8

# Writing Tools: advanced

Gaudi Framework Tutorial, 2001

| Schedule: | Timing | Topic |
|---|---|---|
| | 20 minutes | Lecture |
| | 30 minutes | Practice |
| | 20 minutes | Total |

## Objectives

**After completing this lesson, you should be able to:**

- **Understand tools interfaces**
- **Define and implement a tool**

8-2 Gaudi Framework Tutorial, 2001

**Lesson Aims**

Different ways of performing an operation can be implemented by different type of tools with the same interface. Tools' interfaces will be described.

The second goal of this lesson is to learn the necessary steps to implement a tool.

## Algorithm Tools Types

- Different tools can implement the same functionality
- Since the algorithms interacts with them only through the interface they are interchangeable
- The choice can be done via the job options at run time
- This offers 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.

Since algorithms using this category of tools interact with them only through their interface they are interchangeable. In fact the choice is done changing the string specifying the tool type in the retrieveTool method. If this string is a property of the algorithms the concrete tool used can be chosen at run-time via the job options, as in the tutorial example.

When knowing a priori that there will be concrete tools with a common *functional* interface (like vertexers, associators, etc.) it is worth to ask if they will have common properties or methods and implement them in a base class.

For the scope of the tutorial the tools are in component libraries, without bothering with different type of libraries.

In reality for tools things are a little more complicated: the interfaces and base classes (for any tool you will want to allow various implementations) should be in a linker library, while tools concrete implementations and algorithms using them should be in a (many) component library.

## Tools Specific Interfaces

- A tool must have additional interface based on its functionality
  - This additional interface must inherit from the IAlgTool interface
  - Tools performing a similar operation in different ways will share the same interface (ex. Vertexer)

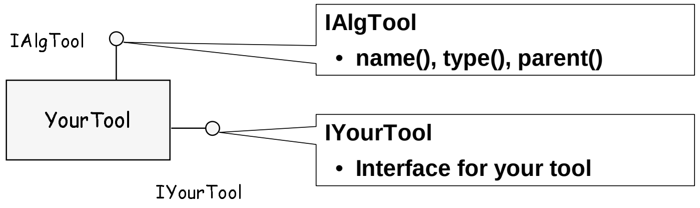
Remember: The implementation of the IAlgTool interface is done by the AlgTool base class, you don’t need to worry about it

**Tools Specific Interfaces**

Interfaces based on tool functionality must be defined. This is the interface with which Algorithms interact. Many tools can perform similar operation in different ways but with the same well defined protocol. For example fitting a vertex from a list of particles can be done in more than one way but it will always require a list of particles and return a vertex.

In order for a tool to interact with the ToolSvc via this additional interface, the interface itself has to inherit from IAlgTool. The implementation of the IAlgTool interface is done in the AlgTool base class and does not have to be implemented in concrete tools.

## Tools Interfaces



**Interfaces**

•**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.

## How to write concrete tools

**When encapsulating some reoccurring functionality in a tool you need to:**

- **Identify the tool functionality and define its special interface: *ITool.h***
  - Unless the interface already exist
- **Inherit from the AlgTool base class in *MyTool.h***
- **Implement the tool specific functionality in *MyTool.cpp***

**How to write concrete tools: header file**

By inheriting from the *AlgTool* Base class, concrete tool are managed by the *ToolSvc*. The base class in fact, implements the *IAlgTool* interface that is the protocol used by the ToolSvc to interact with tools.

Tools can be configured in the constructor or via the *job options*. Tools could need to use services, for this reason the AlgTool base class provides a serviceLocator() method to help in retrieving the necessary services.

Access to the Message Service is also provided.

## Tools interfaces in practice

```
#include “GaudiKernel/IAlgTool.h”
static const InterfaceID
    IID_IMCAcceptanceTool(“IMCAcceptanceTool”, 1, 0)

 class IMCAcceptanceTool : virtual public IAlgTool {
 public:
   /// Retrieve interface ID
   static const InterfaceID& interfaceID() {
      return IID_IMCAcceptanceTool;
   /// + specific signature
   virtual bool accepted( const MCParticle* mother ) = 0;
   }
 }
```

Necessary for Inheritance

Unique interface ID

Pure virtual method(s)

**Tools interfaces In practice**

A tool additional interface has to conform to the rules of a Gaudi interface.

It will have only pure virtual methods, with the exception of the static method InterfaceID. This method returns a unique interface identifier to be used by the query interface mechanism.

See *Gaudi User Guide* for more details.

