appropriate factory is done by a macro. Please see the RootIO example for details how to instantiate the factory.

13.8 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 Gaudi release. If you desperately want to use Objectivity or one of the ODBC databases, please contact Markus Frank (*Markus.Frank@cern.ch*).

Chapter 14
Scripting and Interactivity

14.1 Overview

A scripting capability has been added to the Gaudi framework. The current 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.

In keeping with the design philosophy of the Gaudi architecture, scripting is defined by an abstract scripting service interface, with the possibility of there being several different implementations. The first implementation available is based on Python, a public-domain programming language. Python is ideal both as a scripting interface for modern systems, and as a standalone rapid-development language. Its object-oriented nature mixes well with frameworks written in C++.

The Python scripting language will not be described in detail here. There are many Python books available, among them we recommend:

- *Learning Python*, by M. Lutz & D. Ascher, O'Reilly, 1999
- *Programming Python (2nd ed.)*, by M. Lutz, O'Reilly, 2001

14.2 How to enable Python scripting

Three 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.
3. Load and start a Gaudi application from a Python shell.

14.2.1 Using a Python script for configuration and control

One can avoid using a job options text file for configuration by specifying a Python script as a command line argument, as shown in Listing 14.1.

Listing 14.1 Using a Python script for job configuration

```
myjob MyPythonScript.py                          [1]
```

Note:

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 Gaudi-specific Python commands described in Section 14.3.6.

14.2.2 Using a text JobOptions file and giving control to the Python interactive shell

Python scripting is enabled when using a job options text file for job configuration by adding the lines shown in Listing 14.2 to the job options file.

Listing 14.2 Job Options text file entries to enable Python scripting

```
ApplicationMgr.DLLs    += { "GaudiPython" };                    [1]
ApplicationMgr.Runable  = "PythonScriptingSvc";                 [2]
PythonScriptingSvc.StartupScript = "../options/AnalysisTest.py"; [3]
```

Notes:

1. This entry specifies the component library that implements Python scripting.
2. This entry specifies that the Python scripting should take the control (runable) of the application.
3. Optional startup python script.

Once the initial configuration has been established by the job options text file, control will be handed over to the Python shell when the startup script, if specified, will be executed. The user can then issue interactive commands.

14.2.3 Starting a Gaudi application from the Python shell

It is also possible to bootstrap a Gaudi application directly from a Python shell. The user needs to import the Python extension module called *gaudimodule*, which allows the interaction with Gaudi from Python. Listing 14.3 shows a small Python program that instantiates a Gaudi

application, configure it and runs for a number of events. This program would work from the Python shell a long as the environment (LD_LIBRARY_PATH / PATH) is properly set up.

Listing 14.3 Example of a Python program that executes a Gaudi program

```
from gaudimodule import *
theApp = AppMgr()
theApp.JobOptionsType = 'NONE'
theApp.EvtSel = 'NONE'
theApp.config()
theApp.Dlls = ['GaudiAlg']
myseq = theApp.algorithm('Sequencer/MySeq')
myseq.Members = ['EventCounter/Count1', 'EventCounter/Count2']
theApp.topAlg = ['MySeq']
theApp.initialize()
theApp.run(10)
theApp.exit()
```

14.3 Current functionality

The current functionality is limited to the following capabilities:

1. The ability to set and get basic properties for all framework components (Algorithms, Services, Auditors etc.) and the main ApplicationMgr that controls the application. Arrays of simple properties are mapped into Python *Lists*.
2. The ability to interact with the transient data stores. Browsing store contents, registering, unregistering and retrieving objects, getting and setting object data members (with the help of the IntrospectionSvc) and limited method invocation.
3. The ability to interact with the Histograms (1D and 2D) in the transient store. This includes booking, filling, dumping contents, etc.
4. The ability to add new services and component libraries and access their capabilities.
5. The ability to control the execution of the application by adding Algorithms into the list of top level Algorithms, executing single events or a set of events, executing single Algorithms, etc.
6. The ability to define Python Algorithms that will be managed and scheduled as normal Gaudi Algorithms.

14.3.1 Property manipulation

An example of the use of the scripting language to display and set component properties is shown in Listing 14.4:

Listing 14.4 Property manipulation from the Python interactive shell

```
>>> theApp                                                    [1][2]
<AppMgr object at 00AD22E8>
>>> theApp.ExtSvc                                             [3]
['IntrospectionSvc', 'ParticlePropertySvc']
>>> theApp.ExtSvc = theApp.ExtSvc + ['AnotherSvc']            [4]
>>> theApp.ExtSvc
['IntrospectionSvc', 'ParticlePropertySvc', 'AnotherSvc']
>>> theApp.EvtMax = 100
>>> theApp.properties()                                       [5]
{'EvtMax': 100, 'JobOptionsType': 'NONE', 'TopAlg': ['seq1', 'PhysAnalAlg'],
'Go': 0, 'Exit': 0, 'Dlls': ['GaudiAlg', 'GaudiIntrospection'],
'JobOptionsPath': '..\\home\\test.py', 'OutStream': [], 'OutputLevel': 3,
'EventLoop': 'EventLoopMgr', 'HistogramPersistency': 'NONE', 'EvtSel':
'NONE', 'ExtSvc': ['IntrospectionSvc', 'ParticlePropertySvc', 'AnotherSvc'],
'Runable': 'PythonScriptingSvc'}
>>> theApp.algorithms()                                       [6]
['seq1', 'WriteAlg', 'PhysAnalAlg']
>>> alg = Algorithm('WriteAlg')                               [7]
>>> alg.properties()
{'ErrorCount': 0, 'OutputLevel': 0, 'AuditExecute': 1, 'AuditInitialize': 0,
'Enable': 1, 'AuditFinalize': 0, 'ErrorMax': 1}
```

Notes:

1. The ">>>" is the Python shell prompt. Typing the name of a variable, Python prints its value in textual form.
2. The variable `theApp` is always defined and it represents the ApplicationMgr. An alias `g` has also been defined.
3. The name of the property is used as a data member in Python. It returns the correct type directly.
4. You can use the properties in normal Python expressions
5. The list of all properties (as a Python *Dictionary*) of a component can be obtained with the method `properties()`.
6. The list of algorithms can be obtained with the method `algorithms()`.
7. To access an Algorithm by name (creating it if it does not exist) the constructor `Algorithm()` is used. Similarly for services with `Service()`.

14.3.2 Creating Algorithms and Services

It is possible to create new Algorithms or Services as a result of a scripting command. Examples of this are shown in Listing 14.5:

If the specified Algorithm or Service does not exist, it is created. Its properties can immediately be accessed for read and write. They will be initialized when the application will start processing events.

