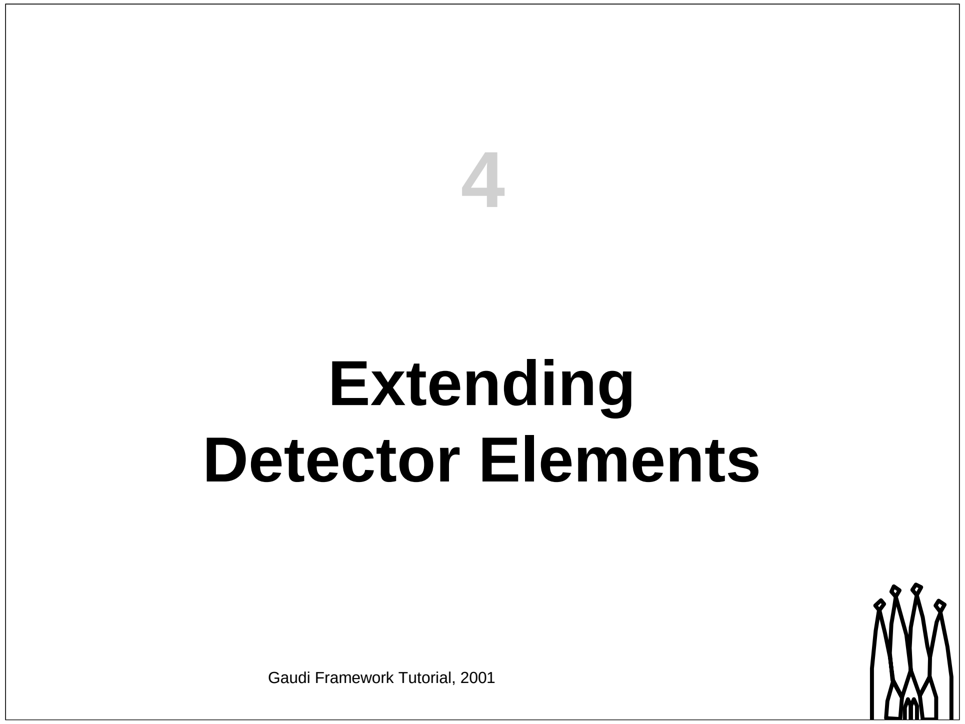


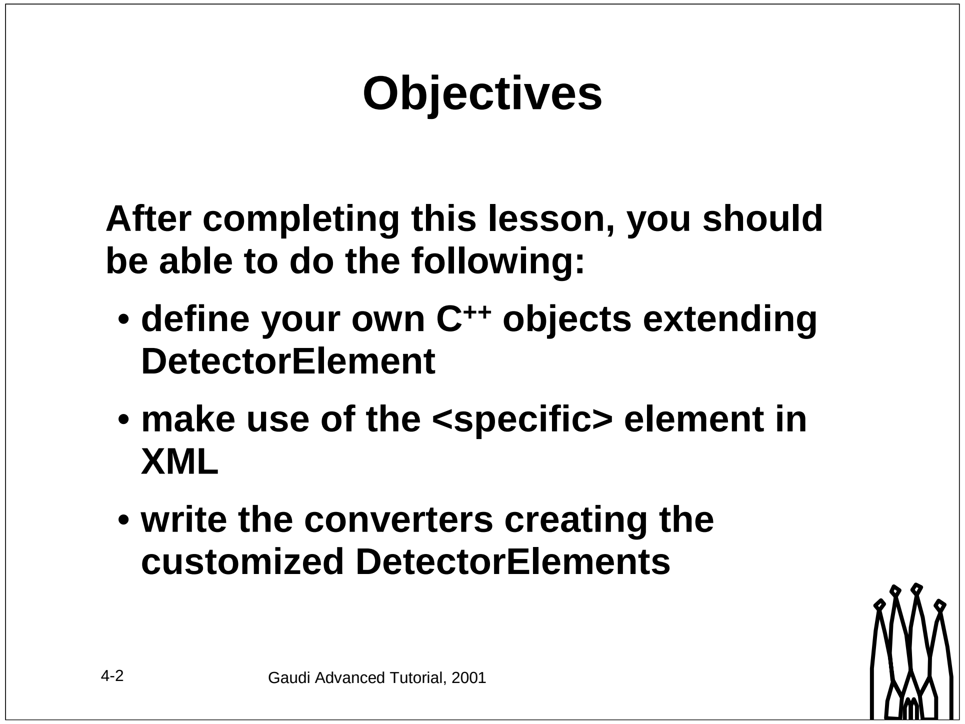
4

Extending Detector Elements

Gaudi Framework Tutorial, 2001

This part of the tutorial deals with 2 ways of extending the default schema presented in previous lessons for user specific purposes. It will be followed by a small exercise, aiming at using these extensions to add specific data to the XML code written in previous exercises.

