6

Writing Physics Tools

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:

- **Define and implement a tool**
- **Retrieve and use tools in an algorithm**
- **Use private instances of tools**
- **Extend existing tools**

Lesson Aims

Different physics analysis often perform the same operation on similar objects. Typical examples are making a vertex from a list of particles, swimming the parameter of a particle to a specified position, associating a reconstructed particle with Monte Carlo generator particles.

Tools allow to share this code between separate algorithms. Instances of tools are also shared at run-time minimizing the creation of new objects. Algorithms request tools on a per need basis to the Tool Service that provides to their management.

Different ways of performing an operation can be implemented by different type of tools with the same interface. It will be shown how to write a tool interface.

Tools properties can be controlled via *job options*, when a tool is required to have a special configuration a private instance of the tool can be used.

When different type of tools with the same interface exist, the choice of the concrete tool can be done at run-time provided the Algorithm interacts with the tool via this interface.

Caveat

Things that will be used in the tutorial and are assumed to be know:

- **How to write a simple algorithm**
- **How to print**
- **How to use Job Options**
- **How to use Particle Properties and Particle Property Service**

Caveat:

The aim of the tutorial is to learn how to write and use tools.

Some of the topics covered in the “Gaudi Basics Tutorial” are assumed to be known and are consequently used.

How to write concrete tools: header file

A concrete tool will inherit from the AlgTool base class:

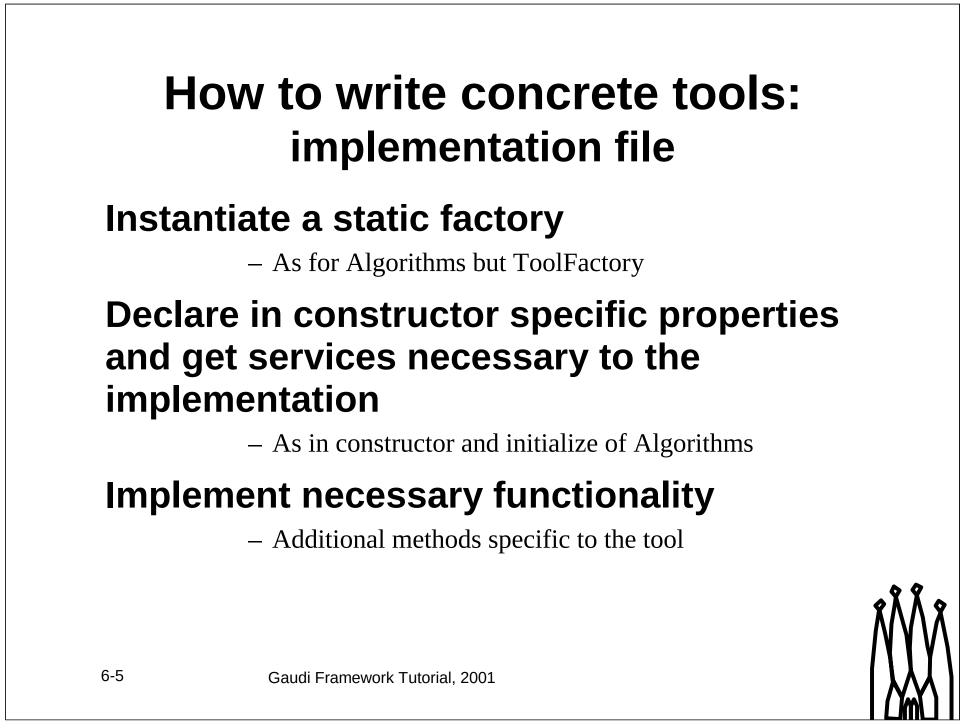
- **no initialize(), finalize()** — Configured at creation
- **serviceLocator()** — Retrieve necessary services
- **msgSvc()**
- **has properties** — Configured via job options
- **the IAlgTool interface is implemented**

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.