Listing 14.5 Examples of Python commands that create new Algorithms or Services

```
>>> myseq = theApp.algorithm('Sequencer/MySeq')
>>> myseq.members = ['HelloWorld', 'WriteAlg']
MySeq        INFO HelloWorld doesn't exist - created and
                  appended to member list
MySeq        INFO WriteAlg already exists - appended to member list
>>> theApp.topAlg = ['MySeq']
>>> g.run(1)
HelloWorld   INFO initializing....
HelloWorld   INFO executing....
WriteAlg     INFO Generated event 5
StatusCode::SUCCESS
```

14.3.3 Interaction with Transient Data Stores

It is possible to get the list of data stores and to interact with them. The following commands are available:

`theApp.datastores()`
: Returns the list of all available data stores (all services that implement the `IDataProviderSvc` interface).

`theApp.datastore(name)`
: Returns a generic data store.

`theApp.histoSvc()`
: Returns the standard histogram data service (“HistogramDataSvc”)

`theApp.evtSvc()`
: Returns the standard event data service (“EventDataSvc”)

`theApp.detSvc()`
: Returns the standard detector data service (“DetectorDataSvc”)

`theApp.ntupleSvc()`
: Returns the standard ntuple data service (“NTupleSvc”)

`datastore.dump()`
: Prints the contents of the transient data store (names and types)

`datastore.clear()`
: Clears the contents of the transient data store

`datastore.object(name), datastore[name], datastore.retrieve(name)`
: Retrieves the named object from the transient data store. If the IntrospectionSvc[1] is loaded and the dictionaries are available for the requested object then it creates an object that can be introspected.

`datastore.register(name, obj), datastore.unregister(name)`
: Registers and unregisters an object to/from the data store

1. See Section 11.10 on page 105

`datastore.clear(),datastore.clear(name)`
Clears the whole store or a sub-tree.

A complete example of the capabilities on the interaction with the event transient store is shown in Listing 14.6.

Listing 14.6 Example of interaction with the event transient store taken from LHCb

```
evt = theApp.evtSvc()
evt.dump()
parts = evt['/Event/MC/MCParticles']
for p in parts :
  print p.particleID.id
```

14.3.4 Interaction with Histograms

The data store commands described in the previous section are also available for the histogram data store. The following histogram specific commands are available in addition:

`histosvc.histo(name), histosvc[name]`
Retrieves the histogram from the histogram data store. It returns either an 1D or 2D histogram.

`histosvc.book(id, title, xbin, xmin, xmax [,ybin, ymin, ymax])`
Books 1D or 2D histogram and registers it in the histogram store.

`histo.title(), dim(), mean(), rms(), maxbin(), minbin()`
Returns the title, dimensions, mean, rms, maximum bin contents, minimum bin contents of the histogram

`histo.fill(x [,y,w]), histo.reset()`
Fills 1D or 2D histogram, resets the contents

`histo.heights(), entries(), errors(), edges()`
Returns in a Python list the heights, entries, errors and edges of the 1D histogram

`histo.projections()`
Returns X and Y projections (tuple) of a 2D histogram

14.3.5 Interaction with Data Objects

If the dictionaries of the classes have been loaded by the object introspection service, then it is possible to browse and interact with the data objects in the transient store. Loading the dictionaries is done by configuring the application as it is shown in Listing 14.7

Listing 14.7 Configuring the Gaudi Introspection Service using the JobOptions text file

```
ApplicationMgr.DLLs   += { "GaudiIntrospection"};
ApplicationMgr.ExtSvc += { "IntrospectionSvc" };
IntrospectionSvc.Dictionaries = { "PhysEventDict" };
```


The following commands are currently available:

`obj.<attribute>`
: Returns the value of the attribute for the object. If the attribute is of a complex type it returns a reference to it, such that it can be browsed recursively.

`obj.<attribute> = value`
: Sets the value of attribute for the object if this is a simple type.

`obj.<method>()`
: Invokes the class method. This is currently only available for methods without arguments.

`obj.values()`
: Returns a Python dictionary with all the attributes and their values.

`obj.names()`
: Returns a Python list with all the available attribute names.

`obj.desc()`
: Prints the description of the class of the object.

`obj.type()`
: Returns the object type (C++ class)

14.3.6 Controlling job execution

There exist a few commands to control the job execution interactively:

`theApp.run(number)`
: The control is returned from the Python shell to the Gaudi environment with this command. The argument is the number (-1 for infinite) of events to be processed, after which control will be returned to the Python shell.

`theApp.exit()`
: Typing Ctrl-D (or Ctrl-Z in Windows) at the Python shell prompt will cause an orderly termination of the job. Alternatively, this command will also cause an orderly application termination.

14.4 Physics Analysis Environment

It is possible to declare an Algorithm in Python that can be declared to the list of top level algorithms to be executed for each event by the ApplicationMgr. This can be useful for constructing an interactive physics analysis environment. An example is shown in Listing 14.8.

Listing 14.8 Example Analysis

```
# -- User analysis algortihm class
class PhysAnalAlg(PyAlgorithm):                                    [1]
def initialize(self):
  global h1
  h1 = his.book('h1','Histogram Test', 10, 0., 10.)
  print '....User Analysis Initialized'
  return 1
def finalize(self):
  print 'Finalizing User Analysis...'
  return 1
def execute(self)
  cands = evt['Anal/AxPartCandidates']
  print 'Found '+ `len(cands)` + ' candidates'
  for c in cands :
    h1.fill(c.momentum)
  return 1

# -- Initialization and Configuration
his  = theApp.histoSvc()                                           [2]
evt  = theApp.evtSvc()
det  = theApp.detSvc()
pdt  = PartSvc()
physalg = PhysAnalAlg(theApp,'PhysAnalAlg')                        [3]
theApp.topAlg = theApp.topAlg + ['PhysAnalAlg']
```

Notes:

1. The analysis algorithm must inherit from the class PyAlgorithm
2. Useful variables to avoid long typing
3. An instance of the new class needs to be instantiated and declared in the list of top level algorithms.

Chapter 15
Visualization Facilities

15.1 Overview

In this chapter we describe how visualization facilities are provided to the applications based on the Gaudi framework. A prototype implementation (GaudiLab) exists in LHCb but is not distributed with the framework. It is based on the packages that constitute the Open Scientist suite (OpenGL, OpenInventor(soFree), Lab,... see *http://www.lal.in2p3.fr/OpenScientist/*). An event and geometry display application has been built using these facilities.