Schedule:	Timing	Topic
	20 minutes	Lecture
	40 minutes	Practice
	60 minutes	Total



Objectives

After completing this lesson, you should be able to do the following:

- **define your own C^{++} objects extending DetectorElement**
- **make use of the <specific> element in XML**
- **write the converters creating the customized DetectorElements**

4-2 Gaudi Advanced Tutorial, 2001

Lesson Aims

- define specific extensions of the DetectorElement object in C^{++}. This will allow the user to define new members and methods.
- make use of the <specific> element in XML. This is the place where the user will add new XML elements for his specific data.
- write converters dealing with the new elements found inside the <specific> tag and creating extended DetectorElements.

When to Extend ?

- Customize C^{++} DetectorElement object with specific behavior (answering specific reconstruction/simulation questions)
- Add specific information to detector elements which is more complex than userParameters

$\Longrightarrow$ full freedom

When to extend

There are mainly two cases in extending the default schema :

- one needs specific behavior of the DetectorElement object and thus needs to extend it. An example of such a case is the implementation of specific reconstruction or simulations customizations.
- one needs to add some specific information in the XML code that is more complex than the userParameters and that should be expressed using XML. This means that the user wants to add new XML elements and to extend the DTD.

Extending DetectorElement (1)

- new C^{++} class, with a new ClassID
- must inherit from DetectorElement
- You can add any member
- You can add any method
- It must have a constructor with no parameters
- It has an initialize method
  - called after the end of the conversion
  - is the only place where the object can be initialized according to specific parameters (userParameters)

Extending DetectorElement

The extension of the DetectorElement class consist in creating a new C^{++} class, inheriting (directly or not) from DetectorElement. This class must have certain specificities :

- it must have a new classID, unique in LHCb and identifying this new type of detector element
- it may have any new member and method
- it must have a constructor with no parameters. This is required by default converters that will create the object.

On top of that, the object will inherit the initialize method that is called at the end of the conversion mechanism. This method is the place where any initialization of the object that uses specific parameters, like userParameters, should be done. For non specific initialization, the constructor can be used as usual.

Extending DetectorElement (2)

```
class MyDetElem :
  public DetectorElement {

public:
 MyDetElem() {};
 virtual ~MyDetElem() {};

 const CLID& clID() const {
   return classID();
 }
 static const CLID& classID();

 virtual StatusCode initialize();
 int channelNb ();

private :
 int m_channelNb;
};
```

```
const CLID&
MyDetElem::classID() {
  return CLID_MyDetElem;
}

int
MyDetElem::channelNb(){
  return m_channelNb;
}

StatusCode
MyDetElem::initialize() {
  m_channelNb =
    userParameterAsInt
      ("ChannelNb"));
  return SUCCESS;
}
```

Extending DetectorElement

Here a an example of the definition of a new DetectorElement, called MyDetElem. It has a new member, storing the number of channels and a new method retrieving it. The value of the new member is initialized in the initialize method from the value of one of userParameters.

On top of this specific code, note the definition of the default constructors and destructors (empty here) and the two methods concerning the classID. These are mandatory when defining a new kind of detector element. The definition of the CLID_MyDetElem constant is actually not inside the class but in a separate .h file with some #ifdef and #define statements in order to avoid multiple definition of it (especially because the .h is included both in the converter and in the object itself).

