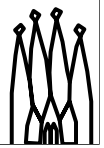
5

Algorithm Tools: Overview



Schedule:	Timing	Topic
	20 minutes	Lecture
	0 minutes	Practice
	20 minutes	Total

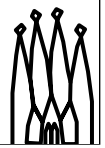
Lesson Goals

Introduce the concept of Algorithm Tool

- **Separation of roles: Algorithm & Data**
- **Understanding the differences between Services, Algorithms and Tools**

Design Issues

- **Use cases**
- **Some examples**

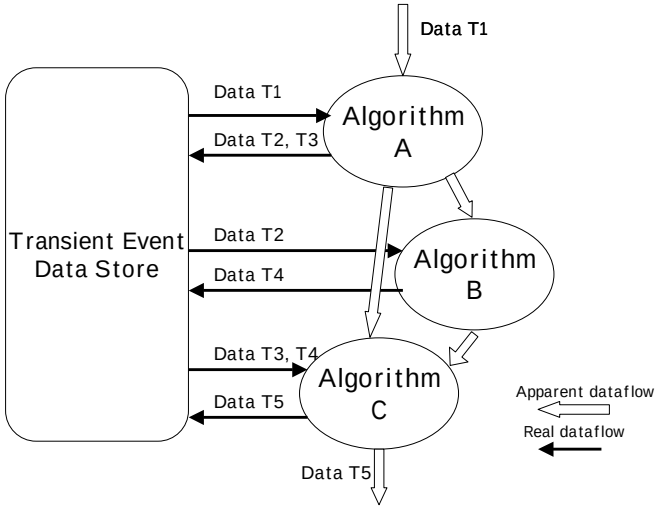


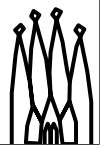
Goals

The goal of this first lesson in Algorithm tools is to introduce this new concept. Understanding the differences between with other components of the framework (Services, Algorithms) is important for making consistent designs across the experiment.

Some cases and examples will be studied in this lesson.

Algorithms



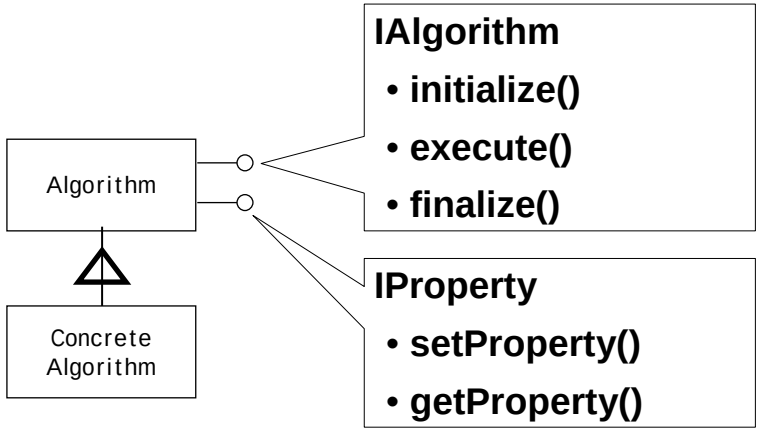


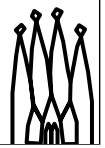
Algorithm communication and execution scheduling

This is to remind that the only way to communicate input and output data to the Algorithms is always achieved by accessing by themselves the transient data store. Input data is located in the transient store and when the algorithm has finished processing it registers in the stone new data that can then be used by other algorithms. Scheduling the algorithms in the adequate order we can obtain the desired data flow.

This communication and scheduling is very simplistic but it allows to minimize the coupling between algorithms, the transient data store acts a “black-board” between algorithms. An Algorithm does not need to know from where the data has been produced, it only knows what data it requires and what data will be producing.

Algorithm Interface





Algorithm Interfaces

Two interfaces are implemented in all algorithms. These interfaces are quite simple but they are sufficient for incorporating any implementing them into any Gaudi application (Brunel, DaVinci, etc.).

- The IAlgorithm interface allows the framework to control the execution of the different algorithm. The methods in this interface not have any argument, therefore any input or output data has to be given to the algorithm by transient stores (event, detector, statistics, etc.)
- The IProperty interface allows the framework to configure the algorithm with any property value. In particular, the JobOptions service that reads the joboptions files at the start of a job sets all the algorithm properties using that interface. In general, any service or algorithm of an application can also set and get properties of any algorithm or service at run-time.