15.2 The data visualization model

The Gaudi architecture envisaged implementing data visualization using a similar pattern to data persistency. We do not want to implement visualization methods in each *data object*. In other words, we do not want to tell an object to “draw” itself. Instead we would implement *converters* as separate entities that are able to create specific graphical representations for each type of data object and for each graphical package that we would like to use. In that way, as for the persistency case, we decouple the definition and behaviour of the data objects from the various technologies for graphics. We could configure at run time to have 2D or 3D graphics depending on the needs of the end-user at that moment.

Figure 15.1 illustrates the components that need to be included in an application to make it capable of visualizing data objects (the "*So*" prefix in the names is taken from the GaudiLab implementation). The interactive user interface is a *Service* which allows the end-user to interact with all the components of the application. The user could select which objects to display, which algorithms to run, what properties of which algorithm to inspect and modify, etc. This interaction can be implemented using a graphical user interface or by using a scripting language.

The User interface service is also in charge of managing one or more GUI windows where views of the graphical representations are going to be displayed.

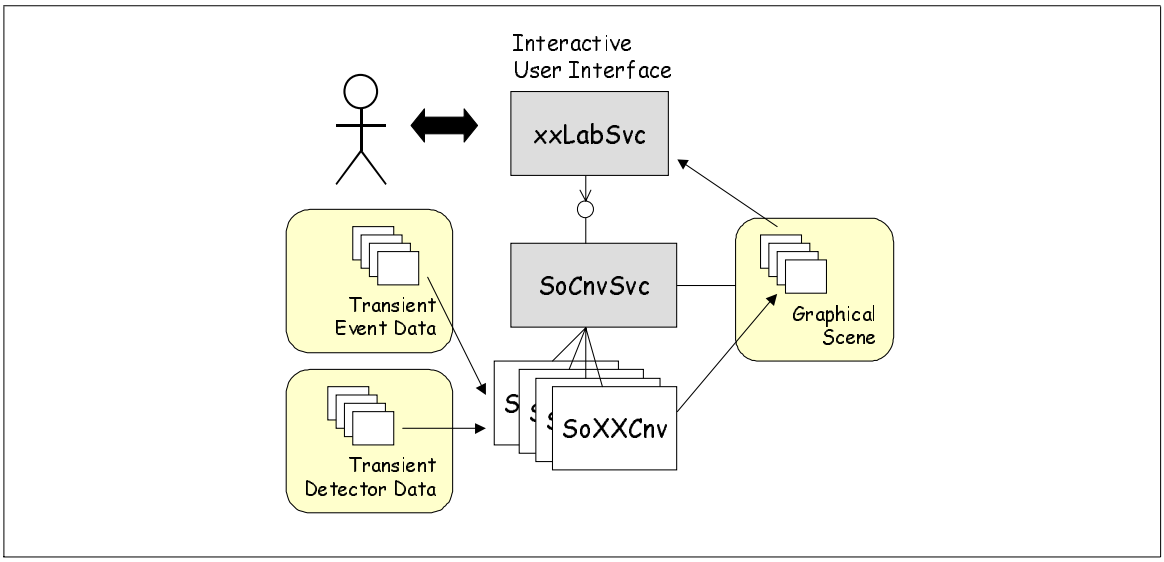

Figure 15.1 Components for visualization

The other main component is a *Conversion Service* that handles the conversion of objects into their graphical representation. This service requires the help of a number of specialized converters, one for each type of data object that needs to be graphically displayed. The transient store of graphical representations is shared by the conversion service, together with the converters, and the user interface component. The form of this transient store depends on the choice of graphics package. Typically it is the user interface component that would trigger the conversion service to start the conversion of a number of objects (next event), but this service can also be triggered by any algorithm that would like to display some objects.

Chapter 16
Framework packages, interfaces and libraries

16.1 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. We also discuss abstract inerfaces, which are one of the main design features of Gaudi

16.2 Gaudi Package Structure

The Gaudi software is decomposed into the packages shown in Figure 16.1.

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 [7] 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`, `GaudiIntrospection`), 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