Writing a Dummy Converter

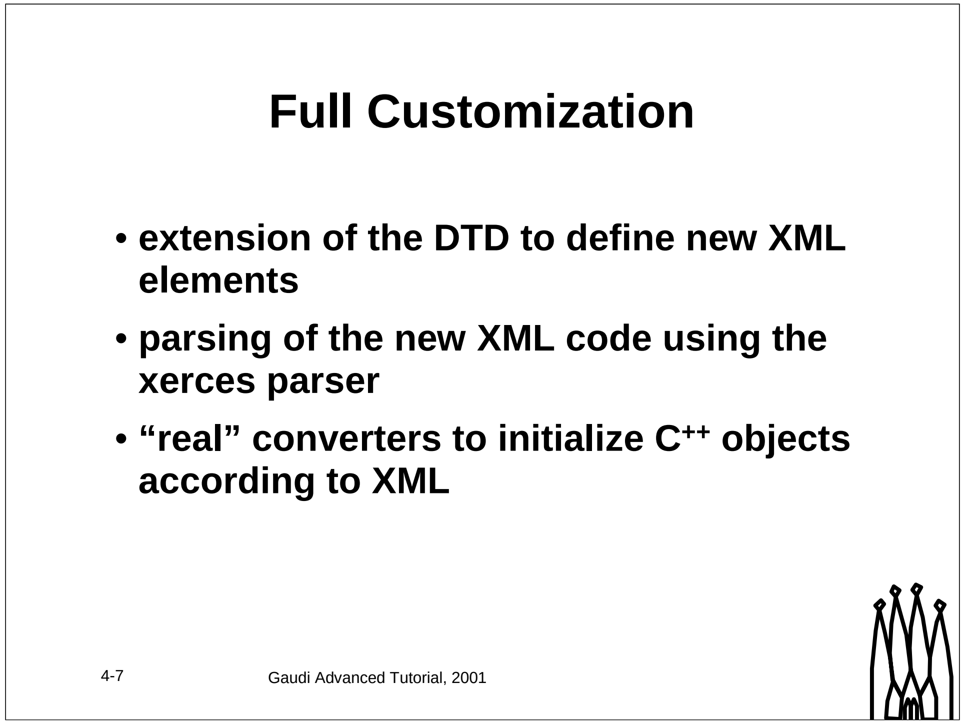
- **One must declare a new converter that will create MyDetElem instead of DetectorElements**
- **The templated class called XmlUserDetElemCnv may be used**

```
#include "DetDesc/XmlUserDetElemCnv.h"
#include "MyDetElem.h"

static CnvFactory
  <XmlUserDetElemCnv<MyDetElem>> s_factory;
const ICnvFactory& XmlMyDetElemCnvFactory = s_factory;
```

Writing a dummy converter

We have just defined a new kind of DetectorElement. But at this point, no converter is able to create it, so we will still get regular DetectorElements out of the XML to C^{++} conversion. In order to fix that, we have to write a "dummy" converter. It is more or less a copy of the regular converter for detector elements, except that it creates MyDetElem.

In order to achieve that easily, the templated class XmlUserDetElemCnv was created. The creation of the new converter is thus only 4 lines that you can copy from this slide, by replacing MyDetElem by the name of the new DetectorElement object.



Full Customization

- extension of the DTD to define new XML elements
- parsing of the new XML code using the xerces parser
- “real” converters to initialize C^{++} objects according to XML

4-7 Gaudi Advanced Tutorial, 2001

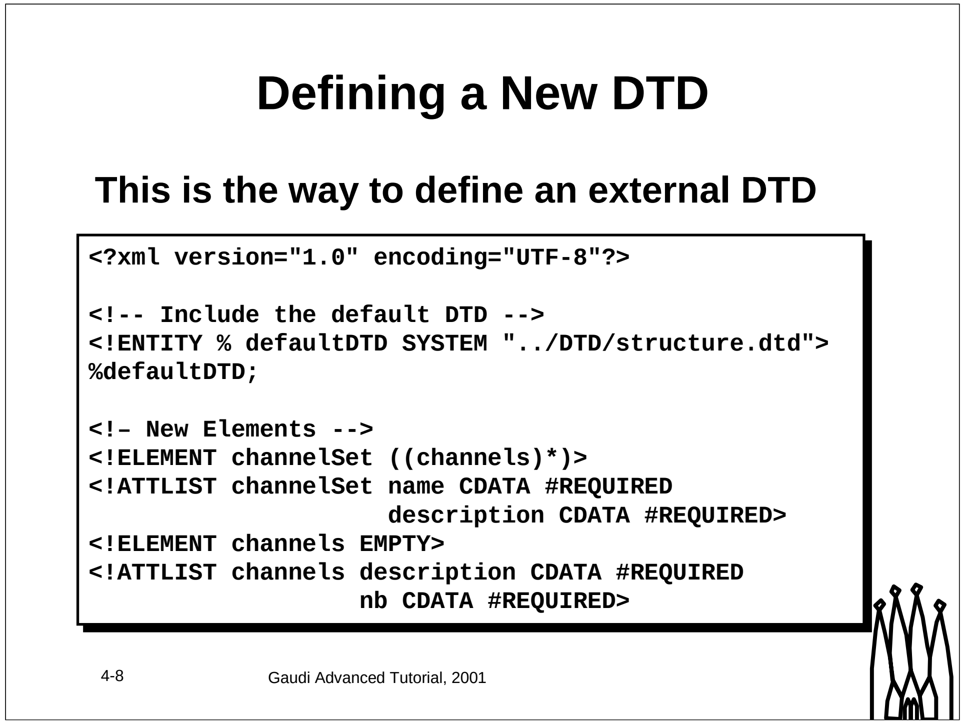
Full Customization

Up to know, we learnt how to use userParameters and how to extend the DetectorElement object. This already allows some customization but does not allow a real extension of the default schema in the sense that you have no way to add data to the XML by using new XML elements that were not defined in the LHCb DTD.

