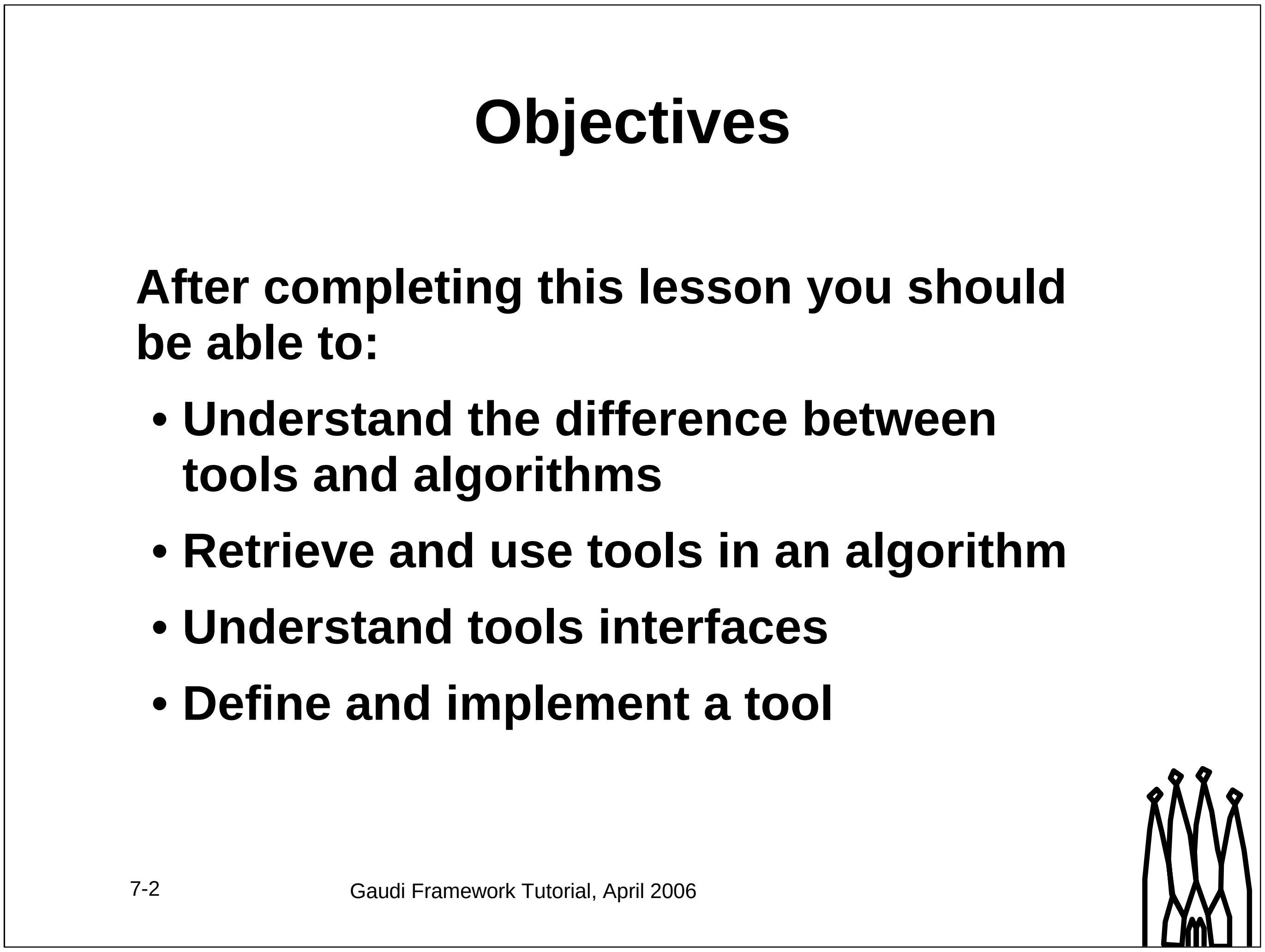
7

Algorithm Tools: what they are, how to write them, how to use them

Gaudi Framework Tutorial, April 2006

Schedule:	Timing	Topic
	20 minutes	Lecture
	30 minutes	Practice
	50 minutes	Total



Objectives

After completing this lesson you should be able to:

- **Understand the difference between tools and algorithms**
- **Retrieve and use tools in an algorithm**
- **Understand tools interfaces**
- **Define and implement a tool**

Goals

The first goal of this 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.

The second objective of the lesson is to enable you to use tools, understanding the difference between public and private instances of tools.

The third objective is to learn to write a new tool

Why Algorithm Tools?

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. Or the same operation might need to be performed by several different algorithms.