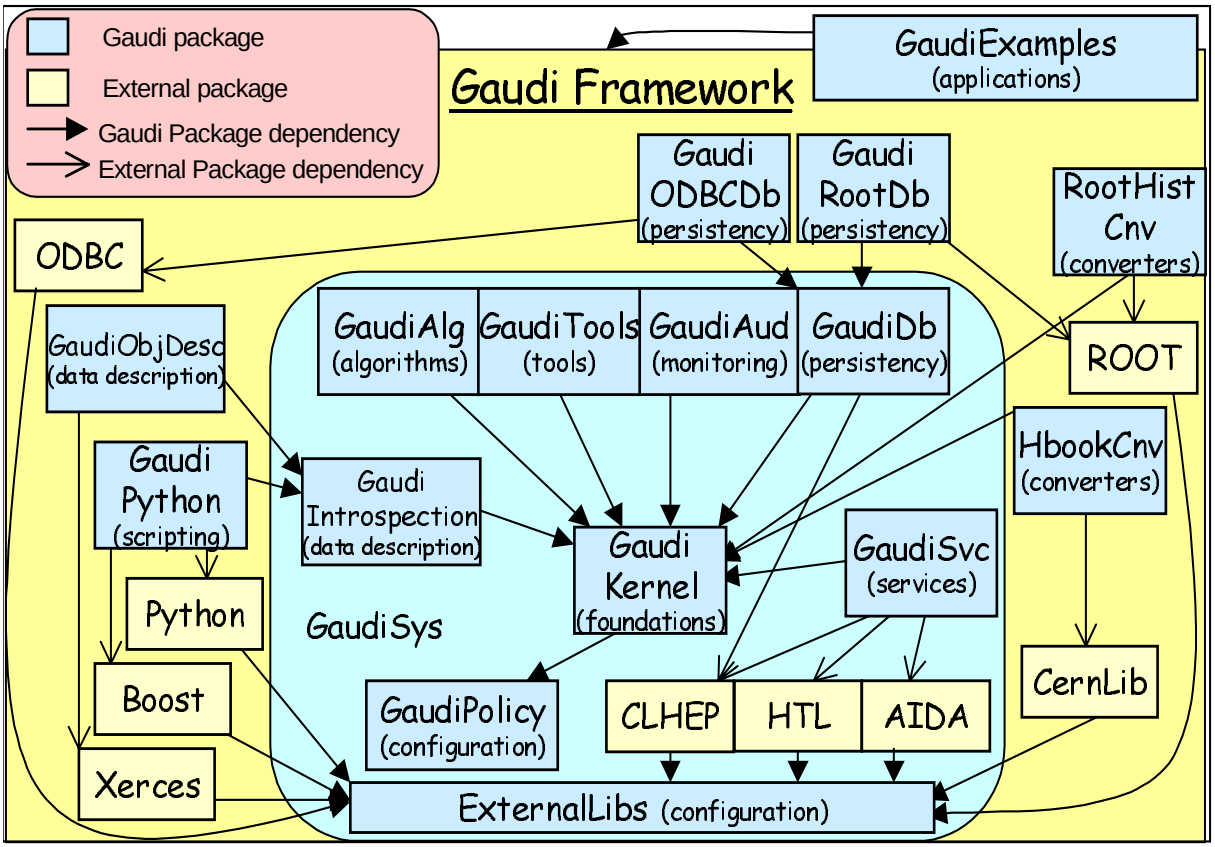

Figure 16.1 Package structure of the Gaudi software

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 two implementations of the event data persistency service (`GaudiRootDb` and `GaudiODBCDb`) which understand ROOT and ODBC compliant databases respecively. There is also a scripting service (`GaudiPython`) depending on the Python scripting language and a data description service (`GaudiObjDesc`) based on the Xerces XML parser. Finally, at the top level we find the applications (`GaudiExamples`) which depend on `GaudiSys` and the scripting and persistency services.

16.2.1 Gaudi Package Layout

Figure 16.2 shows the layout for Gaudi packages. Note that the binaries directories are not in CVS, they are created by CMT when building a package.

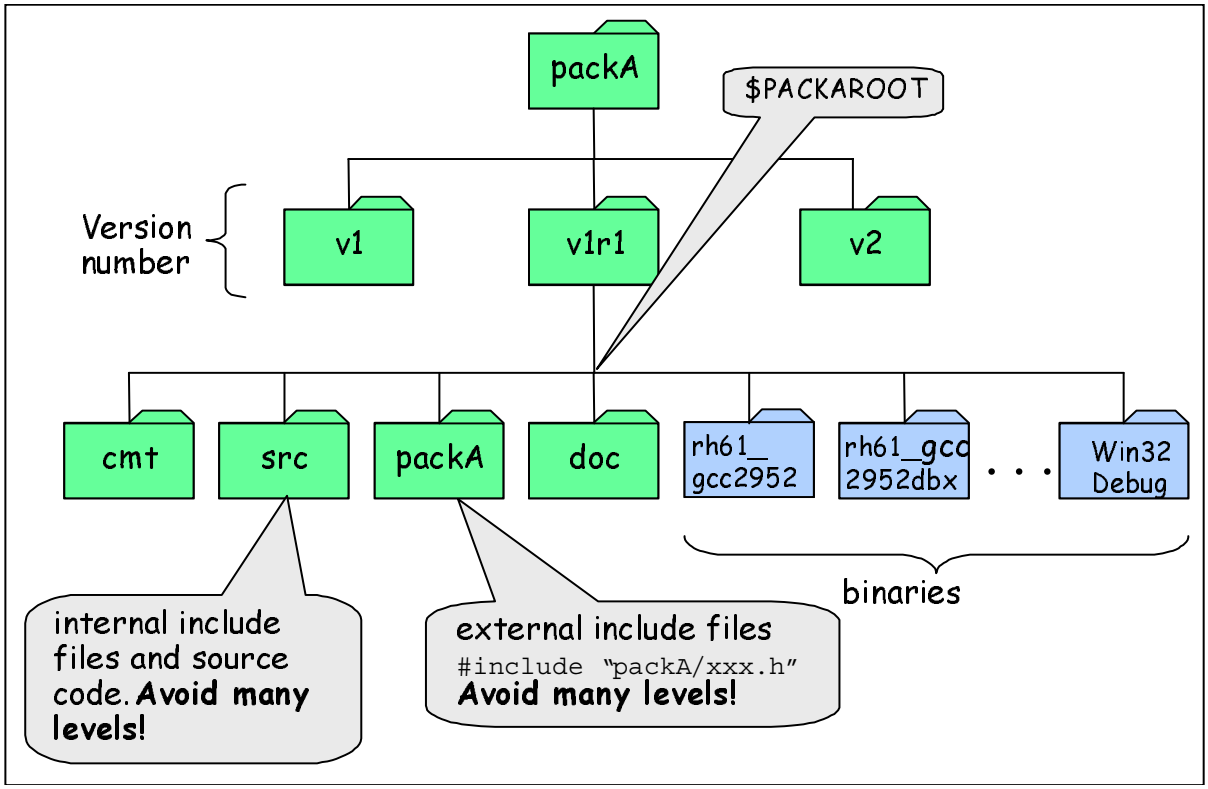

Figure 16.2 Layout of Gaudi software packages

16.2.2 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.

16.3 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 16.1:

Listing 16.1 Example of an abstract interface (IService)

```
 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;
22:   /// Retrieve ID of the Service. Not really used.
23:   virtual const IID&          type() const = 0;
24:   /// Initilize 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:
32: #endif  // GAUDIKERNEL_ISERVICE_H
```

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 Section 16.3.2).
- **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 Section 16.3.1) used by the query interface mechanism.

16.3.1 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 (generated from the interface name by a hash function) and 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 (by giving a unique name to the constructor), 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.

```
static const InterfaceID IID_Ixxx("Ixxx" /*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.

16.3.2 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 and allows us to navigate from one interface of a component to another, as shown for example in Listing 16.2, where we navigate from the `IMessageSvc` interface of the message service to its `IProperty` interface, in order to discover the value of its "`OutputLevel`" property.

Listing 16.2 Example usage of `queryInterface` to navigate between interfaces

```
1:  IMessageSvc* msgSvc();
2:  ...
3:  IProperty* msgProp;
4:  msgSvc()->queryInterface( IID_IProperty, (void**)&msgProp );
5:  std::string dfltLevel;
6:  StatusCode scl = msgProp->getProperty( "OutputLevel", dfltLevel );
```

The implementation of `queryInterface()` is usually not very visible since it is done in the base class from which you inherit. A typical implementation is shown in Listing 16.3:

Listing 16.3 Example implementation of queryInterface()

```
 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 16.3.1.

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.

16.4 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.

16.4.1 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.

16.4.1.1 Declaring Components

Components within the component library are declared 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 16.4 The `MyComponents_load.cpp` file

```
#include "GaudiKernel/DeclareFactoryEntries.h"

DECLARE_FACTORY_ENTRIES( MyComponents ) {        [1]
    DECLARE_ALGORITHM( MyAlgorithm );            [2]
    DECLARE_SERVICE  ( MyService );
}
```


Listing 16.4 The `MyComponents_load.cpp` file

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.

16.4.1.2 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 16.5 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`).