This possibility exists but it requires some work :

- one should first extend the LHCb DTD to define correctly the new elements
- then this extended DTD should be parsed to retrieved the data. This is done by using a free XML parser called xerces
- at last, specialized converters are needed to deal with the new data and store them into dedicated DetectorElements

We will now detail every step.



Defining a New DTD

This is the way to define an external DTD

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Include the default DTD -->
<!ENTITY % defaultDTD SYSTEM "../DTD/structure.dtd">
%defaultDTD;

<!– New Elements -->
<!ELEMENT channelSet ((channels)*)>
<!ATTLIST channelSet name CDATA #REQUIRED
                     description CDATA #REQUIRED>
<!ELEMENT channels EMPTY>
<!ATTLIST channels description CDATA #REQUIRED
                   nb CDATA #REQUIRED>
```

4-8 Gaudi Advanced Tutorial, 2001

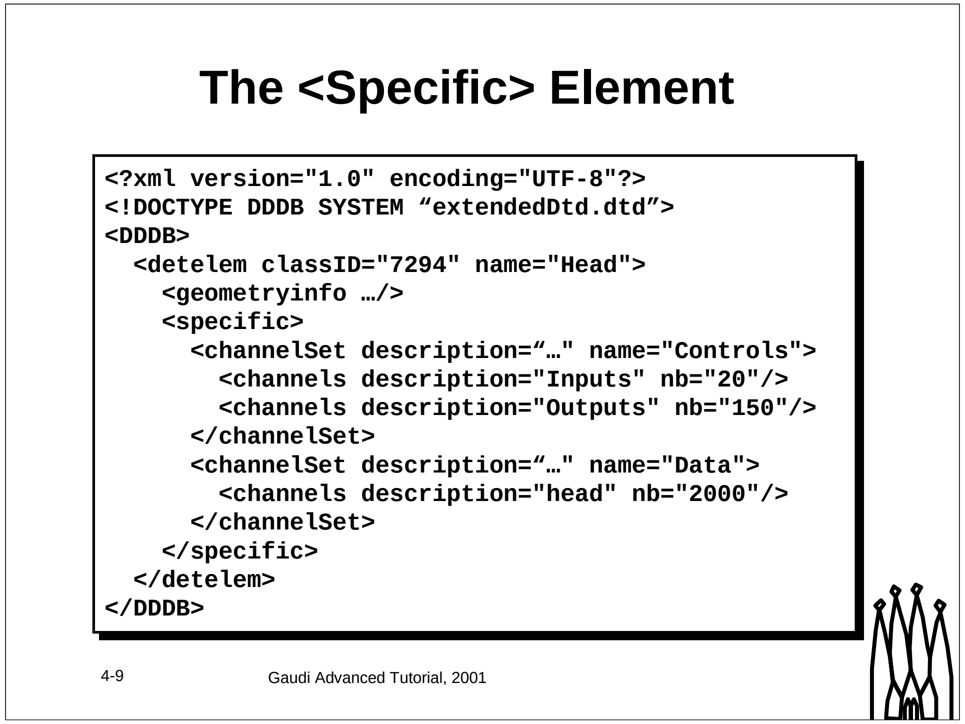
Defining a new DTD

There are actually two ways of extending a DTD, either by defining an internal DTD, or by defining an external DTD.

An internal DTD is so called because it is included inside the XML file itself, in the DOCTYPE element. This possibility should not be used here because the extension of the DTD can only be used inside the file itself. Thus, several file will have several copy of the extended DTD and the synchronization will be hard.

External DTDs are DTDs defined in a separate file. We already presented this kind of DTD in lesson 2. The advantage here is that the XML files using this DTD all point to the same instance of the DTD. Thus, in you change it, it will be changed for every file.

We won’t detail here how to write the extended DTD, you may just copy and modify the example above, which should be pretty clear. The only important thing is not to forget to include the old DTD since it is only an extension. This is done by using entities, as show on the example.



The <Specific> Element

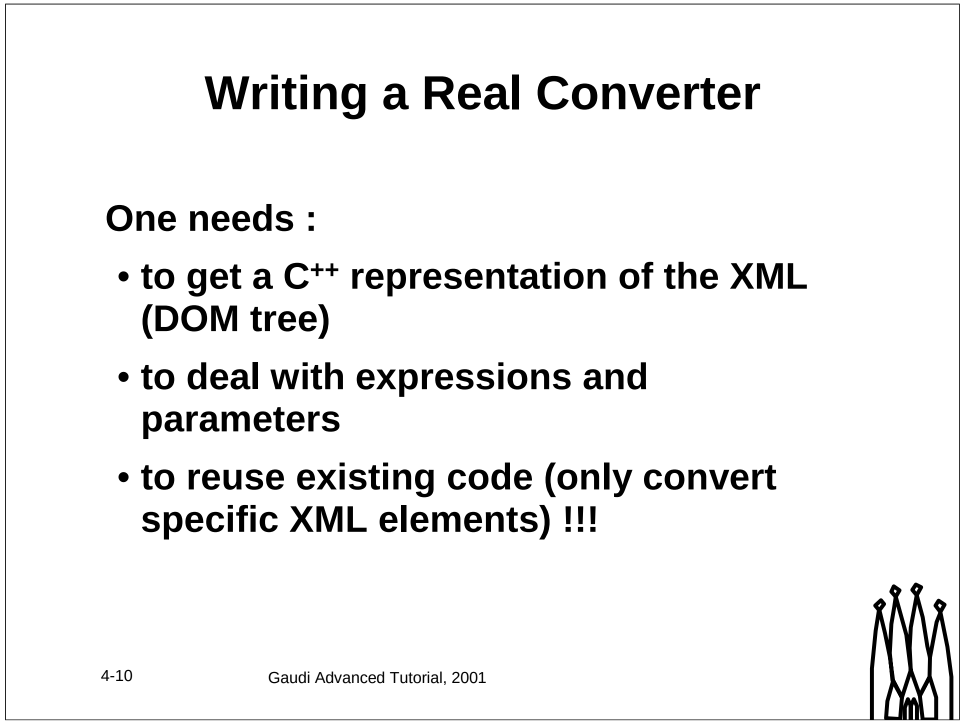
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM “extendedDtd.dtd”>
<DDDB>
  <detelem classID="7294" name="Head">
    <geometryinfo …/>
    <specific>
      <channelSet description=“…" name="Controls">
        <channels description="Inputs" nb="20"/>
        <channels description="Outputs" nb="150"/>
      </channelSet>
      <channelSet description=“…" name="Data">
        <channels description="head" nb="2000"/>
      </channelSet>
    </specific>
  </detelem>
</DDDB>
```