➔ 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 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

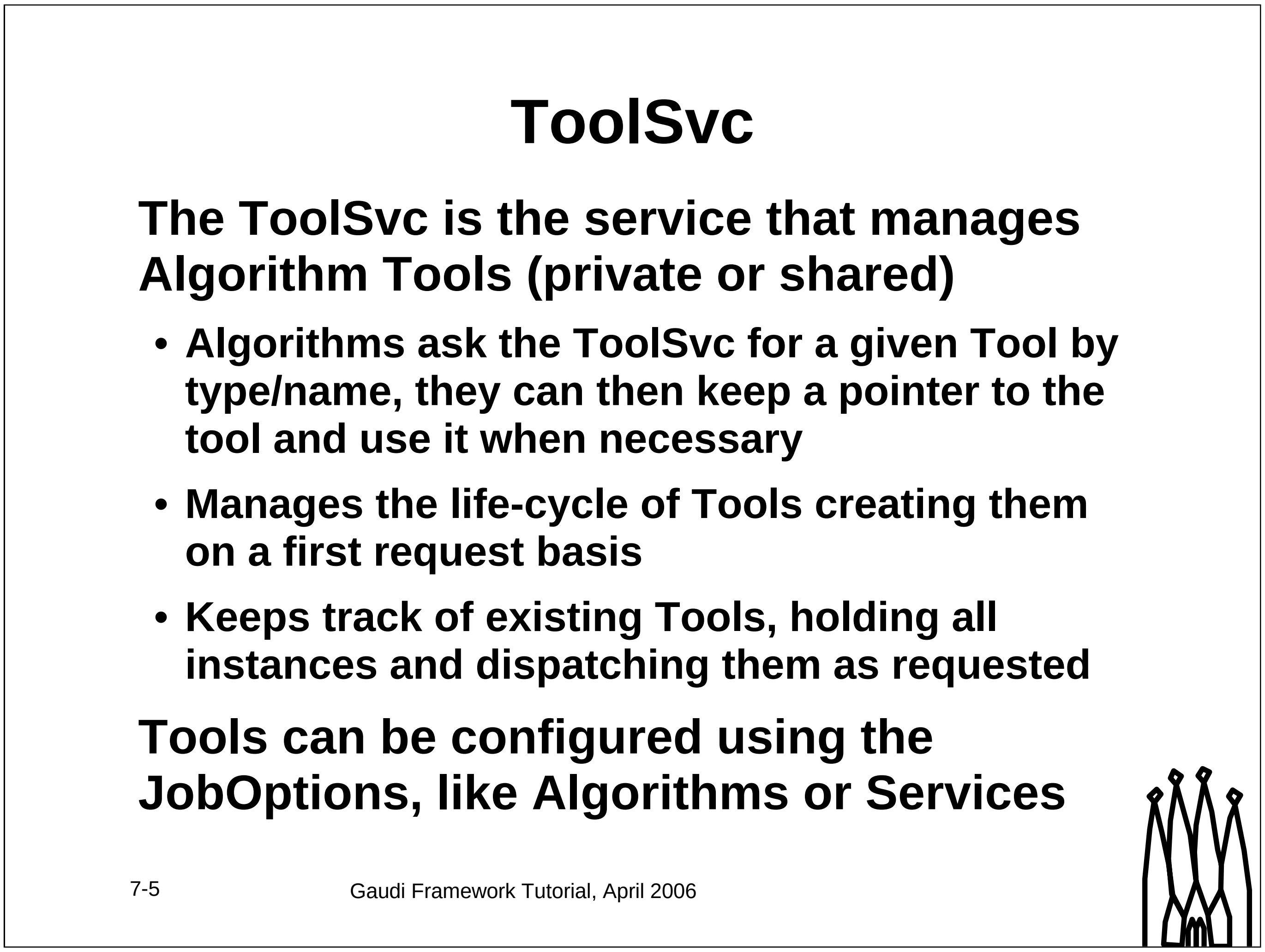
- **Examples**
  - **Vertexing**
  - **Track transport**
  - **Association to truth**
  - **Filtering of particles based on a pID CL**

Designing for Re-use

Algorithm Tools are useful small algorithms that can be packaged in a way that will make it 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)



ToolSvc

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

- **Algorithms ask the ToolSvc for a given Tool by type/name, they can then keep a pointer to the tool and use it when necessary**
- **Manages the life-cycle of Tools creating them on a first request basis**
- **Keeps track of existing Tools, holding all instances and dispatching them as requested**

Tools can be configured using the JobOptions, like Algorithms or Services

7-5 Gaudi Framework Tutorial, April 2006

The ToolSvc Service

This service is managing Algorithm Tools. It is the service in charge of tools in their life-cycle, it creates them on first request, 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 type/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) it does make 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 a state between invocations since it is not guaranteed that other Algorithms may have used it meanwhile.

Accessing Tools from GaudiAlgorithm

In MyAlgorithm.h:

```
private:
  IMCAcceptanceTool* m_accTool; ///< Tool interface
```

In MyAlgorithm::initialize()

```
 // Get the tool from the toolSvc
  m_accTool = tool<IMCAcceptanceTool>(
                  "MCAcceptanceTool/MyTool" );
```

In MyAlgorithm::execute()

```
// Do something with it
if( m_accTool->accepted( *itP )) { ++numAccepted; }
```

Accessing tools from GaudiAlgorithm

GaudiAlgorithm hides the technicalities of contacting the toolSvc to obtain a tool, and of properly releasing it at the end of the job.