How to write concrete tools: implementation file

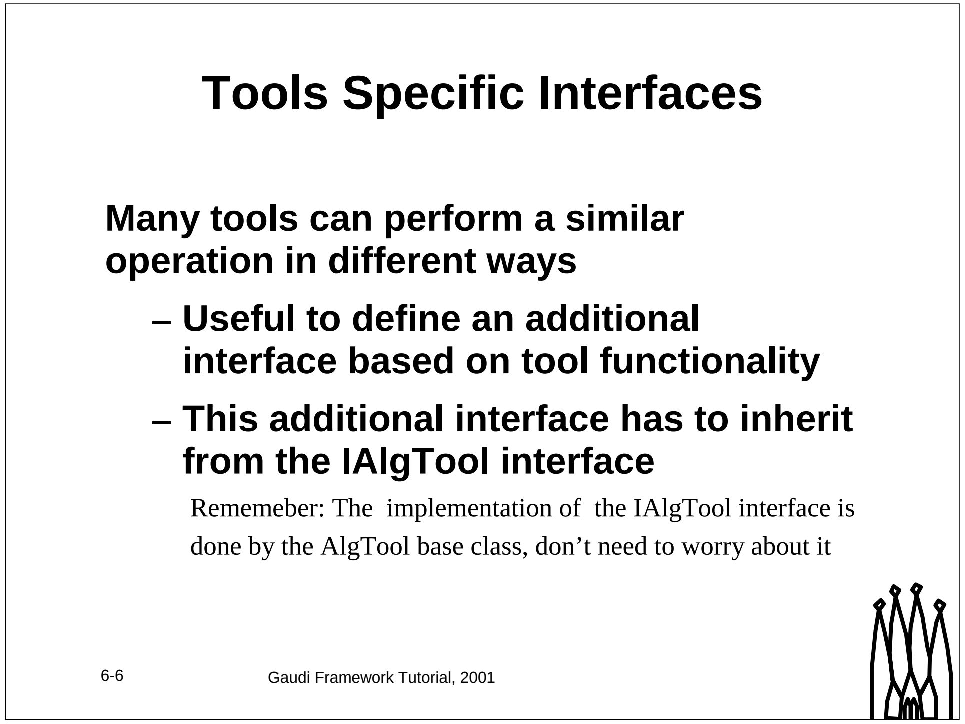
- **Instantiate a static factory**
  - As for Algorithms but ToolFactory
- **Declare in constructor specific properties and get services necessary to the implementation**
  - As in constructor and initialize of Algorithms
- **Implement necessary functionality**
  - Additional methods specific to the tool

How to write concrete tools: implementation file

Concrete tools, as algorithms are instantiate using a factory method. This allows new tools to be introduced at any time without having to include all their headers file in the ToolSvc.

Anything that need to be held through the lifetime of a tool has to be set in the constructor: this include properties as well as pointers to necessary services. If reset mechanisms are implemented their management has also to be taken care of.

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.



Tools Specific Interfaces

- **Many tools can perform a similar operation in different ways**
  - **Useful to define an additional interface based on tool functionality**
  - **This additional interface has to inherit from the IAlgTool interface**

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

Tools Specific Interfaces

Many tools can perform similar operation in different ways but with the same well defined protocol.

When useful, interfaces based on tool functionality can be defined. 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 in practice

Necessary for Inheritance

```
#include "GaudiKernel/IAlgTool.h"
static const InterfaceID IID_IMCUtilityTool(440, 1, 0)

class IMCUtilityTool : virtual public IAlgTool {
public:
  /// Retrieve interface ID
  static const InterfaceID& interfaceID() {
     return IID_IMCUtilityTool;
  }
}
```

Unique interface ID

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.