The <specific> element

This slide shows how to use the new elements defined in the extended DTD.

The first point is that the DOCTYPE element should now reference the extended DTD instead of the default one.

Then, all new elements must appear as children of a special tag of the default DTD called specific. This is the only place where one can insert new elements in the XML code. This element appears as child of the detelem element. One can add as many specific tag as he wants to a given detector element.



Writing a Real Converter

One needs :

- **to get a C^{++} representation of the XML (DOM tree)**
- **to deal with expressions and parameters**
- **to reuse existing code (only convert specific XML elements) !!!**

Writing a real converter

Now that we extended the DTD and we used the new elements in the XML, we have to write a converter to deal with this new code. This requires several steps :

- one should first get a representation of the XML in C^{++} to be able to get the data. This will be given by the xerces parser under the form of a DOM Tree.
- one should then deal with expressions and parameters that are used inside XML
- one should at last only write code for the new XML elements. The old one should still be converted by the default converters.

Accessing the XML : DOM

- **DOM is an interface to XML parsers based on tree representation of XML files**
- **This tree is essentially made of :**
  - **DOM_Document : the root of the tree**
  - **DOM_Element : the xml elements**
  - **DOM_Text : the bunches of text in XML**
  - **Comments, Attributes, ...**

Accessing the XML : DOM

DOM is an interface to XML parsers based on tree representation of XML files. This tree, called DOM tree, is a one to one map of the XML file, where nodes correspond to XMl elements and links to inclusion of elements inside another element.

The nodes of the tree are all DOM::Node objects but there exist different specializations of them :

- DOM_Document : this is the root of the tree, containing the DOCTYPE statement and the reference to the DTD
- DOM_Element : this is a regular XML element, with a name, attributes and sub elements
- DOM_Text : this is a text node, typically the value of one element, given as plain text in XML
- Comments, attributes and some other. These are not really needed for our purpose.

DOM Tree

```
<A>
  <B1>
    <C/>
  </B1>
  blabla
  <B2/>
</A>
```

XML File

```
Document
  A (Element)
    B1 (Element)
      C (Element)
    "blabla" (Text)
    B2 (Element)
```

DOM Tree

easy to browse

DOM Tree

Here is a simple and stupid example of a DOM tree. You can see how similar the XML file and the tree are. This makes the DOM tree very easy to browse inside converters to parse the specific parts.