Every tool implements an interface specific to that tool. The algorithm requests the tool using its tool() method. This method is templated by the Tool interface; the argument is a string containing the class name of the tool, optionally followed by an instance name. If a *public* tool with this name already exists, the ToolSvc returns a pointer to it. If it doesn’t, the ToolSvc creates the tool instance, initializes it, and returns the pointer. This pointer should then be stord by the algorithm in a member variable, for later use.

Sometimes it may be useful to have a *private* instance of a tool, configured specially for the calling algorithm, rather than a shared *public* one. In this case one should call the tool() function with an additional argument:

m_accTool = tool<IMCAcceptanceTool>("MCAcceptanceTool/MyTool“, this);

Which tells the toolSvc to create and return a private instance of the tool, whose owner is the object pointed to by “this” (i.e. the calling algorithm)

Configuring a tool

A concrete tool can be configured using the jobOptions

Follow usual convention:

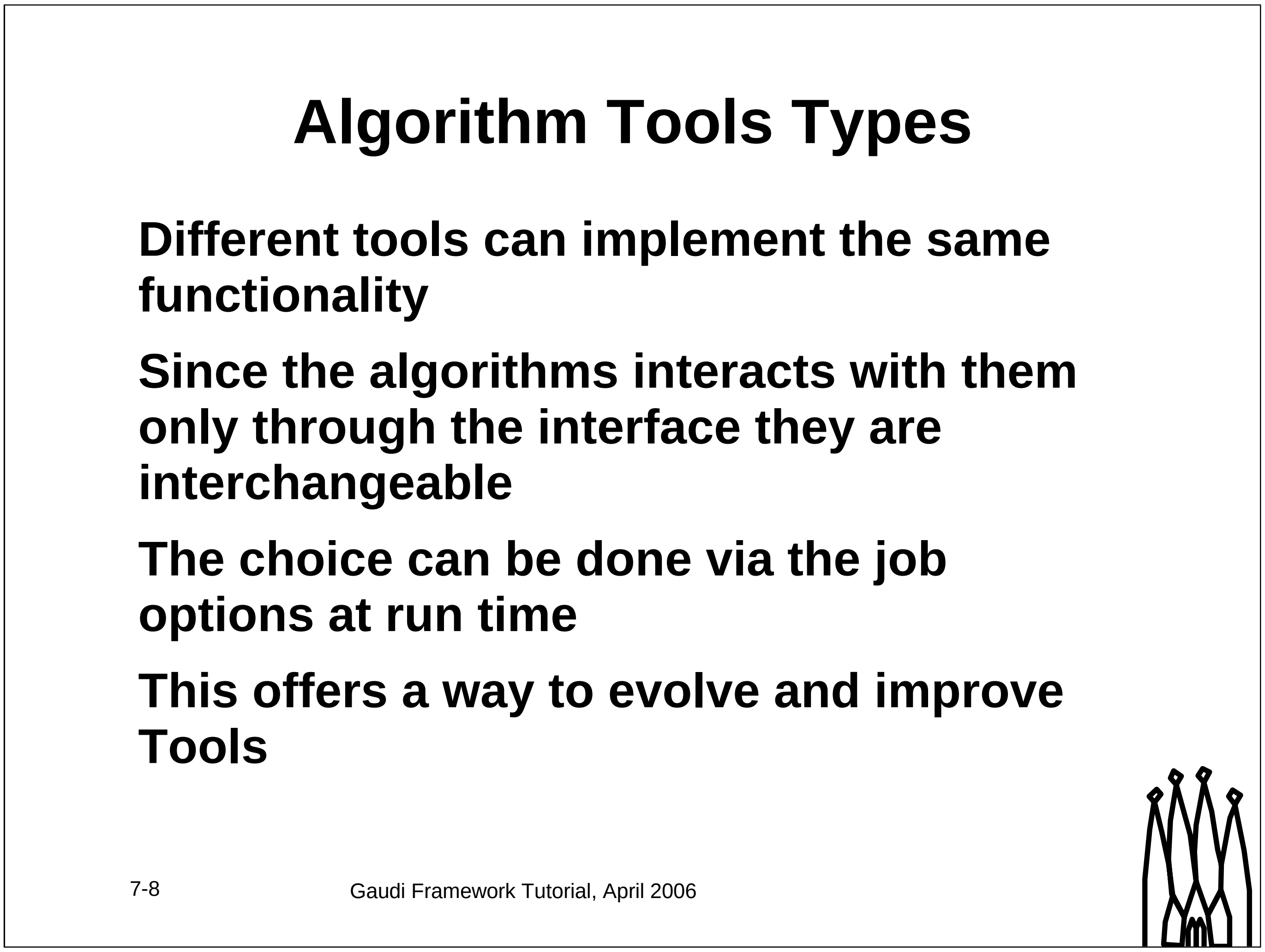
IdentifyingNameOfTool.NameOfProperty

```
ToolSvc.MyTool.MinPz = 0.1*GeV;
```

ParentName.ToolName

Through the base class all tools have the OutputLevel property

- The default value is that of the parent



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

7-8 Gaudi Framework Tutorial, April 2006

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 tool() method. If this string is a property of the algorithms, the concrete tool used can be chosen at run-time via the job options.