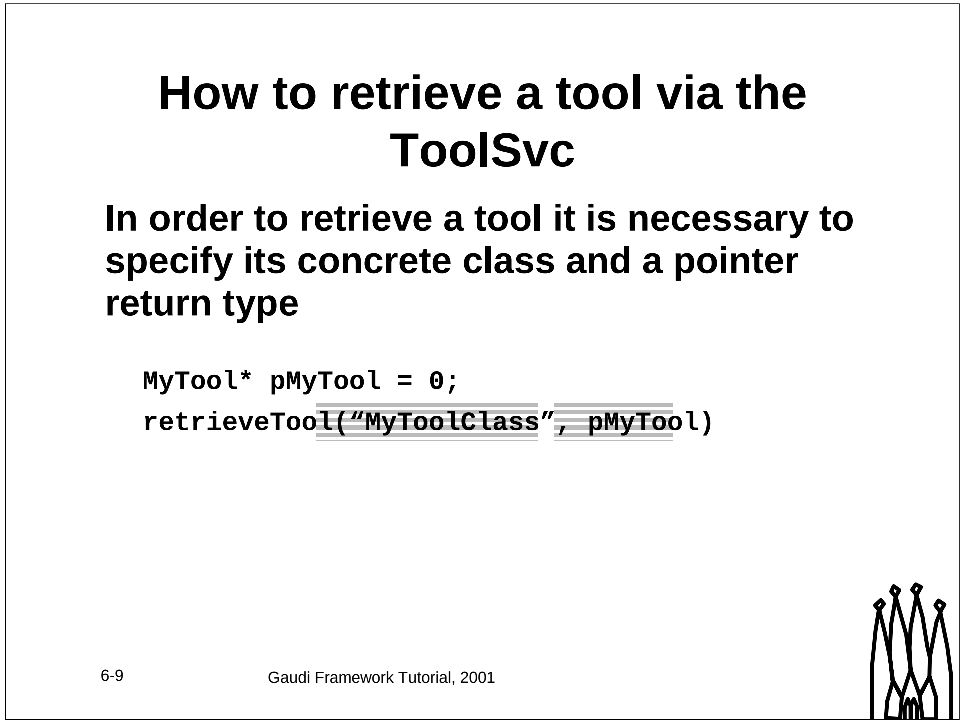
Requesting and using tools

- Algorithms (or Services) request a specific tool to the ToolSvc
- All tool management is done by the ToolSvc
  - Creates tools on first request
  - Holds all instances and dispatches requested tool
- Algorithm can keep a pointer to a tool and use it when necessary

Requesting and using tools

Algorithms, services or tools themselves request specific instances of tools to the ToolSvc on a per need basis.

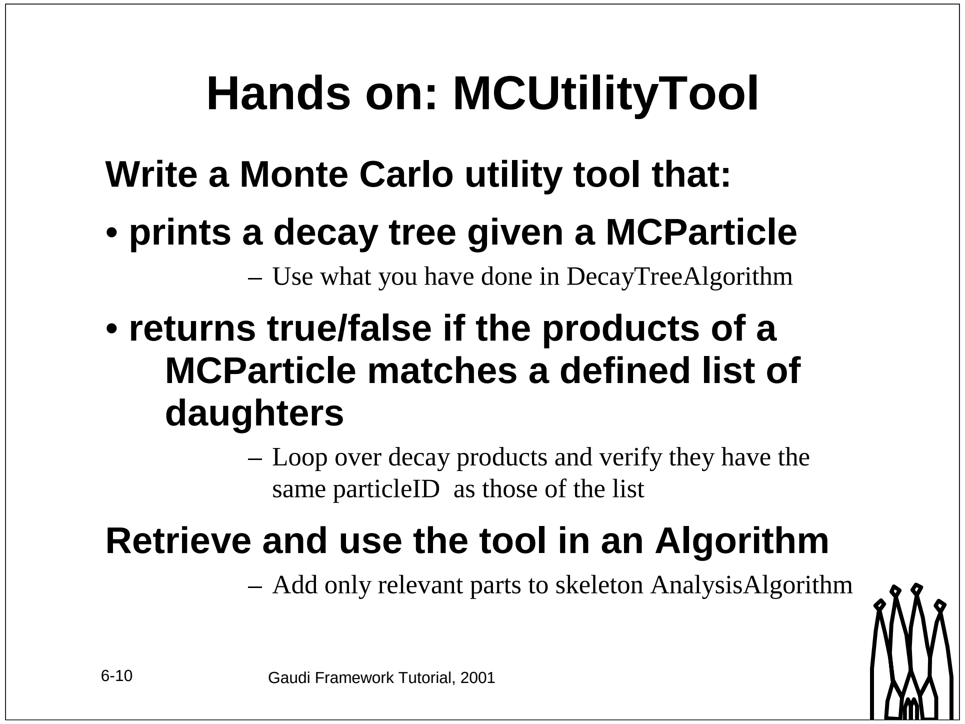
The ToolSvc creates them upon the first request, holds all the existing instances and dispatch the requested tool to the algorithms that require it.