Some DOM Methods

- DOMString DOM_Element.getNodeName ()
- DOMString DOM_Element.getAttribute (DOMString)
- DOMString DOM_Element.getNodeValue ()
- DOM_NodeList DOM_Element.getChildNodes ()
- DOM_NodeList DOM_Element.getElementsByTagName (string)
- unsigned int DOM_NodeList.getLength ()
- DOM_Node DOM_NodeList.item (unsigned int)

4-13 Gaudi Advanced Tutorial, 2001

Some DOM methods

This is part of the DOM API together with a short explanation of each method :

- DOMString DOM_Element.getNodeName () : returns the name of an element, ie the name of the tag. E.g., it returns “specific” for the <specific> tag.
- DOMString DOM_Element.getAttribute (DOMString) : returns the value of a given attribute or an empty string if the attribute does not exist. This value is the exact string that is inside the XML file.
- DOMString DOM_Element.getNodeValue () : returns the value of an element, ie the plain text between its opening and its closing.
- DOM_NodeList DOM_Element.getChildNodes () : returns a list of the children of an element.
- DOM_NodeList DOM_Element.getElementsByTagName (string) : returns a list of the children of an element with a given name.
- unsigned int DOM_NodeList.getLength () : return the length of a list of elements.
- DOM_Node DOM_NodeList.item (unsigned int) : returns an element of the list by index. This method returns a DOM_Node. One should cast it to get a DOM_Element.

Some Useful Features

- DOMString is DOM specific
  - ➢ call std::string dom2Std (DOMString) on the converter class to get a regular string
- Attribute have string value
  - ➢ double XmlCnvSvc::eval (std::string, boolean) must be called to get a double
  - ➢ the boolean tells whether to check for units or not. It can be omitted if true
  - ➢ call xmlSvc () on the converter to get the XmlCnvSvc service

4-14 Gaudi Advanced Tutorial, 2001

Some useful features

The previous slide gave some ideas of the DOM interface. Still, there are two small problems :

- all strings are DOMString instead of regular std::string
- the attribute values are only strings without any possibility of evaluating them to numeric types.

These problems are solved as follow :

- the method std::string dom2Std (DOMString) present on any converter allows to convert DOMString to std::string.
- the XmlCnvSvc has an eval method that compute a value of any expression and returns a double. It takes all parameters, constants and functions into consideration. This method can be either called with a single string parameter or with two parameters, the second one being a boolean. In this case, the boolean tells whether the method should check for the presence of a unit in the expression. The default value (when one parameter only) is true.
- on any converter, you can get a pointer to the XmlCnvSvc by calling xmlSvc().

Implementing the Converter

Real converter =

1. **extension of XmlUserDetElemCnv<DeType>**
2. **implementation of method**
   StatusCode i_fillSpecificObj (DOM_Element, DeType*)

- **`i_fillSpecificObj` is called once per direct child of <specific>**
- **the DOM_Element is given, the DeType object was created and must be populated**
- **all other elements (not inside <specific>) are automatically converted**

Implementing a converter

- Implementing a converter is pretty simple if one uses the templated class XmlUserDetElemCnv<DeType>, where DeType should be replaced by the name of the DetectorElement to be created. It is just a matter of filling a single method called i_fillSpecificObj.
- The i_fillSpecificObj method is automatically called during the parsing of the XML file for every XML element that is a direct child of the <specific> element. All other elements are parsed automatically but these may be new and are the responsibility of the user. This method is given the DOM_Element corresponding to the element found and the DetectorElement being currently built. This one was created but still need to be initialized, using the data contained in the DOM_Element.

Converter Example (1)

```
class XmlMyDetElemCnv :
  public XmlUserDetElemCnv<MyDetElem> {

public:
  XmlMyDetElemCnv (ISvcLocator* svc);
  ~XmlMyDetElemCnv() {}

protected:
  virtual StatusCode i_fillSpecificObj
    (DOM_Element childElement,
     MyDetElem* dataObj);
};

static CnvFactory<XmlMyDetElemCnv> s_Factory;
const ICnvFactory& XmlMyDetElemCnvFactory = s_Factory;

XmlMyDetElemCnv::XmlMyDetElemCnv(ISvcLocator* svc) :
  XmlUserDetElemCnv<MyDetElem> (svc) {
}
```

Converter example

This is an example of a converter implementation. There is not distinction here between .h and .cpp.

Several points need to be stressed :

- a constructor and a destructor must be defined, as shown here. One may add something inside but one must not forget to call the ancestor's constructor inside the new one.
- a factory must be declared, pretty identical to what we did in lesson 2.