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 an additional interface to define its functionality

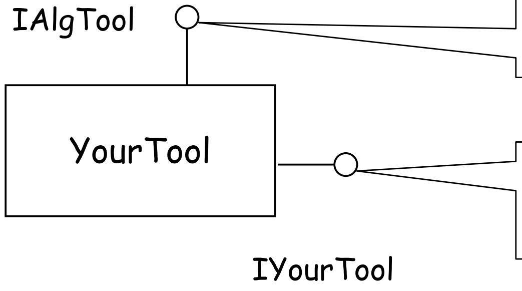
- which must inherit from the IAlgTool interface, so that the tool can be managed by ToolSvc
  - Remember: The implementation of the IAlgTool interface is done by the AlgTool base class, you don't need to worry about it
- Tools performing a similar operation in different ways will share the same interface (ex. Vertexer)

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



IAlgTool

- **name(), type(), parent()**

IYourTool

- **Interface for your tool**

Interfaces

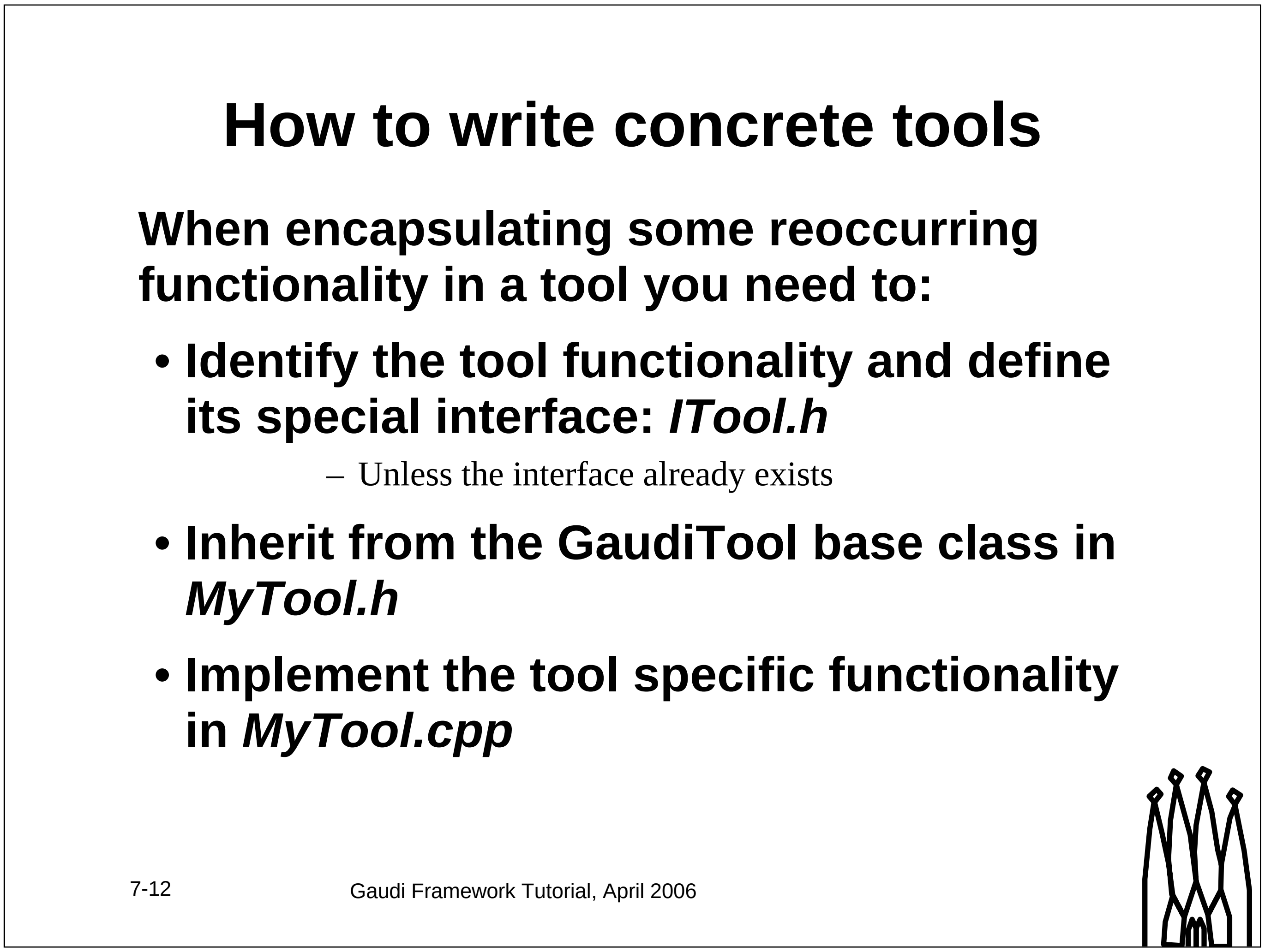
•**IAlgTool**. Basic interface that any AlgTool implements and is used for bookkeeping purposes of the ToolSvc.