How to retrieve a tool via the ToolSvc

In order to retrieve a tool it is necessary to specify its concrete class and a pointer return type

```
MyTool* pMyTool = 0;
retrieveTool(“MyToolClass”, pMyTool)
```



Hands on: MCUtilityTool

Write a Monte Carlo utility tool that:

- **prints a decay tree given a MCParticle**
  - Use what you have done in DecayTreeAlgorithm
- **returns true/false if the products of a MCParticle matches a defined list of daughters**
  - Loop over decay products and verify they have the same particleID as those of the list

Retrieve and use the tool in an Algorithm

- Add only relevant parts to skeleton AnalysisAlgorithm

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.

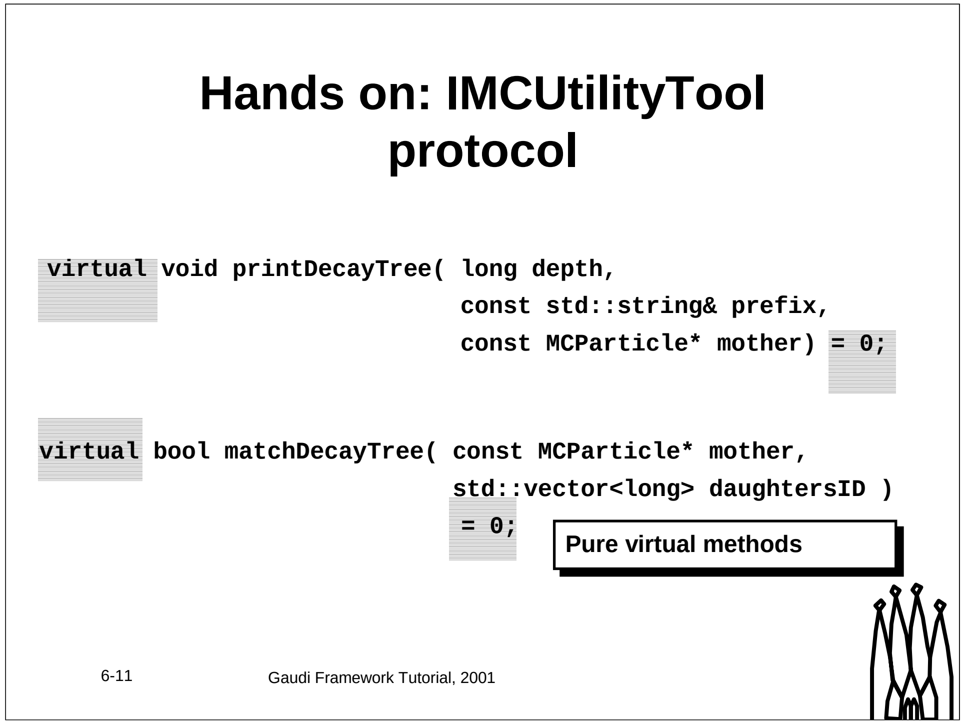
Printing a MonteCarlo tree given the parent particle is in fact something that many algorithms could want to do: we will transfer that functionality in a tool. At the same time many algorithms could want to check if a specified decay is in the event. To keep things simple we will restrict ourselves to a one step decay of n particles.