Converter Example (2)

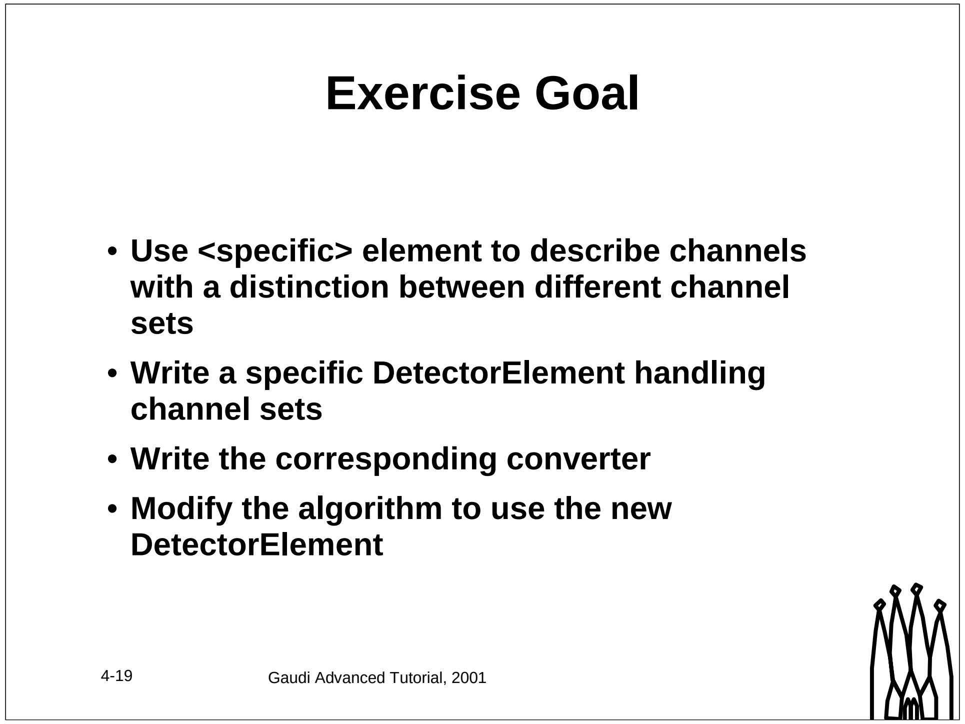
```
StatusCode XmlMyDetElemCnv::i_fillSpecificObj
  (DOM_Element childElement, MyDetElem* dataObj) {

  std::string elementName =
     dom2Std (childElement.getNodeName());

  if ("channelSet" == elementName) {
    const std::string name = dom2Std
      (childElement.getAttribute ("name"));
    const std::string description = dom2Std
      (childElement.getAttribute ("description"));
    dataObj->addChannelSet(name, description);
    …
  } else {
    …
}
```

Converter example (end)

Some more remarks :

- the only method to implement is I_fillSpecificObj.
- the usual way to implement it is to retrieve the element name first and to make a big if on it
- once the element is identified, one may retrieve some attribute values or even loop on the children elements. The data are then stored in the dataObj parameter, of type MyDetElem here.