•**IYourTool**. This represents the interface (abstract) for this particular tool or class of tools.

GaudiTool base class

All tools should inherit from GaudiTool

- **Similar functionality to GaudiAlgorithm**
- **Hides details of**
  - **MsgStream**
  - **Event Data Service access**
  - **etc.**



How to write concrete tools

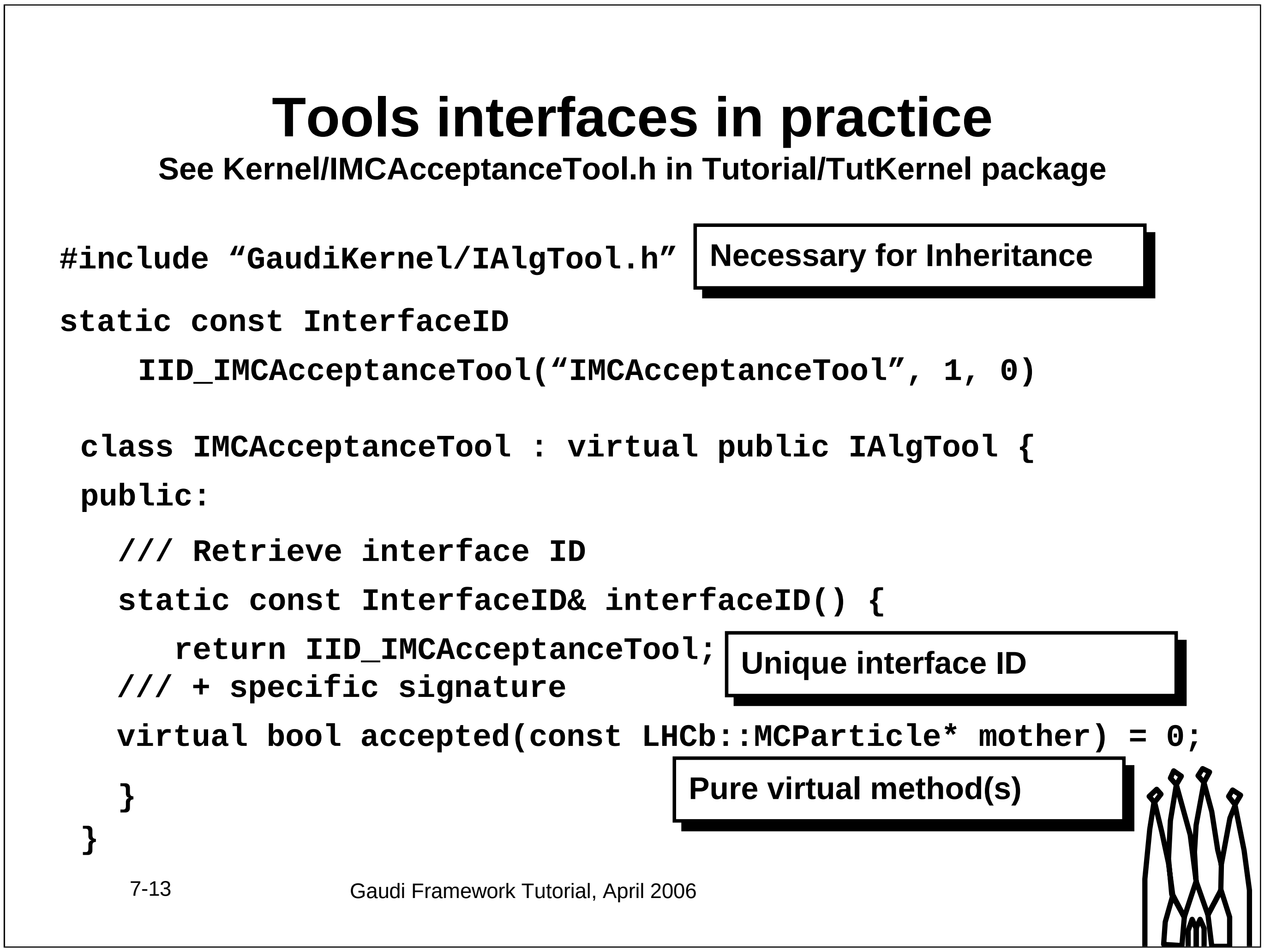
When encapsulating some reoccurring functionality in a tool you need to:

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

How to write concrete tools: header file

By inheriting from the *GaudiTool* Base class, concrete tools are managed by the *ToolSvc*. The base class in fact inherits from the *AlgTool* base class, which in turn 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 interfaces in practice

See Kernel/IMCAcceptanceTool.h in Tutorial/TutKernel package

```
#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 LHCb::MCParticle* mother) = 0;
   }
 }
```

Necessary for Inheritance

Unique interface ID

Pure virtual method(s)

7-13 Gaudi Framework Tutorial, April 2006

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.