Eventually we will use the tool in an Algortihm.

We will need to

- Define the interface methods and write the interface
  IMCUtility.h
- Write the tool itself
  MCUtility.h, MCUtility.cpp
    we will make use of Particle Properties
- Write the algorithm that retrieves and uses the tool
  AnalysisAlgorithm.h, AnalysisAlgorithm.cpp

Skeleton files where only the relevant parts to the tool tutorial are missing have been prepared for you.



Hands on: IMCUtilityTool protocol

```
virtual void printDecayTree( long depth,
                             const std::string& prefix,
                             const MCParticle* mother) = 0;

virtual bool matchDecayTree( const MCParticle* mother,
                             std::vector<long> daughtersID )
                             = 0;
```

Pure virtual methods

IMCUtilityTool

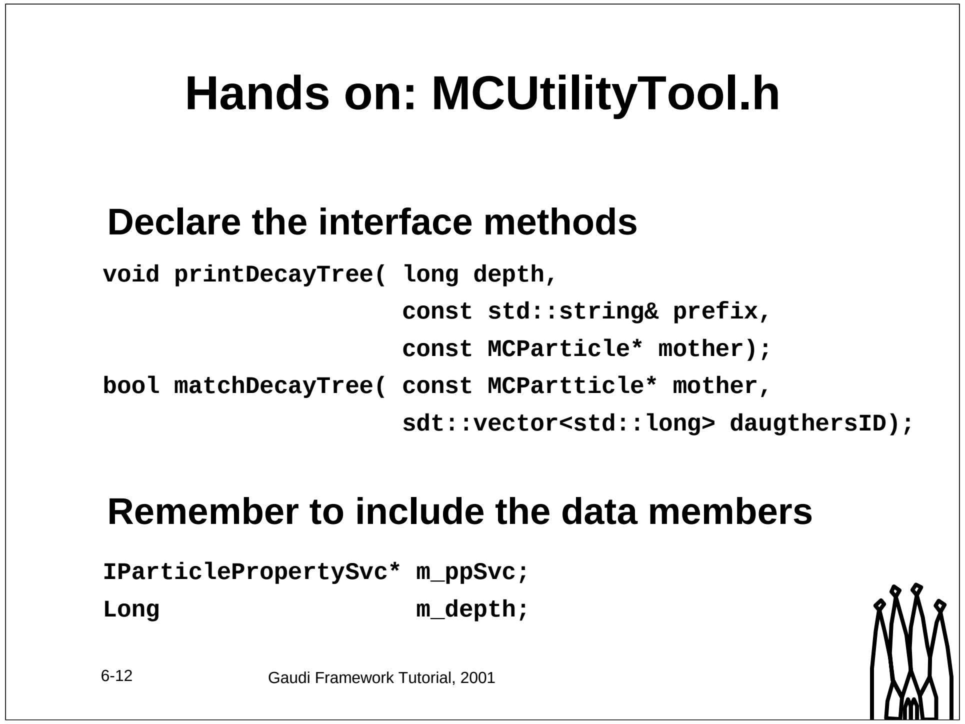
Start by having a IMCUtilityTool interface with two methods

printDecayTree

matchDecayTree

A skeleton file is provided in the directory Analysis.