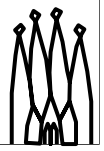
Algorithm Scheduling

Simple and explicit scheduling

- **List of Algorithms**
  - **Primarily given by the property *ApplicationMgr.topAlg***

Algorithms are executed only once per event

- **Several instances of the same algorithm class are possible**



Algorithm Scheduling

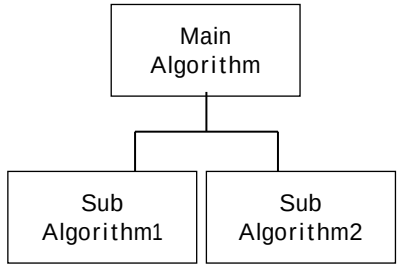
On the first approximation the scheduling of Algorithms is controlled by the property “topAlg” of the Application Manager. This property is list of the “top” level Algorithms. They will be executed by the Application Manager (EventLoopMgr) in the order they have been declared.

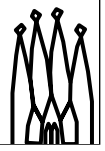
The execution of Algorithm is done by calling the “execute()” method once per event.

Algorithm Hierarchies

An Algorithm can have Sub-Algorithms

- **Parent Algorithm is responsible for**
  - **Creation**
  - **Execution**
- **Framework does initialization and finalization**





Algorithms Hierarchies

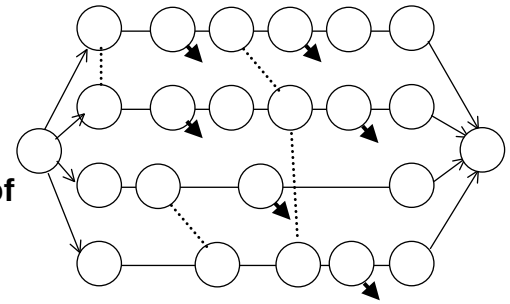
Algorithms can be organized in hierarchies to allow more complex implicit scheduling. Any algorithm can have sub-algorithms associated. These sub-algorithms are created by the “parent” Algorithm (eventually controlled by job options) and they are executed under the control of the parent algorithm.

The Framework takes care of the initialization and finalization of sub-algorithms.

Sequences

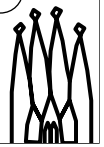
Sequences of *Algorithms*

- **Avoid re-calling same algorithm on same event**
- **Different instances of the same algorithm possible**



Event filtering

- **Avoid passing all the events through all the processing chain**



Sequences & 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.

Algorithm life-cycle

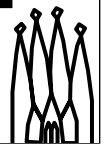
Algorithms are created by a factory

```
static const AlgFactory<HelloWorld>  Factory;
const IAlgFactory& HelloWorldFactory = Factory;
```

Run-type configuration

- **List of algorithms and their properties**

```
ApplicationMgr.topAlg = {'HelloWorld'};
HelloWorld.OutputLevel = 4;
```

Algorithm creation

Algorithms are created using the "factory" design pattern. Using factories, opposite to using the new() operator directly, the creator of the Algorithm does not need to know the concrete type. Technically this means that the header file containing the defining of the concrete type no not need to be included in the creators code.

The way to achieve this is by instantiating a static object that know to create an instance (the factory). The convention is to use a the templated class AlgFactory<T> for that purpose.

Similarly to what is done with Algorithm, the Algorithms Tools will need to instantiated using factories.

Algorithm configuration

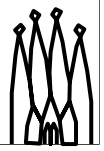
Algorithm are configured using the job options

Services

Are used by Algorithms to help them to perform their work

Are setup and initialized at the beginning of a job by the framework and used by many algorithms as often as necessary

They do not have a “state”



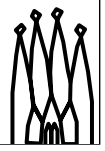
Services

Services are used by Algorithms to help them to perform their work. Services are initialized at the beginning of the job and are used by many Algorithms. This implies that Services in general can not keep an state if they are used by several Algorithms.

The Problem

Sometimes in an Algorithm it is necessary to execute the same operation more than once per event, only for some events, on specific non identifiable data objects, producing new objects that will be put in the Event Store later

➔ Needed to introduce a new concept



The need for Algorithm Tools

When implementing an Algorithm, very often it is necessary to perform some operations or complex processing for each object in a list (e. tracks in the track collection). Therefore, the concept of sub-algorithm is not adequate for that purpose. In this kind of operations is more efficient to pass the pass the data as arguments instead of registering and retrieving from the transient store. In addition, it can be that the same kind of operation can be re-used in other algorithms.