Note that the only specific code in this file is the interface name, and the specific signature. Everything else is common to all tool interfaces. These files are easily generated by Emacs

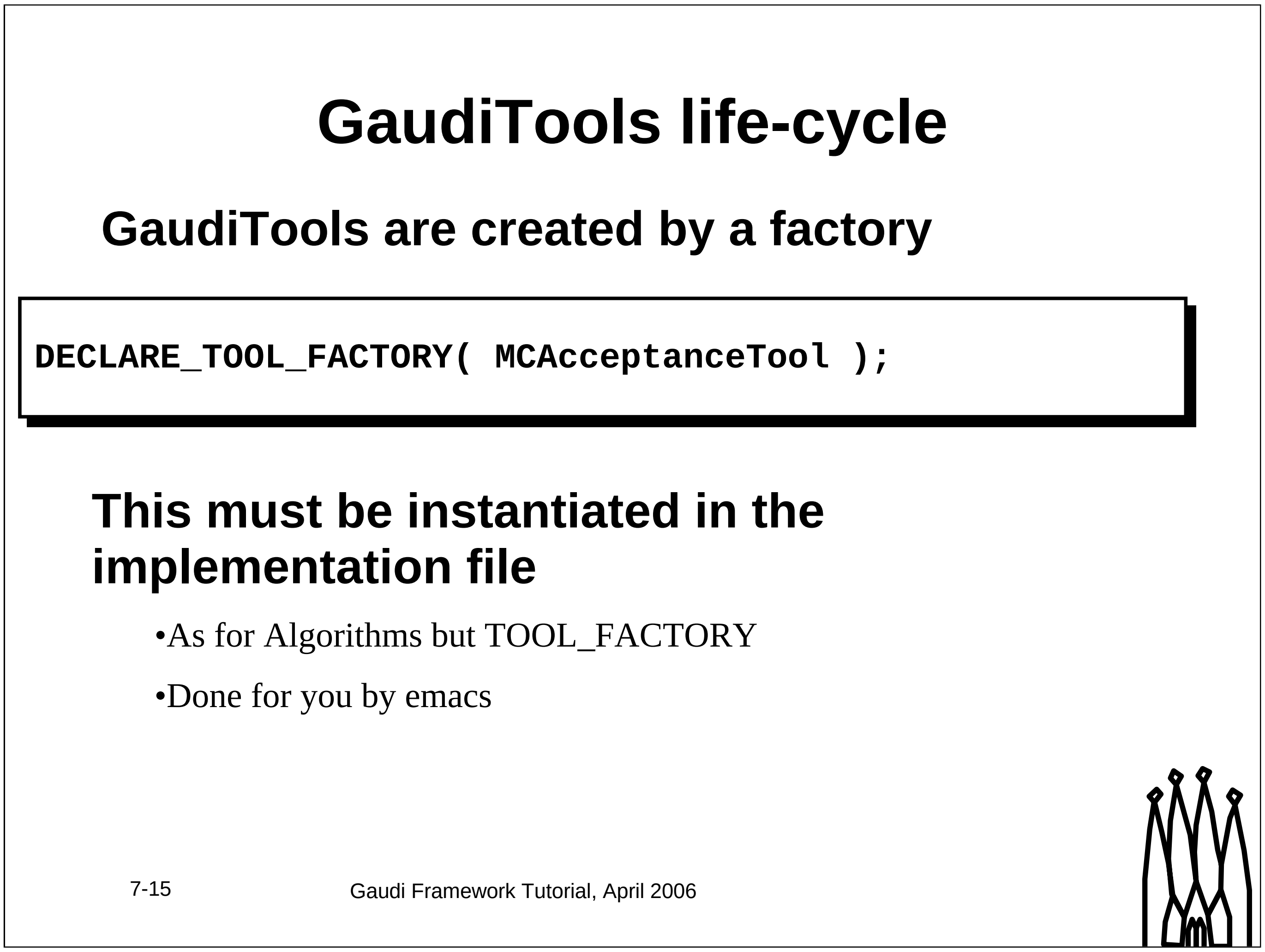
GaudiTool inheritance

A concrete tool will inherit from the GaudiTool base class:

- has properties — Configurable via job options
- debug(), info(), warning() etc.
- get() — access to event data
- tool(), svc() — access to other tools, services
- possible initialize(), finalize() — Called after creation by ToolSvc — Configured at creation

How to write concrete tools: header file

By inheriting from the *GaudiTool* Base class, all the necessary communication with the ToolSvc is implemented, and you have access to all the nice shortcuts also available in GaudiAlgorithm



GaudiTools life-cycle

GaudiTools are created by a factory

```
DECLARE_TOOL_FACTORY( MCAcceptanceTool );
```

This must be instantiated in the implementation file

- •As for Algorithms but TOOL_FACTORY
- •Done for you by emacs

7-15 Gaudi Framework Tutorial, April 2006

GaudiTool creation and Configuration

GaudiTools 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 GaudiTool 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 macro DECLARE_TOOL_FACTORY 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

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.