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



Hands on: MCUtilityTool.h

Declare the interface methods

```
void printDecayTree( long depth,
                     const std::string& prefix,
                     const MCParticle* mother);
bool matchDecayTree( const MCPartticle* mother,
                     sdt::vector<std::long> daugthersID);
```

Remember to include the data members

```
IParticlePropertySvc* m_ppSvc;
Long                  m_depth;
```

Hands on: MCUtilityTool.cpp

Remember to include the necessary headers

To handle errors in constructor

```
#include "GaudiKernel/GaudiException.h"
#include "GaudiKernel/IParticlePropertySvc.h"
#include "GaudiKernel/ParticleProperty.h"
#include "CLHEP/Units/PhysicalConstants.h"
#include "LHCbEvent/MCParticle.h"
```

6-13 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

- **Retrieve the Particle Property Service**

```
m_ppSvc = 0;
StatusCode sc=serviceLocator()->service("ParticlePropertySvc",
                                        m_ppSvc)
```

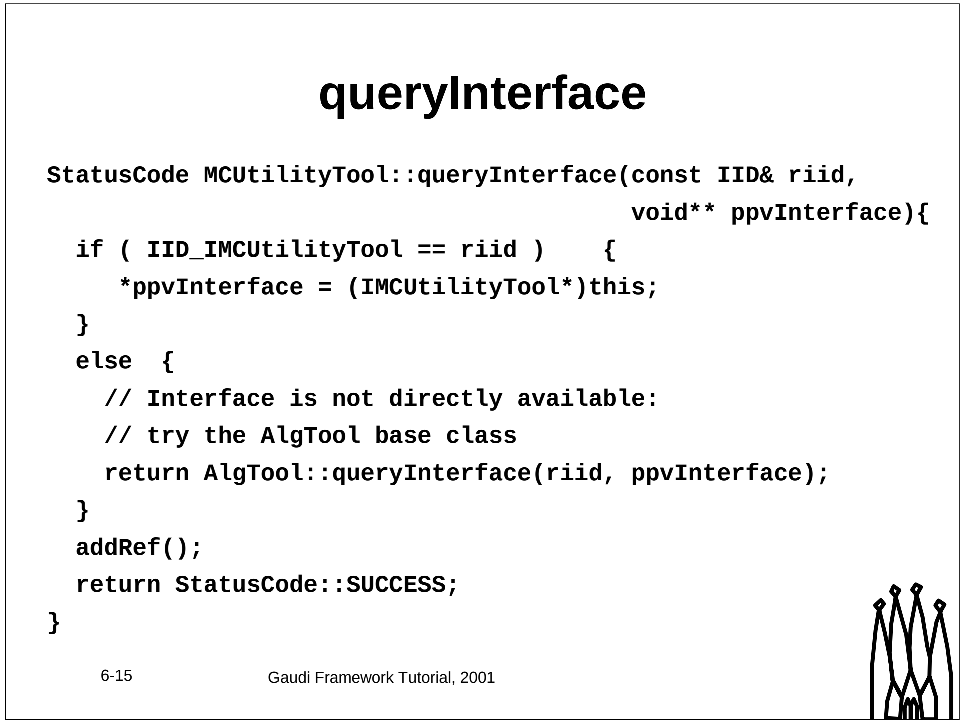
- **In case of problems throw a GaudiException**

```
throw GaudiException( "ParticlePropertySvc not found",
                      "MCUtilityToolException",
                      StatusCode::FAILURE );
```

- **Declare Properties**

```
declareProperty( "PrintDepth", m_depth = 999 );
```