16.4.1.3 Loading Components

Components within the component library are loaded in a file `MyComponents_dll.cpp`. By convention, the name of this file is the package name concatenated with `_dll`. The contents of this file are shown below:

Listing 16.6 The `MyComponents_dll.cpp` file

```
#include "GaudiKernel/LoadFactoryEntries.h"

LOAD_FACTORY_ENTRIES( MyComponents )          [1]
```

Notes:

1. The argument of `LOAD_FACTORY_ENTRIES` is the name of the component library.

16.4.1.4 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 16.7 Selecting and running the desired tutorial example

```
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).

16.4.2 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`).

16.4.3 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.

16.4.4 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.

16.4.4.1 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 16.8. 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 16.4.1.4.

Listing 16.8 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)
```

16.4.5 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 17
Analysis utilities

17.1 Overview

In this chapter we give pointers to some of the third party software libraries that we use within Gaudi or recommend for use by algorithms implemented in Gaudi.

17.2 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 Gaudi, in the `GaudiSvc` and `GaudiDb` packages.

17.3 HTL

HTL ("Histogram Template Library") is used internally in Gaudi (GaudiSvc package) to provide histogramming functionality. It is accessed through its abstract AIDA[10] compliant interfaces. Gaudi uses only the transient part of HTL. Histogram persistency is available with ROOT or HBOOK.

The documentation on HTL is available at *http:cern.ch/anaphe/documentation.html*.

17.4 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://cern.ch/anaphe/documentation/Nag_C/NAGdoc/cl/html/mark6.html*

NAG C is not explicitly used in the Gaudi framework, but developers are encouraged to use it for mathematical computations. Instructions for linking NAG C with Gaudi can be found at *http://cern.ch/lhcb-comp/Support/NagC/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 Gaudi to redirect these messages to the Gaudi MessageSvc. This is documented at *http://cern.ch/lhcb-comp/Support/NagC/GaudiNagC.html*

17.5 ROOT

ROOT is used by Gaudi 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 10. Information about ROOT can be found at *http://root.cern.ch/*

Appendix A
References

1 GAUDI - Architecture Design Report [LHCb 98-064 COMP]

2 GAUDI online code documentation (*http://cern.ch/proj-gaudi/Doxygen/v9/*)

3 GAUDI - User Requirements Document [LHCb 98-065 COMP]

4 G.Barrand et al., GAUDI - A software architecture and framework for building LHCb data processing applications, [Proc CHEP 2000, Computer Physics Communications 140 (2001) 45-55] (*http://cern.ch/lhcb-comp/General/Publications/longpap-a152.pdf*)

5 LHCb Physical Units Convention (*http://cern.ch/lhcb-comp/Reconstruction/Conventions/units.pdf*)

6 Revised LHCb coding conventions [LHCb 2001-054 COMP]

7 CMT configuration management environment *http://www.lal.in2p3.fr/technique/si/SI/CMT/CMT.htm*R.Chytracek et al., The LHCb Detector Description Framework, [Proc CHEP 2000] *http://cern.ch/lhcb-comp/General/Publications/pap-a155.pdf*

8 I.Belyaev et al., Integration of GEANT4 with GAUDI, [Proc CHEP 2001] *http://cern.ch/lhcb-doc/presentations/conferencetalks/BelyaevProceedingsCHEP01.pdf*

9 M.Frank et al., Data Persistency Solution for LHCb, [Proc CHEP 2000], *http://cern.ch/lhcb-comp/General/Publications/pap-c153.pdf*

10 AIDA: Abstract Interfaces for Data Analysis *http://aida.freehep.org/*

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]

Option name	Default value	Meaning
EvtSel	""	If "NONE", no event input[a]
EvtMax	-1	Maximum number of events to process. The default is -1 (infinite) unless EvtSel = "NONE"; in which case it is 10.
TopAlg	{}	List of top level algorithms. Format: {<Type>/<Name>[, <Type2>/<Name2>,...]};
ExtSvc	{}	List of external services to be explicitly created by the ApplicationMgr (see section 11.2). 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 and N-tuple persistency mechanism. Available options are "HBOOK", "ROOT", "NONE"
Runable	"AppMgrRunable"	Type of runable object to be created by Application manager
EventLoop	"EventLoopMgr"	Type of event loop: "EventLoopMgr" is standard event loop "MinimalEventLoop" executes algorithms but does not read events
OutputLevel	MSG::INFO	Same as MessageSvc.OutputLevel. See Table B.2 for possible values

Table B.1 Standard Options for the Application manager [ApplicationMgr]

Option name	Default value	Meaning
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 a environment variable called JOBOPTPATH or setting the option to the application manager directly from the main program[b]. The coded option takes precedence.		
JobOptionsType	“FILE”	Type of file (FILE implies ascii)
JobOptionsPath	“jobOptions.txt”	Path for job options source

a. A basic DataObject object is created as event root ("/Event")

b. The setting of properties from the main program is discussed in Chapter 4.

Table B.2 Standard Options for the message service [MessageSvc]

Option name	Default value	Meaning
OutputLevel	0	Verboseness threshold level: 0=NIL,1=VERBOSE, 2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL, 7=ALWAYS
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:		
Option name	**Default value**	**Meaning**
OutputLevel	0	Message Service Verboseness threshold level. See Table B.2 for possible values
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:		
Option name	**Default value**	**Meaning**
OutputLevel	0	Message Service Verboseness threshold level. See Table B.2 for possible values

Table B.5 Standard Options for all Tools [<myTool>]

Any tool derived from the AlgTool base class can override the global `MessageSvc.OutputLevel` thus:		
Option name	**Default value**	**Meaning**
OutputLevel	0	Message Service Verboseness threshold level. See Table B.2 for possible values

Table B.6 Standard Options for all Associators [<myAssociator>]

Option name	Default value	Meaning
FollowLinks	true	Instruct the associator to follow the links instead of using cached information
DataLocation	""	Location where to get association information in the data store

Table B.7 Standard Options for Auditor service [AuditorSvc]

Option name	Default value	Meaning
Auditors	{};	List of Auditors to be loaded and to be used. See section 11.7 for list of possible auditors

Table B.8 Standard Options for all Auditors [<myAuditor>]

Any Auditor derived from the Auditor base class can override the global Auditor options thus:		
Option name	**Default value**	**Meaning**
OutputLevel	0	Message Service Verboseness threshold level. See Table B.2 for possible values
Enable	true	If false, application manager skips execution of the auditor

Table B.9 Options of Algorithms in GaudiAlg package (see Section 5.5)

Algorithm name	Option Name	Default value	Meaning
EventCounter	Frequency	1;	Frequency with which number of events should be reported
Prescaler	PercentPass	100.0;	Percentage of events that should be passed
Sequencer	Members		Names of algorithms in the sequence
Sequencer	BranchMembers		Names of algorithms on the branch
Sequencer	StopOverride	false;	If true, do not stop sequence if a filter fails