## AlgTool inheritance

**A concrete tool will inherit from the AlgTool base class:**

- has properties
- serviceLocator()
- msgSvc()
- the IAlgTool interface is implemented
- possible initialize(), finalize()

Configurable via job options

Retrieve necessary services

Called after creation by ToolSvc

Configured at creation

**How to write concrete tools: header file**

By inheriting from the *AlgTool* Base class, concrete tool are managed by the *ToolSvc*. The base class in fact, implements the *IAlgTool* interface that is the protocol used by the ToolSvc to interact with tools.

Tools can be configured in the constructor or via the *job options*. Tools could need to use services, for this reason the AlgTool base class provides a serviceLocator() method to help in retrieving the necessary services.

Access to the Message Service is also provided.

## AlgTools life-cycle

**AlgTools are created by a factory**

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

**This must be instantiated in the implementation file**

•As for Algorithms but ToolFactory

8-9 Gaudi Framework Tutorial, 2001

**AlgTool creation and Configuration**

AlgTools are created and configured the same way as Algorithms, using the “factory” design pattern. Using factories, opposite to using the new() operator directly, the creator of the AlgTool does not need to know the concrete type. Technically this means that the header file containing the defining of the concrete type does not need to be included in the creators code.

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

## How to write concrete tools: implementation file

**Declare in constructor specific properties**

**Get services necessary for implementation in constructor or initialize method**

– Very similarly to Algorithms

**Implement necessary functionality**

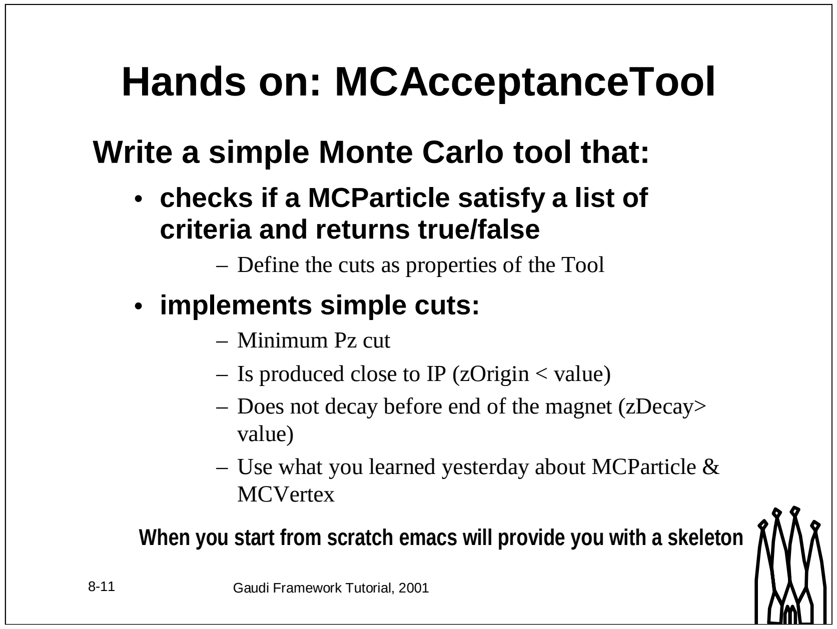
– Additional methods specific to the tool

8-10 Gaudi Framework Tutorial, 2001

**How to write concrete tools: implementation file**

Anything that need to be held through the lifetime of a tool has to be set in the constructor or the initialize method. While properties must be declared in the constructor, pointers to necessary services can be set in either one. If reset mechanisms are implemented their management has to be taken care of by the tool.

The necessary functionality of a tool is implemented in additional methods, this methods can be executed as often (or rarely) as deemed necessary by the algorithm using the tool.



## Hands on: MCAcceptanceTool

**Write a simple Monte Carlo tool that:**

- **checks if a MCParticle satisfy a list of criteria and returns true/false**
  - Define the cuts as properties of the Tool
- **implements simple cuts:**
  - Minimum Pz cut
  - Is produced close to IP (zOrigin < value)
  - Does not decay before end of the magnet (zDecay> value)
  - Use what you learned yesterday about MCParticle & MCVertex