A note about packaging

Tools, like Algorithms, are *components*

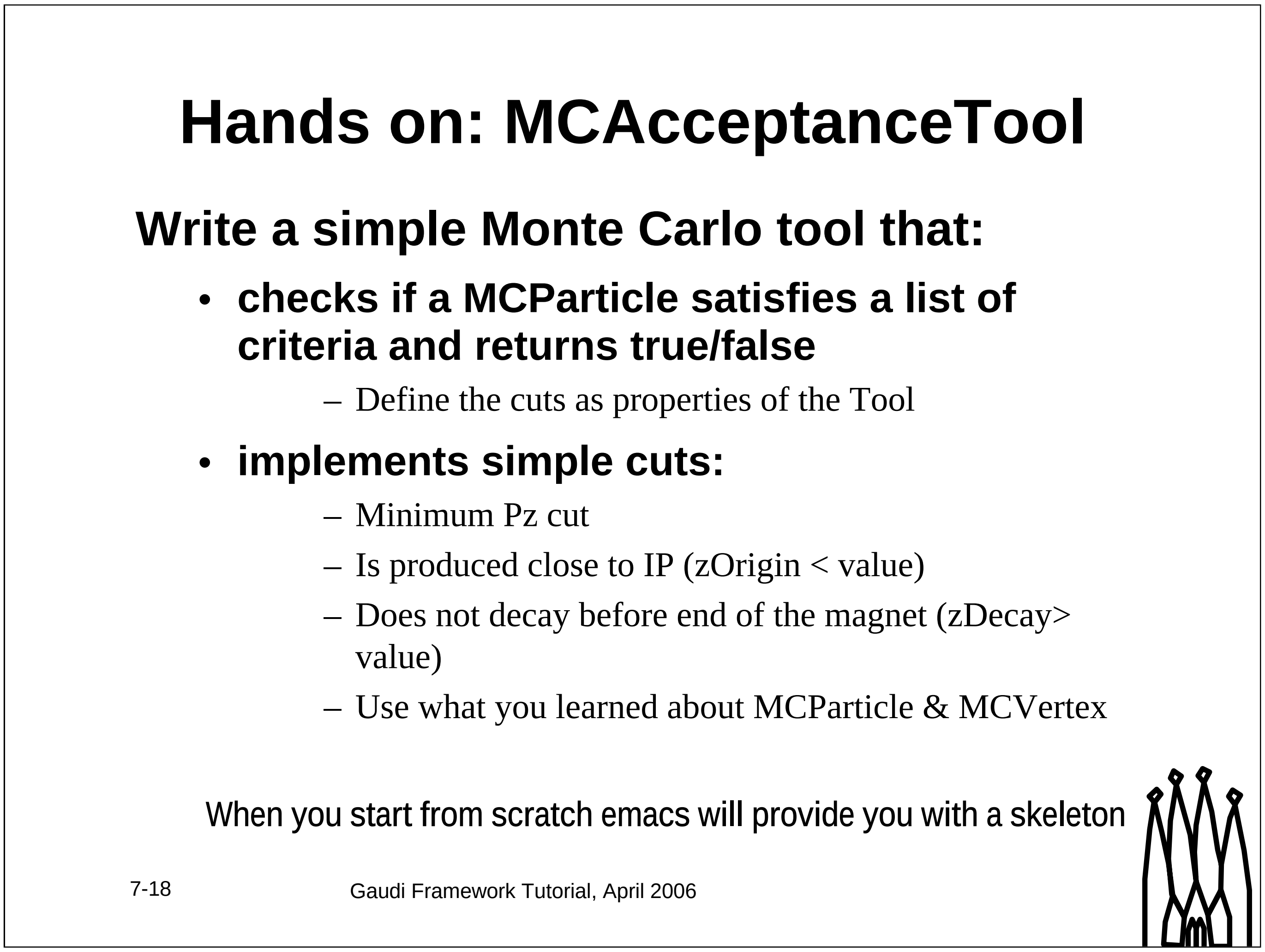
- They should reside in a component library

Header files of Tool interfaces are public

- They should reside in a public include directory

Component packages should not export an include directory

- include_path none
- Put the interface in a separate *Kernel* package
  - e.g. TutKernel/v*x*r*y*/Kernel
- Component library will *depend* on Kernel package
  - If interface changes, Components must be recompiled
  - THINK TWICE before changing an interface!