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		
Option name	**Default value**	**Meaning**
ItemList	{}	The list of data objects to be written to this stream, e.g. {“/Event#1”,”Event/MyTracks/#1”};
Preload	true;	Preload items in ItemList
Output	""	Output data stream specification. Format: {“DATAFILE='mydst.root' TYP='ROOT'”};
OutputFile	""	Output file specification - same as DATAFILE in previous option
EvtDataSvc	“EventDataSvc”	The service from which to retrieve objects.
EvtConversionSvc	"EventPersistencySvc"	The persistency service to be used
AcceptAlgs	{}	If any of these algorithms sets filterflag=true; the event is accepted
RequireAlgs	{}	If any of these algorthms 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)

Option name	**Default value**	**Meaning**
CnvServices	{}	Conversion services to be used by the service to load or store persistent data (e.g. "RootEvtCnvSvc")

Table B.12 Standard Options for conversion services (e.g. RootEvtCnvSvc)

Option name	**Default value**	**Meaning**
DbType	""	Persistency technology (e.g. "ROOT")

Table B.13 Standard Options for the histogram service [HistogramPersistencySvc]

Option name	**Default value**	**Meaning**
OutputFile	""	Output file for histograms. Histograms not saved if not given.
RowWiseNTuplePolicy	"FLOAT_ONLY"	Persistent representation of NTuple data types. Other possible value is "USE_DATA_TYPES". See Section 10.2.3.2 for details
PrintHistos	false	Print the histograms also to standard output (HBOOK only)

Table B.14 Standard Options for the N-tuple service [NTupleSvc] (see Section 10.2.3.2)

Option name	Default value	Meaning
Input	{}	Input file(s) for n-tuples. Format: {“FILE1 DATAFILE='tuple1.typ' OPT='OLD' ”, [“FILE2 DATAFILE='tuple2.typ' OPT='OLD' ”,...]}
Output	{}	Output file(s) for n-tuples. Format: {“FILE1 DATAFILE='tuple1.typ' OPT='NEW'”, [“FILE2 DATAFILE='tuple2.typ' OPT='NEW'”,...]}
StoreName	"/NTUPLES"	Name of top level entry

Table B.15 Standard Options for the Event Collection service [TagCollectionSvc] (see Section 10.3.2)

Option name	Default value	Meaning
Output	{}	Output file specification. See Section 10.3.2 for details
StoreName	"/NTUPLES"	Name of top level entry

Table B.16 Standard Options for the standard event selector [EventSelector]

Option name	Default value	Meaning
Input	{}	Input data stream specification. Format: "<tagname> = ’<tagvalue>’ <opt>" Possible tags are different depending on input data type. For Event data, see Section 6.10.2 For Event Collections, see Section 10.3.2
FirstEvent	1	First event to process (allows skipping of preceding events)
PrintFreq	10	Frequency with which event number is reported

Table B.17 Event Tag Collection Selector [EventCollectionSelector]

The following options are used internally by the EventCollectionSelector. They should not normally be used directly by users, who should set them via the "tags" of the EventSelector.Input option			
Option name	**Corresponding tag of EventSelector.Input**	**Default value**	**Meaning**
CnvService	SVC	“EvtTupleSvc”	Conversion service to be used
Authentication	AUTH	""	Authentication to be used
Container		"B2PiPi"	Container name
Item		"Address"	Item name
Criteria	SEL	""	Selection criteria
DB	DATAFILE	""	Database name
DbType	TYP	""	Database type
Function	FUN	"NTuple::Selector"	Selection function

Table B.18 Standard Options for Random Numbers Generator Service [RndmGenSvc]

Option name	**Default value**	**Meaning**
Engine	“HepRndm::Engine<RanluxEngine>”	Random number generator engine
Seeds		Table of generator seeds
Column	0	Number of columns in seed table -1
Row	1	Number of rows in seed table -1
Luxury	3	Luxury value for the generator
UseTable	false	Switch to use seeds table

Table B.19 Standard Options for Particle Property Service [ParticlePropertySvc]

Option name	**Default value**	**Meaning**
ParticlePropertiesFile	“($LHCBDBASE)/cdf/particle.cdf”	Particle properties database location

Table B.20 Standard Options for Chrono and Stat Service [ChronoStatSvc]

Option name	**Default value**	**Meaning**
ChronoPrintOutTable	true	Global switch for profiling printout
PrintUserTime	true	Switch to print User Time
PrintSystemTime	false	Switch to print System Time
PrintEllapsedTime	false	Switch to print Elapsed time (Note typo in option name!)
ChronoDestinationCout	false	If true, printout goes to cout rather than MessageSvc
ChronoPrintLevel	3	Print level for profiling (values as for MessageSvc)

Table B.20 Standard Options for Chrono and Stat Service [ChronoStatSvc]

Option name	Default value	Meaning
ChronoTableToBeOrdered	true	Switch to order printed table
StatPrintOutTable	true	Global switch for statistics printout
StatDestinationCout	false	If true, printout goes to cout rather than MessageSvc
StatPrintLevel	3	Print level for profiling (values as for MessageSvc)
StatTableToBeOrdered	true	Switch to order printed table

B.1 Obsolete options

The following options are obsolete and should not be used. They are documented here for completeness and may be removed in a future release.

Table B.21 Obsolete Options

Obsolete Option	Replacement
EventSelector.EvtMax	ApplicationMgr.EvtMax (Table B.1)

Appendix C
Job Options Grammar and Error Codes

C.1 The EBNF grammar of the Job Options files

The syntax of the Job-Options-File is defined through the following EBNF-Grammar.

```
Job-Options-File =
   {Statements} .

Statements =
   {Include-Statement} | {Assign-Statement} | {Append-Statement} |
{Platform-Dependency} .

AssertableStatements =
   {Include-Statement} | {Assign-Statement} | {Append-Statement} .

AssertionStatement =
   '#ifdef' | '#ifndef' .

Platform-Dependency =
   AssertionStatement 'WIN32' <AsertableStatements> [ #else <Asserta-
bleStatements> ] #endif

Include-Statement =
   '#include' string .

Assign-Statement =
   Identifier '.' Identifier '=' value ';' .

Append-Statement =
   Identifier '.' Identifier '+=' value ';' .
```


```
Identifier =
   letter {letter | digit} .

value =
   boolean | integer | double | string | vector .

vector =
   '{' vectorvalue { ',' vectorvalue } '}' .


vectorvalue =
   boolean | integer | double | string .

boolean =
   'true' | 'false' .

integer =
   prefix scientificdigit .

double =
   ( prefix <digit> '.' [ scientificdigit ] ) |
   ( prefix '.' scientificdigit ) .

string =
   '"' {char} '"' .

scientificdigit =
   < digit> [ ( 'e' | 'E' ) < digit> ] .

digit =
   <figure> .

prefix =
   [ '+' | '-' ] .

figure =
   '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.

char =
   any character from the ASCII-Code

letter =

   set of all capital- and non-capital letter
```


C.2 Job Options Error Codes and Error Messages

The table below lists the error codes and error messages that the Job Options compiler may generate, their reason and how to avoid them.