6-14 Gaudi Framework Tutorial, 2001

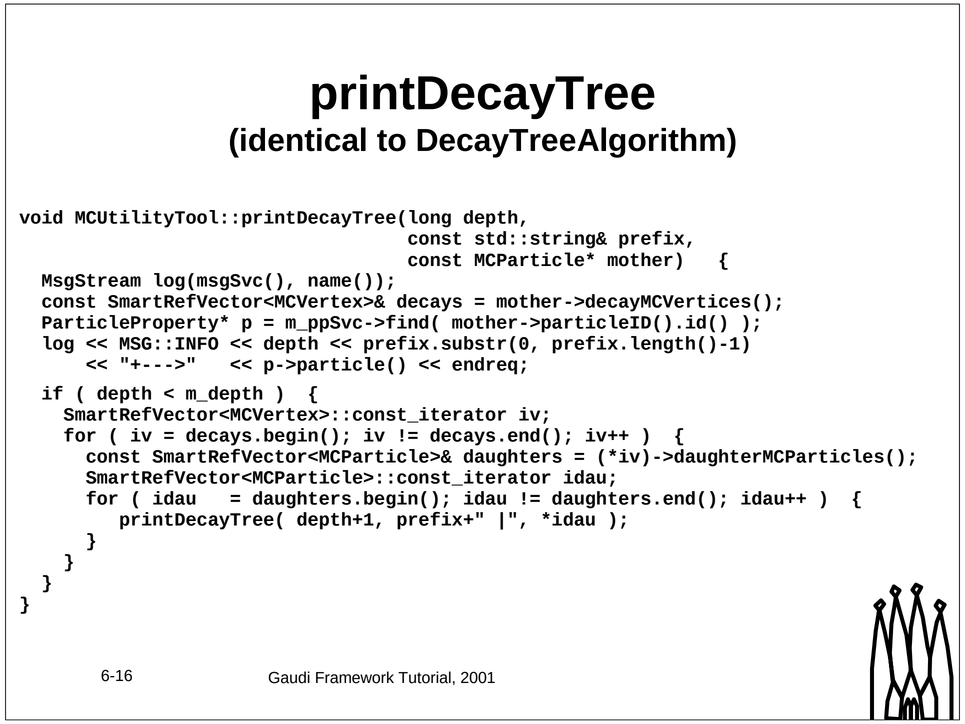


queryInterface

```
StatusCode MCUtilityTool::queryInterface(const IID& riid,
                                         void** ppvInterface){
  if ( IID_IMCUtilityTool == riid )    {
     *ppvInterface = (IMCUtilityTool*)this;
  }
  else  {
    // Interface is not directly available:
    // try the AlgTool base class
    return AlgTool::queryInterface(riid, ppvInterface);
  }
  addRef();
  return StatusCode::SUCCESS;
}
```

Note:

A tool with an additional interface has to implement the *queryInterface* method that returns the pointer to the requested interface out of those implemented. This can be done in a base class.



printDecayTree
(identical to DecayTreeAlgorithm)

```
void MCUtilityTool::printDecayTree(long depth,
                                   const std::string& prefix,
                                   const MCParticle* mother)   {
  MsgStream log(msgSvc(), name());
  const SmartRefVector<MCVertex>& decays = mother->decayMCVertices();
  ParticleProperty* p = m_ppSvc->find( mother->particleID().id() );
  log << MSG::INFO << depth << prefix.substr(0, prefix.length()-1)
      << "+--->"   << p->particle() << endreq;
  if ( depth < m_depth )  {
    SmartRefVector<MCVertex>::const_iterator iv;
    for ( iv = decays.begin(); iv != decays.end(); iv++ )  {
      const SmartRefVector<MCParticle>& daughters = (*iv)->daughterMCParticles();
      SmartRefVector<MCParticle>::const_iterator idau;
      for ( idau   = daughters.begin(); idau != daughters.end(); idau++ )  {
         printDecayTree( depth+1, prefix+" |", *idau );
      }
    }
  }
}
```

matchDecayTree

If you don’t have time just return a false

Using IToolSvc to retrieve a tool

```
...AnalysisAlgorithm.h
  std::string        m_toolName;     ///< Tool name
  IMCUtilityTool*    m_pUtilTool;    ///< Reference to tool

...AnalysisAlgorithm::initialize()...
  IToolSvc* toolsvc = 0;
  StatusCode sc = service("ToolSvc", toolsvc );
  if ( sc.isFailure() ) {// You have to handle the error!
  }
  sc = toolsvc->retrieveTool( m_toolName, m_pUtilTool );
  if ( sc.isFailure() ) {// You have to handle the error!
  }
```