Algorithm Tools

The concept of Algorithms Tools, being a sort of simple algorithms callable many times and with arguments, was introduced in Gaudi

- **Examples**
  - **Vertexing**
  - **Track transport**
  - **Association to truth**
  - **Selection of particles based on a pID CL**

Designing for Re-use

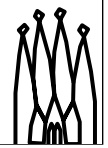
Algorithm Tools are useful small algorithms that can be packages in a way that will be easy to re-use them in other Algorithms. They are callable many times during the execution of an event and the user can pass arguments.

Examples:

- Vertexing (to produce one or many vertexes from a list of tracks or particle candidates)
- Track transport (to obtain the track parameters on other points of the detector)
- Association to truth (to obtain the Monte Calo information corresponding to a reconstructed object)
- Selection from a container of objects (to reduce the a list ob objects according to some selection criteria)

Requirements

- **An Algorithm requires a Tool on a per need base**
- **The ToolSvc checks if the requested type of Tool is available and returns an instance of the requested type after configuring it**
- **An Algorithm can have a private instance of a Tool that can be configured according to its need via JobOptions**
- **An Algorithm can use a Tool "as-it-is", in this case the ToolSvc will provide a "common" Tool (shared-Tool)**
- **An Algorithm should be able to use a Tool without worrying about the implementation**
- **The ToolSvc keeps track of the Tools instantiated and if a tool exists it does not re-create it**
- **An Algorithm can tell the ToolSvc if it will not need a tool anymore**
- **The ToolSvc will decide which tools instances can be deleted (always done at finalization of the Service)**
- **Not only Algorithms would want to use Tools (also Services …)**

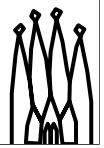


ToolSvc

The ToolSvc is the service that manages Algorithm Tools (private or shared)

- **Algorithms ask the ToolSvc for a give Tool by name**
- **Manages the life-cycle of Tools**
- **Keeps track of existing Tools**

Tools can be configured using the JobOptions as Algorithms or Services

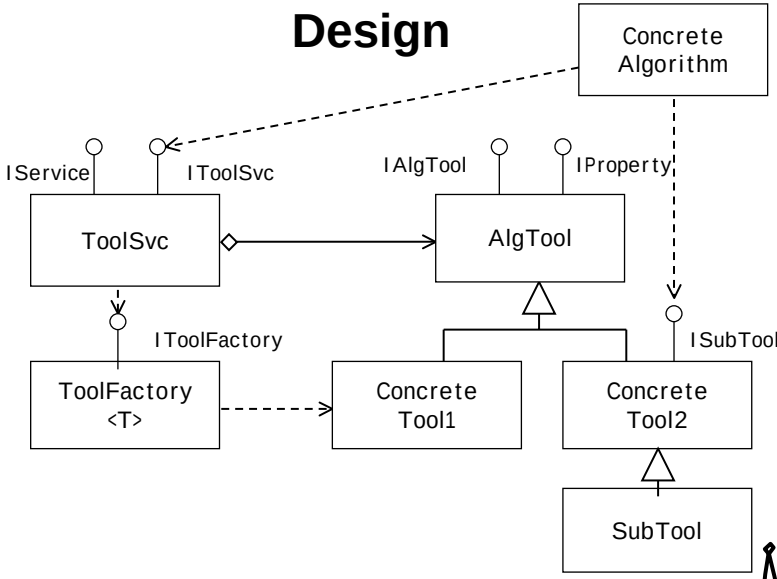


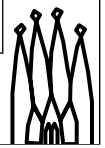
The ToolSvc Service

This service is managing Algorithm Tools. It is the service in charge of tools in their life-cycle, it creates them, configures them and destroys them at the finalize phase of the job.

An Algorithm requests the ToolSvc to obtain a reference to a Tool by its name. Tools can be private or shared. The idea is that if a Tool requires a fair amount of resources (memory, cpu time for configuration) does makes sense to share the Tool among the various Algorithms that may require this functionality. The problem with a shared tool is that is can not keep an state between invocations since it is not guaranteed that other Algorithms may have used it meanwhile.