Table 17.1 Possible Error-Codes

Error-Code	Reason	How to avoid it
Error #000 Internal compiler error	-	This code normally should never appear. If this code is shown there is maybe a problem with your memory, your disk-space or the property-file is corrupted.
Error #001 Included property-file does not exists or can not be opened	* wrong path in #include-directive * wrong file or mistyped filename * file is exclusively locked by another application * no memory available to open this file	Please check if any of the listed reasons occured in your case.
Warning #001 File already included by another file	The file was already included by another file and will not be included a second time. The compiler will ignore this #include-directive and will continue with the next statement.	Remove the #include-directive
Error #002 syntax error: Object expected	The compiler expected an object at the given position.	Maybe you mistyped the name of the object or the object contains unknown characters or does not fit the given rules.
Error #003 syntax error: Missing dot between Object and Propertyname	The compiler expect a dot between the Object and the Propertyname.	Check if the dot between the Object and the Propertyname is missing.
Error #004 syntax error: Identifier expected	The compiler expected an identifier at the given position.	Maybe you mistyped the name of the identifier or the identifier contains unknown characters or does not fit the given rules.
Error #005 syntax error: Missing operator '+=' or '='	The compiler expected an operator between the Propertyname and the value.	Check if there is a valid operator after the Propertyname. Note that a blank or tab is not allowed between '+='!

Table 17.1 Possible Error-Codes

Error-Code	Reason	How to avoid it
Error #006 String is not terminated by a “	A string (value) was not terminated by a “.	Check if all your strings are beginning and ending with “. Note that the position given by the compiler can be wrong because the compiler may thought that following statements are part of the string!
Error #007 syntax error: #include-statement is not correct	The next token after the #include is not a string.	Make sure that after the #include-directive there is specified the file to include. The file must be defined as a string!
Error #008 syntax error: #include does not end with a ;	The include-directive was terminated by a ;	Remove the ; after the #include-directive.
Error #009 syntax error: Values must be separated with ’,’	One or more values within a vector were not separated with a ’,’ or one ore more values within a vector are mistyped.	Check if every value in the vector is separated by a ’,’. If so the reason for this message may result in mistyped values in the vector (maybe there is a blank or tab between numbers).
Error #010 syntax error: Vector must end with ’}’	The closing bracket is missing or the vector is not terminated correctly.	Check, if the vector ends with a ’}’ and if there is no semicolon before the ending-bracket.
Error #011 syntax error: Statement must end with a ;	The statement is not terminated correctly.	Check if the statement ends with a semicolon ’;’.
Runtime-Error #012: Cannot append to object because it does not exists	The compiler cannot append the values to the object.propertyname because the object does not exist.	Check if the refered object is defined in one of the included files, if so check if you writed the object-name exactly like in the include-file.
Runtime-Error #013 Cannot append to object because Property does not exists	The compiler cannot append the values to the object.propertyname because the property does not exist.	Check if there was already something assigned to the refered property (in the include-file or in the current file). If not then modify the append-statement into a assign-statement. If there was already something assigned, check if the object-name and the property-name are typed correctly.

Table 17.1 Possible Error-Codes

Error-Code	Reason	How to avoid it
Error #014 Elements in the vector are not of the same type	One or more elements in the vector have a different type than the first element in the vector. All elements must have the same type like the first declarated element.	Check declaration of vector, check the types and check, if maybe a value is mistyped.
Error #015 Value(s) expected	The compiler didn't find values to append or assign	Check the statement if there exists values and if they are written correctly. Maybe this error is a result of a previous error!
Error #016 Specified property-file does not exist or can not be resolved	The compiler was not able to include a property-file or didn't found the file. A reason can be that the compiler was not able to resolve an environment-variable which points to the location of the property-file.	Check if you are using enviornment-variables to resolve the file, if they are mistyped (wether in the system or in the #include-directive) or not set correctly.
Error #017 #ifdef not followed by an identifier	The #ifdef-statement is not followed by the assertion-identifier (WIN32).	Add WIN32 after the #ifdef-statement.
Error #018 identifier in #ifdef / #ifndef not known	The assertion-identifier used in the #ifdef- /#ifndef-statement is not known. At the moment there can only be used WIN32!	Change identifier to WIN32.
Error #019 #ifdef / #ifndef / #else / #endif doesn't end with a ';'	A semicolon was found after the #ifdef- / #ifndef- / #else- / #endif-statement. These statements don't end with a semicolon.	Remove the semicolon after the #ifdef / #ifndef / #else / #endif-statement.

Appendix D
Design considerations

D.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 [6] or specialized literature.

D.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.

D.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.