Using a tool

```
...AnalysisAlgorithm::execute()
// inside loop over MCParticle
if ( (*ipart)->particleID().id() == m_partID )   {
  log << MSG::INFO << "Found Particle of type " << m_partName
      << endreq;
  // Now use tool to print tree and check if requested tree
  m_pUtilTool->printDecayTree( 0, " |", *ipart );
  bool result = m_pUtilTool->matchDecayTree( *ipart,
                                           m_daugID );
  if ( result ) { // Print a message
  } else { // Print a different message
}
```

Configuring a tool

A concrete tool can be configured using the jobOptions

Follow usual convention:

IdentifyingNameOfTool.NameOfProperty

```
ToolSvc.MCUtilityTool.PrinDepth = 0;
```

ParentName.ToolName

Through the base class all tools have the OutputLevel property

- The default value is that of the parent

Public and Private tools

- A tool instance can be shared by many algorithms: it is public and belong to the ToolSvc
- Algorithm can have private instances of tools, that can be configured “ad-hoc”
- To retrieve a private instance the parent has to pass itself to the retrieve method

```
toolSvc->retrieveTool(“MyTool”,pMyTool, this)
```

Public and Private Tools

A tool instance can be shared by different algorithms and services. Its parent is the ToolSvc.

It is possible to re-configure such instances but care has to be taken to ensure no undesired side effects in the algorithms that use it.

In addition an algorithm could need to use two differently configured instances of the same tool

Private instances of tools address these points. Although they are managed by the ToolSvc, they are seen as belonging to the algorithm requesting them. The algorithm has to pass itself in order to notify the ToolSvc that the tool instance has to be private.

Only algorithms and services can be tools’ parents.

Hands on: Private Tools

Use two instances of the same tool in the AnalysisAlgorithm

- A public instance with default configuration
- A private instance with different printing depth

Note:

The tool is not modified, only the algorithm using it.

Using a private and a public tool

```
...AnalysisAlgorithm.h
  std::string       m_myToolName;     ///< Tool name
  IMCUtilityTool*   m_pMyUtilTool;    ///< Reference to tool

...AnalysisAlgorithm::initialize()...
  sc = toolsvc->retrieveTool( m_myToolName, m_pMyUtilTool,
                              this );
...AnalysisAlgorithm::execute()...
  m_pMyUtilTool->printDecayTree( 0, " |", *ipart );
  bool result = m_pMyUtilTool->matchDecayTree( *ipart,
                                               m_daugID );
  if ( !result ) {
    m_pUtilTool->printDecayTree( 0, " |", *ipart );
  }
```

6-23 Gaudi Framework Tutorial, 2001

Extending existing tools

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

6-24 Gaudi Framework Tutorial, 2001

Extending existing tools

The same operation can be performed in a different way than that of a tool you have available.

For example you would like to print different information when you print a decay tree. You can extend the existing tool or implement the tool interface in a new tool. As long as the 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 (like the queryInterface for example) and implement them in a base class.

Hand on: A different tool implementation

Write a NewMCUtilityTool that prints the tree with different type of information for the MCParticle

- **Inherit from MCUtilityTool**
- **Override printDecayTree method**
- **Change name of one of the tools used by AnalysisAlgorithm of previous exercise: only in job options**

Hands on: NewMCUtilityTool

```
...NewMCUtility.h
  #include "MCUtilityTool.h"
  class NewMCUtilityTool : public MCUtilityTool {
  ...
  void printDecayTree( long depth, const std::string& prefix,
                       const MCParticle* mother);
...NewMCUtility.cpp
  void NewMCUtilityTool::printDecayTree(long depth,
                               const std::string& prefix,
                               const MCParticle* mother) {
  // Change the method as you like
```

Hands on: Use different tools

Once the tool exist you only need to load the modified job options.

An example NewMCUtility.opts is provided