**When you start from scratch emacs will provide you with a skeleton**

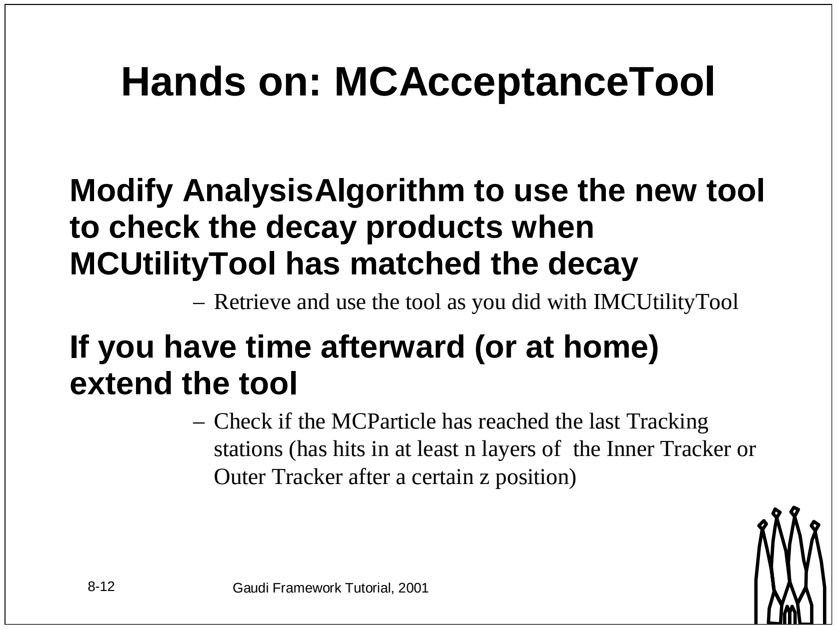
**Hands on**

In the following exercise we will write a simple Monte Carlo utility tool using some of the things learned in the Gaudi Basics Tutorial.

Eventually we will use the tool in an Algortihm.

We will need to

- Look at the interface methods in
  Components/IMCAcceptanceTool.h
- Write the tool itself
  MCAcceptanceTools.h, MCAcceptanceTools.cpp
- Modify the algorithm that retrieves and uses the tool
  AnalysisAlgorithm.h, AnalysisAlgorithm.cpp



## Hands on: MCAcceptanceTool

**Modify AnalysisAlgorithm to use the new tool to check the decay products when MCUtilityTool has matched the decay**

- Retrieve and use the tool as you did with IMCUtilityTool

**If you have time afterward (or at home) extend the tool**

- Check if the MCParticle has reached the last Tracking stations (has hits in at least n layers of the Inner Tracker or Outer Tracker after a certain z position)

## Hands on: IMCAcceptanceTool protocol

- The IMCAcceptanceTool has to be located in the subdirectory Components
  - Follows the Gaudi(LHCb) conventions
  - For your convenience a very simple one has been prepared for you

```
virtual bool accepted( const MCParticle* mother ) = 0;
```

Pure virtual methods

8-13 Gaudi Framework Tutorial, 2001

**IMCAcceptanceTool**

The location of the file follows the packaging convention for public include files.

The interface file has been provided for you, look through it to see what you would have to do to implement a new tool's interface

## Hands on: MCAcceptanceTool.h

- Inherit from IMCAcceptanceTool

```
class MCAcceptanceTool : public AlgTool,
                         virtual public IMCAcceptanceTool {
```

- Declare the interface method(s)

```
virtual bool accepted( const MCPartticle* mother );
```

- Remember to include the data members

```
double m_minPz;          ///< Momentum cut
double m_mazZvertex;     ///< Close to IP
double m_zMag;           ///< Does not decay before
double m_zLastHit;       ///< To extend: Zhit .gt.
int    m_nLayers;        ///< nLayers with Zhit .gt.
IDataProviderSvc* m_EDS; ///< To access the hits
```

8-14 Gaudi Framework Tutorial, 2001

## Hands on: MCAcceptanceTool.cpp

**Remember to include the necessary headers**

To handle errors in constructor

```
#include "GaudiKernel/GaudiException.h"
#include "GaudiKernel/ISvcLocator.h"
#include "GaudiKernel/IDataProviderSvc.h"
#include "GaudiKernel/SmartDataPtr.h"
#include "LHCbEvent/MCParticle.h"
#include "LHCbEvent/MCVertex.h"
#include "LHCbEvent/MCTrackingHit.h"
#include “CLHEP/Units/PhysicalConstants.h”
```

8-15 Gaudi Framework Tutorial, 2001

**Note:**

Tools can be configured only in the constructor. This implies that Status Codes cannot be returned in case of problems during the configuration, but you must throw an exception. The ToolSvc takes care of handling these exceptions and to inform the algorithm requesting the tool of the failure.