Hands on: MCAcceptanceTool

Write a simple Monte Carlo tool that:

- **checks if a MCParticle satisfies 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 about MCParticle & MCVertex

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

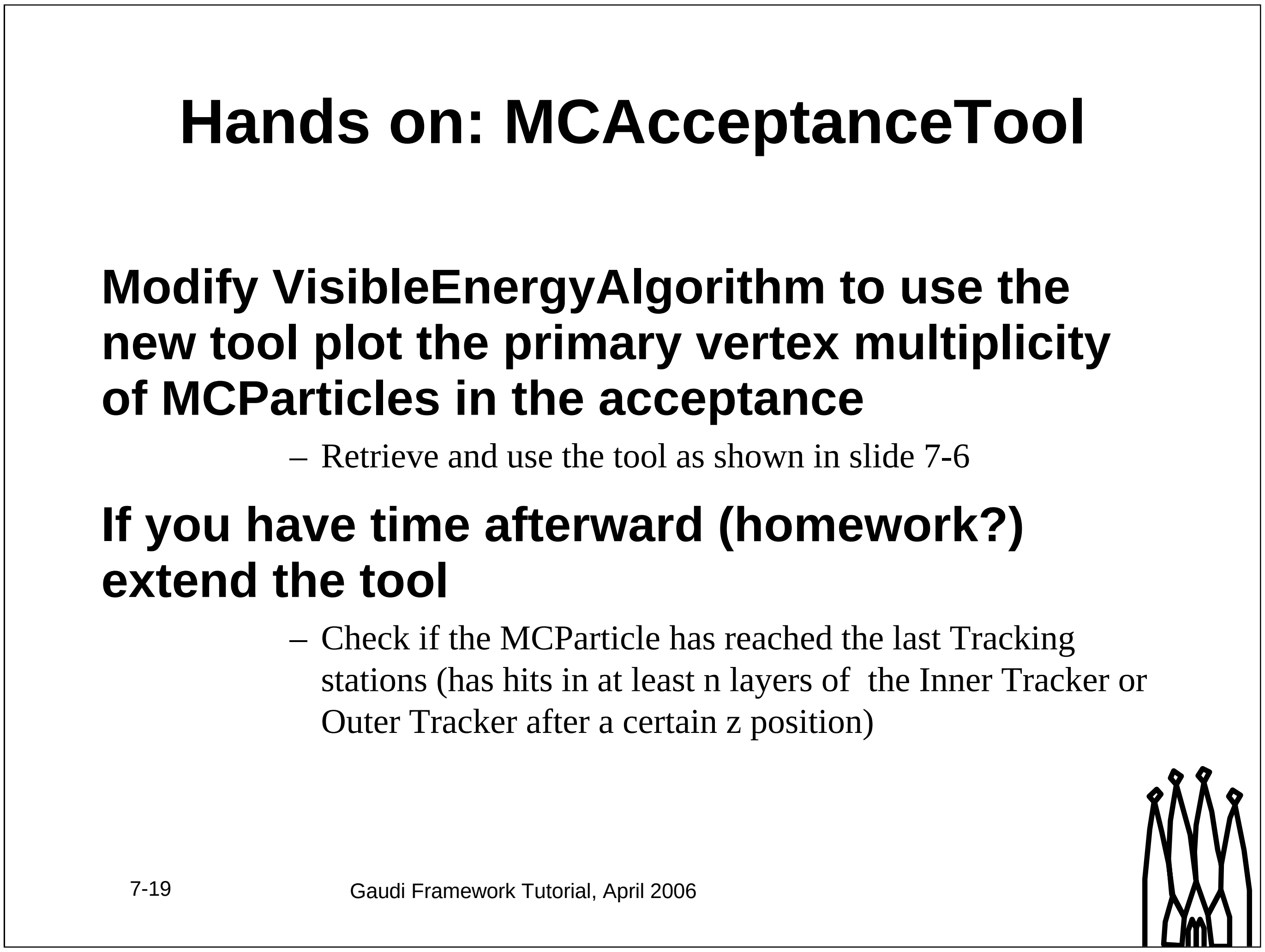
Hands on

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
  VisibleEnergyAlgorithm.h, VisibleEnergyAlgorithm.cpp



Hands on: MCAcceptanceTool

- **Modify VisibleEnergyAlgorithm to use the new tool plot the primary vertex multiplicity of MCParticles in the acceptance**
  - Retrieve and use the tool as shown in slide 7-6
- **If you have time afterward (homework?) 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: MCAcceptanceTool.h

Inherit from IMCAcceptanceTool

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

Declare the interface method(s)

```
virtual bool accepted( const LHCb::MCParticle* mother );
```

Hands on: MCAcceptanceTool.cpp constructor

Declare specific Interface(s)

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

Implement the interface method

```
bool MCAcceptanceTool::accepted( const LHCb::MCParticle* mcpart )
{
  /// Momentum cut (Pz)
  if ( mcpart->momentum().z() < m_minPz ) {
    return false;
  } else {
    return true;
  }
}
```

Solution

In src.data directory of Tutorial/Components package

To try this solution:

```
cd ~/cmtuser/Tutorial/Component/v7r0/src
Move your modified files if you want to keep them
cp ../src.usetool/*.* .
cd ../cmt
gmake
$MAINROOT/$CMTCONFIG/Main.exe $MAINROOT/options/jobOptions.opts
```