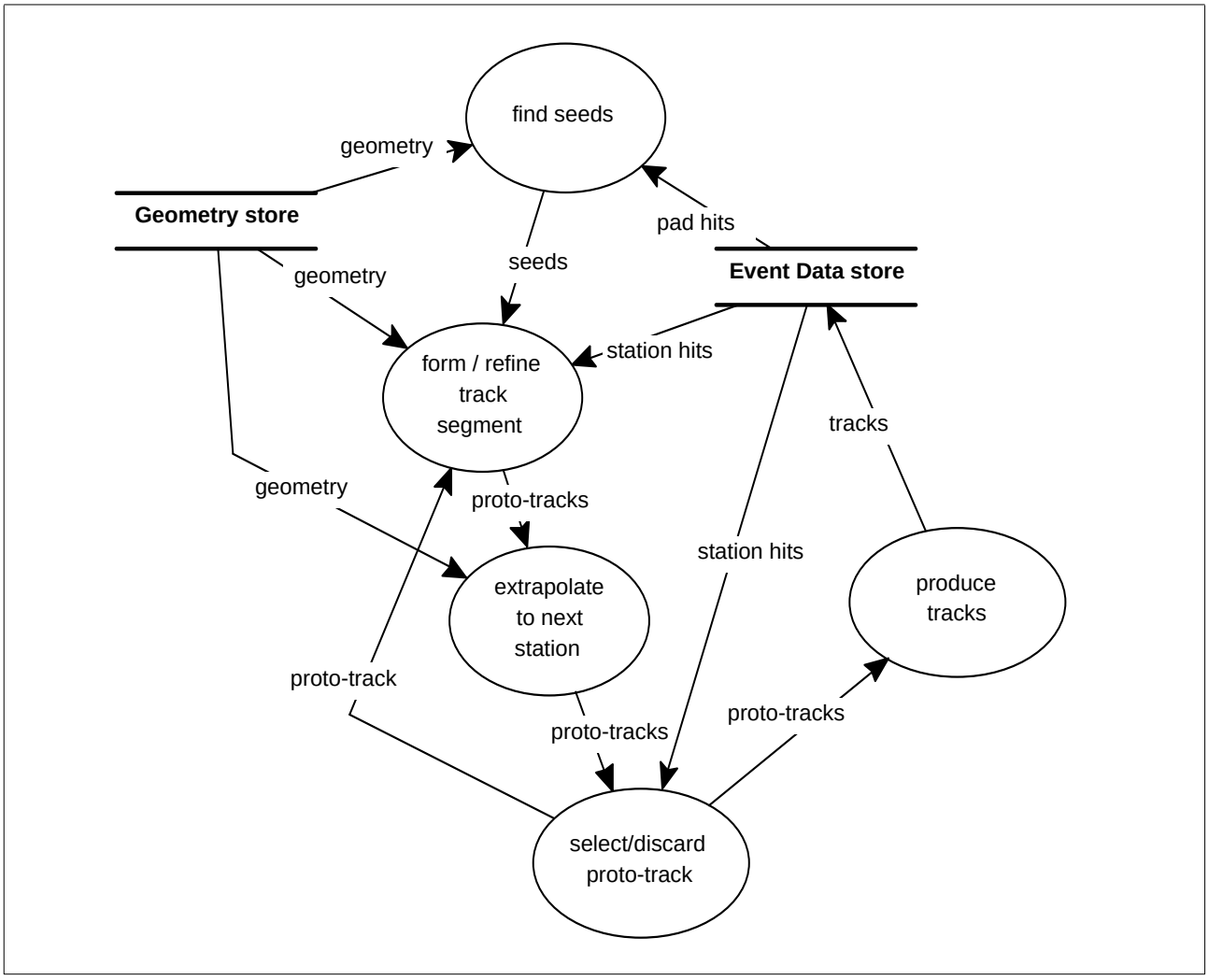

Figure A.1 Hypothetical decomposition of a tracking algorithm based on a Kalman filter using a Data flow Diagram

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.

D.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.

D.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 view point 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 bootstraping 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.

D.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 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.

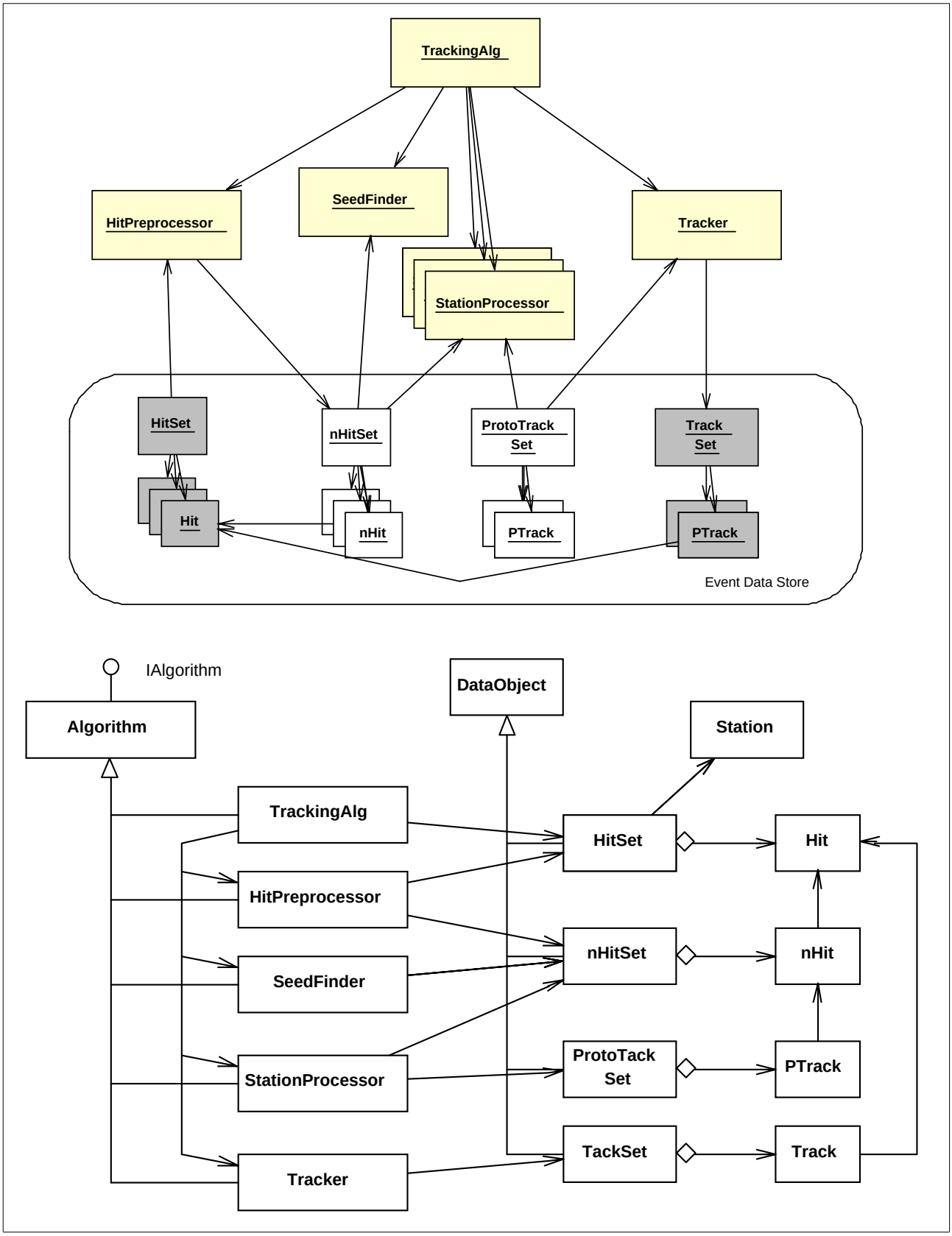

Figure A.2 Object diagram (a) and class diagram (b) showing how the complete example tracking algorithm could be decomposed into a set of specific algorithms that collaborate to perform the complete task.

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.

D.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.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W