GaudiExceptions allow to provide additional information so that appropriate error messages can be printed by the ToolSvc.

## constructor

### Declare specific Interface(s)

```
declareInterface< IMCAcceptanceTool >(this);
```

### Declare Properties

```
declareProperty( “MinPz”,      m_minPz );
declareProperty( “MaxZVertex”, m_mazZvertex );
declareProperty( “Zmagnet”,    m_zMag);
declareProperty( “ZLastHit”,   m_zLastHit);
declareProperty( “MinNLayers”, m_nLayers);
```

### Set their default values

```
MCAcceptanceTool::MCAcceptanceTool( const std::string& type,
                                    const std::string& name,
                                    const IInterface* parent )
  : AlgTool( type, name, parent ),
    m_minPz( 0.0 * GeV ), …
```

8-16 Gaudi Framework Tutorial, 2001

## accepted

```
bool MCAcceptanceTool::accepted( const MCParticle* mcpart )  {
  /// Momentum cut (Pz)
  if ( mcpart->fourMomentum().z() < m_minPz ) { return false; }
  /// Particles are produced close to the interaction point
  const MCVertex* vOrigin = mcpart->originMCVertex();
  if ( 0 == vOrigin ) { return false; }
  double mcZpv = vOrigin->position().z();
  if ( fabs(mcZpv) > m_maxZVertex ) { return false; }
  /// Particles do not dissapear before the end of the magnet
  const SmartRefVector<MCVertex>& vDecay = mcpart->decayMCVertices();
  double zDecay;
  if ( vDecay.size() > 0  ) {
    zDecay = vDecay[0]->position().z();
  } else { zDecay = 20. * m; }
  if ( zDecay < m_zMag ) { return false; }
  return true;
}
```

## Hands on: using the new tool

```
...AnalysisAlgorithm.h
  std::string         m_newToolType;  ///< Tool type
  IMCAcceptanceTool*  m_pAccTool;;    ///< Reference to tool

...AnalysisAlgorithm::initialize()...
  sc = toolSvc()->retrieveTool( m_newToolName, m_pAccTool );

...AnalysisAlgorithm::execute()...
  if ( result ) {
    // Check if daughters the acceptance cuts using the tool
    const SmartRefVector<MCVertex>& decays=(*ipart)->decayMCVertices();
    SmartRefVector<MCVertex>::const_iterator ivtx;
    for ( ivtx = decays.begin(); ivtx != decays.end(); ivtx++ ) {
      const SmartRefVector<MCParticle>& daughters =
              (*ivtx)->daughterMCParticles();
      SmartRefVector<MCParticle>::const_iterator idau;
      int nAccDaug = 0;
      for(idau = daughters.begin(); idau != daughters.end(); idau++ ) {
        bool mcAcc = m_pAccTool->accepted( *idau );
        if ( mcAcc ) { nAccDaug++; }
      }
      log << MSG::INFO << "Number of accepted daughters = "
          << nAccDaug << endreq;
  }
```

## ! Extending the tool : constructor or initialize

- **Retrieve the Event Data Service**

```
m_EDS = 0;
StatusCode sc=serviceLocator()->service("EventDataSvc",
                                        m_EDS, true);
```

- **In case of problems in constructor throw a GaudiException**

```
throw GaudiException("EventDataSvc not found",
                  "ToolException", StatusCode::FAILURE);
```

## Extending the tool: accepted

```
accepted()...
if ( zDecay < m_zMag ) { return false; }
/// Is there a Tracking hit (IT or OT) after the min z position ?
/// Retrieve IT & OT Hits
typedef ObjectVector<MCTrackingHit>   TrHits;
SmartDataPtr<TrHits> itHits(eventSvc(),"/Event/MC/MCInnerTrackerHits");
if( 0 == itHits ) {
  log << MSG::ERROR << "Unable to retrieve the IT data <<endreq;
  return false;
}
SmartDataPtr<TrHits> otHits(eventSvc(), "/Event/MC/MCOuterTrackerHits" );
if( 0 == otHits ) { … }
// Check IT hits
int nITHits = 0;
for( TrHits::const_iterator iHit=itHits->begin(); iHit!=itHits->end(); iHit++ ) {
  if ( (*iHit)->mCParticle() == mcpart ) {
    if ( (*iHit)->entry().z() > m_zLastHit ) { nITHits++; }
  }
}
… the same for OT hits …
int nTrackerHits = nITHits + nOTHits;
if( nTrackerHits >= m_nLayers ) { return true; }
else { return false; }
```