Design



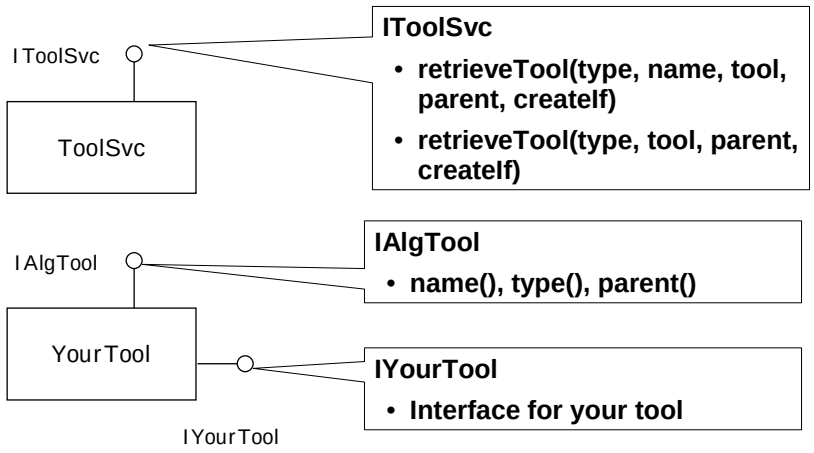


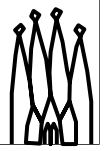
Design

The Design following closely the Algorithm one. We have a base class (AlgTool) from which all the Tools inherits from. This base class ensures that the Tool is manageable and configurable from the TooSvc service and JobOptions service. Concrete Tools can have their own interface specidif to the tool or class of tools. Of course, specializations can always be possible by inheritance of concrete tools.

The ToolSvc uses the factory pattern to instantiate concrete tools without known about them using the generic abstract interface IToolFactory which is implemented by the template class ToolFactory. The service keeps the list of AlgTools existing in the program.

Tools Interfaces





Interfaces

•**IToolSvc**. This is the interface implemented by the TooSvc that allows Algorithms to locate or create new Tools.

•**IAlgTool**. Basic interface that any AlgTool implements and is used for bookkeeping purposes of the ToolSvc.

•**IYourTool**. This represents the interface (abstract) for this particular tool or class of tools. For example Associators will have a common interfaces regardless of their implementation. See next slide.

AlgTools life-cycle

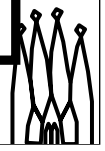
AlgTools are created by a factory

```
static const ToolFactory<VertexSmearer> Factory;
const IAlgFactory& VertexSmearerFactory = Factory;
```

Run-type configuration

- **Convention to name AlgTools**

```
RecPrimaryVertex.VertexSmearer.OutputLevel  = 3;
RecPrimaryVertex.VertexSmearer.dxVtx = 0.009;
RecPrimaryVertex.VertexSmearer.dyVtx = 0.009;
RecPrimaryVertex.VertexSmearer.dzVtx = 0.038;
```

AlgTool creation and Configuration

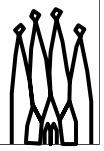
AlgTools are created and configured the same way as Algorithms

Algorithm Tools Types

Tool Taxonomy

- Similar tools could implement the same interface
- Different implementations

This could offer a way to evolve and improve Tools



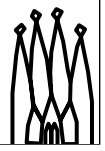
Tools classification

The idea is to classify tools for their functionality and try to define interfaces that are general enough to be applicable to a number of them. In this way we can have different implementation ranging from very simple ones to a more sophisticated ones.

Example: Associator

One possible tool category is an Associator for relating reconstruction objects to the “MC” information

- The interface can be very generic
  - Input: *ContainedObject*
  - Output: list of *ContainedObject*
- The actual navigation in the Event Model and association criteria could be very specific



Associator Example

The Associator is an example of a class of AlgTools. These kind of tools have the role to associate a given reconstruction object (track, hit, candidate, etc.) to the Monte Carlo one. This association involves navigation on the Event Model with various hops in the most general case. In addition, very often the association is not unique and requires some physics criteria to select the best one. This is an example in which the definition of a generic abstract interface makes real sense. This interface can be made quite generic involving base classes like the *ContainedObject*, which is the class that all the objects that we have collections of them inherit from.

Summary

AlgTools have been introduced to overcome some limitations of Algorithms and with the desire of re-use

Very similar to Algorithms for configuration and life-cycle

➔ **Lesson 6: Algorithm Tools in Practice**