Exercise 3



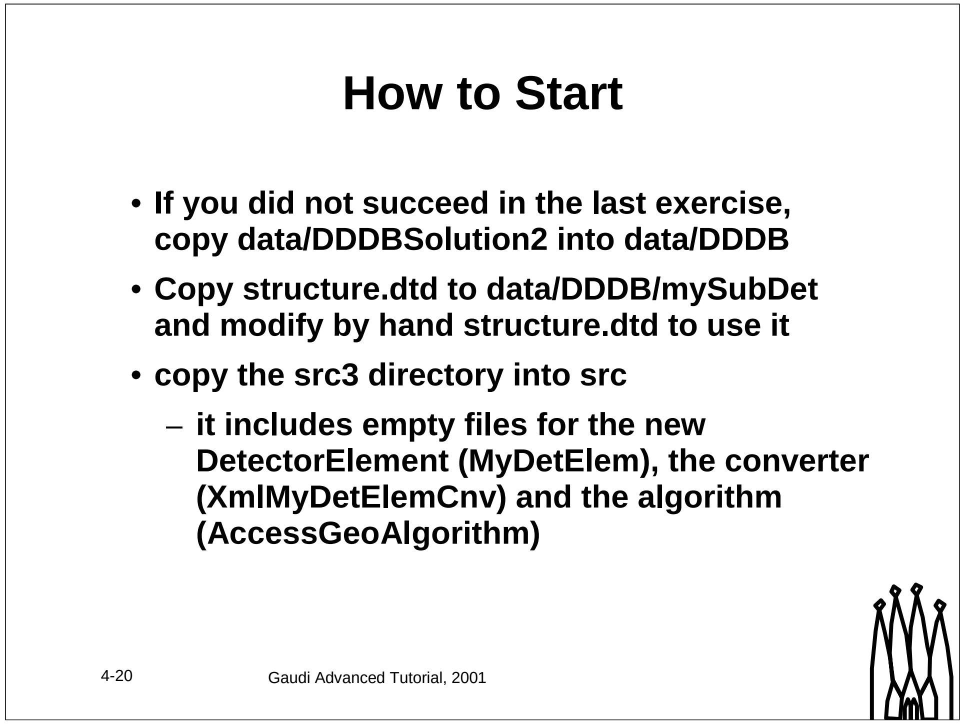
Exercise Goal

- Use <specific> element to describe channels with a distinction between different channel sets
- Write a specific DetectorElement handling channel sets
- Write the corresponding converter
- Modify the algorithm to use the new DetectorElement

4-19 Gaudi Advanced Tutorial, 2001

Exercise goal

The goal of the last exercise is to describe a bit more precisely the channels of the detector elements. This description will be stored in XML and handled inside DetectorElement objects. A new converter will be needed in order to use this new description inside the AccessGeoAlgorithm.



How to Start

- If you did not succeed in the last exercise, copy data/DDDBSolution2 into data/DDDB
- Copy structure.dtd to data/DDDB/mySubDet and modify by hand structure.dtd to use it
- copy the src3 directory into src
  - – it includes empty files for the new DetectorElement (MyDetElem), the converter (XmlMyDetElemCnv) and the algorithm (AccessGeoAlgorithm)

4-20 Gaudi Advanced Tutorial, 2001

How to start

This is what you have to do in order to start working on this second exercise :

• if you did not succeed in the second exercise, you should start with a valid geometry by copying the data/DDDBSolution2 directory into data/DDDB :

```
mv data/DDDB data/myDDDB2
cp –r data/DDDBSolution2 data/DDDB
```

• You must then copy the new DTD, called structure.dtd and provided in data to data/DDDB/mySubDet :

```
cp data/structure.dtd data/DDDB/mySubDet/
```

You also need to edit mySubDet/structure.xml by hand (I mean with emacs, not with XmlEditor) in order to use this new DTD. The only modification is to change the DOCTYPE line and to put “structure.dtd” instead of “../dtd/structure.dtd”.

• at last, you must copy the src3 directory into src :

```
mv src mysrc
cp –r src3 src
```

Don’t try here to reuse you old code many files will be missing. The new directory provides empty files for the AccessGeoAlgorithm, as well as for the converter and for the new DetectorElement called MyDetElem. As in the previous exercise, you may just fill the blanks.

Hints for XML

- The point is to describe more precisely the channels of a given detector element by dividing them into channel sets
- The DTD defines 2 new elements
  - channelSet with 2 attributes : name & description
  - channels with 2 attributes : nb & description
- A channelSet has many channels for children, channels has no children
- Examples of sets are Control and Data
- Examples of sub sets are Input and Output

Hints for XML

The point is to describe more precisely the channels of a given detector element by dividing them into channel sets. As you can see on the DTD presented on slide 8, the DTD defines two new elements :

- channelSet is a set of channels. It may contain several channels elements. It has a name and a description attribute. Example of such sets are Control or Data.
- channels represents several channels of the same kind. It has no child but a description and a number of channels. The distinction between several channels objects inside a given channel set allows, for example, to distinguish between input and output channels inside a given set.

Note that slide 9 gives you a good idea of what you should do !

Hints for MyDetElem

- MyDetElem should store the list of channel sets and for each of them the list of channels
- std::vector<…> should be used for list management and structs should be used for the definition of sets and channels
- The main two functionalities needed are
  - being able to get the number of channels of a given set from its name
  - being able to get the number of channels of channels objects from the name of the set and their index in the set

Hints for MyDetElem

MyDetElem is the new DetectorElement created to store all the list of channelSets and channels. This will be achieved via members of the object. One should use the std::vector object to store lists and iterators to retrieve them. Slide 8 in lesson 3 gives an example of the use of an iterator.

New methods should also be added to the MyDetElem object in order to retrieve the number of channels on the element itself, on a given set or even on a channels base. Note that one may only implement the retrieval of the number of channels for a given set as a first try.

Some More Hints

Concerning the converter

- **The only thing to do is to implement i_fillSpecificObj**
- **All methods that you need are given in these slides**

Concerning the algorithm

- **you should now retrieve a MyDetElem, so just use the methods you defined**

Some more Hints

Concerning the converter, just remember that you only need to fill I_fillSpecifcObj (and there is already a beginning) and that all useful methods are given in slides 13 and 14.

Concerning the algorithm, you get now a MyDetElem object from the transient store. You can thus use the methods you have just defined on it.

Good luck !