Extending detector elements and implications

Sebastien Ponce

31 August 2001

1 Overview

The goal of this documentation is to explain the different ways of extending and customizing detector elements. This customization is needed in order to manage the diversity of informations that different subdetectors have to handle.

We describe here three kinds of customization from the simplest but less flexible one to the most complicated, but most flexible one :

- the first one only allows the user to add parameters (or vectors of it) to the default detector element object in C++.
- the second one allows to define a new C++ object, inheriting from the default detector element. Some computation inside the object can thus be done, using the parameters defined in the first case.
- the last method allows the user to redefine the DTD used in the XML files and to manage structured data instead of just raw parameters.

Some prerequisites are needed in order to understand this document correctly, such as a good knowledge of the LHCb detector description framework. The new user should maybe read first the corresponding chapter ("Detector Description") in the Gaudi user's guide, available at *http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/Gaudi/*.

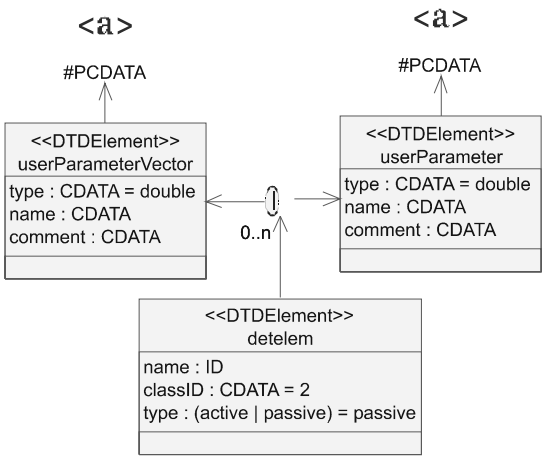
2 UserParameters

This is the simplest way to add specific information to a detector element without having to write too much code. It is based on the *userParameter* and *userParameterVector* tags defined in the structure DTD of the LHCb detector description files.

2.1 XML syntax

The *userParameter* and *userParameterVector* tags are children of the *detelem* tag. They can appear anywhere in the definition of a detector element. These tags have no child and 3 attributes :

- name : used to be retrieved in the C++code after conversion.



- type : this is the way the parameter should be handled in the converters. The value of the parameter itself is always a string but the converters will try to convert it into a double or an integer if this attributes equals "double" or "int". Otherwise, it will be taken as a string.
- comment : this is supposed to keep a quick description of the meaning of the parameter and of its use.

The value of the parameter itself is the value of the XML element. In case of a vector, the values must be separated by spaces and/or carriage returns.

Here are examples of the usage of user parameters :

```
<detelem name="Ecal">
   ....
   <userParameter name="CodingBit" type="int"> 6 </userParameter>
   <userParameter name="EtSlope"
                  type="double"
                  comment="10 + .3*7 = 12.1 at 300 mrad">
       7.*GeV
    </userParameter>
    <userParameter name="detelem_name">
       Ecal inner cylinder
    </userParameter>
    <userParameterVector name="aVector" type="int">
       10 20 30
       40 50 60
    </userParameterVector>
</detelem>
```

If you want more information, the full XML syntax can be found in a dedicated documentation : "The LHCb detector description DTD", by Sebastien Ponce, at *http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/DetDesc/lhcbdtd.html.*

2.2 Usage in C++

Once user parameters are defined in XML, they are converted by the regular converter for detector elements and are reachable in the C++ code using the following methods of the class `DetectorElement` :

```
virtual std::vector<std::string> userParameters();
virtual std::vector<std::string> userParameterVectors();

virtual double userParameter (std::string name);
virtual std::string userParameterAsString (std::string name);
virtual int userParameterAsInt (std::string name);
virtual double userParameterAsDouble (std::string name);
virtual std::string userParameterType (std::string name);
virtual std::string userParameterComment (std::string name);

virtual std::vector<double> userParameterVector (std::string name);
virtual std::vector<std::string> userParameterVectorAsString (std::string name);
virtual std::vector<int> userParameterVectorAsInt (std::string name);
virtual std::vector<double> userParameterVectorAsDouble (std::string name);
```


```
virtual std::string userParameterVectorType (std::string name);
virtual std::string userParameterVectorComment (std::string name);
```

The methods' names should be self-comprehensive. To be short, the two first methods return a vector containing the names of every defined user parameters or parameter vectors. Then the two blocks of methods are accessors to parameters and parameter vectors. Depending on its type, one can retrieve the parameter as string, as an int or as a double. The type and comment associated to a given parameter can also be retrieved. At last, note that the default accessor returns parameters as doubles, it is an equivalent of the AsDouble accessor.

If one tries to retrieve a non-existent parameter, a `DetectorElementException` is raised. This also occurs if one tries to retrieve a parameter with the wrong type (string as double or int, double as int). However, ints can be retrieved as doubles and every parameter can be retrieved as string.

2.3 Examples

Here are examples of usage for userParameters in C^{++}. The first one displays every parameter of a given element, with type, value and comment. The second goes with the XML code and computes the total length of the described detector element.

```
std::vector<std::string> parameterList = myDetElem->userParameters();
std::vector<std::string>::iterator it;
for (it = parameterList.begin(); it != parameterList.end(); it++) {
  log << MSG::INFO << "userParameter " << *it << " ("
      << myDetElem->userParameterType(*it) << ") = "
      << myDetElem->userParameterAsString(*it) << " -- "
      << myDetElem->userParameterComment(*it) << endreq;
}
```

```
<detelem name="MyDetelem">
  ....
  <userParameterVector
    name="SubpartLengths"
    type="double"
    comment="this is the list of the lengths for every subpart of this element">
        10.56*mm 4.53*cm 7.23*mm 6*mm
  </userParameterVector>
</detelem>
```

```
double totalLength = 0;
std::vector<double>::iterator it;
std::vector<double> subpartLengths =
  myDetElem->userParameterVectorAsDouble ("SubpartLengths");
for (it = subpartLengths.begin(); it != subpartLengths.end(); it++) {
  totalLength += *it;
}
```


3 Customizing the detector element object

3.1 Why to customize detector elements

The preceeding section showed us how to define user parameters inside a detector element. But the `DetectorElement` C^{++} object remains unchanged and not customizable. Thus, we are not allowed to define our own private members, nor to add usefull methods, be it to reach our parameters in a more friendly way, or to compute new ones on the fly.

All these problems find their solution in the ability to define new C^{++} classes, inheriting from the original `DetectorElement` class. This has for consequence that the generic converter provided with Gaudi can no more do the job, since it is not aware of the existence of this new class. Thus, we have to create a new converter, by customizing a bit the default one.

3.2 The class XmlUserDetElemCnv<DeType>

This is a templated class that allows the user to define his own converter for his given type of detector element. It actually inherits the behavior of the default converter, which avoids the user to rewrite existing code. The only difference here is that it creates an object of type `DeType` instead of a regular `DetectorElement`.

The parameter class `DeType` has to deal with some constraints :

- it must inherit from `DetectorElement`.
- it must have a default constructor with no parameters.
- it must have a new classID, which has to be unique in whole LHCb software. See *http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/EventModel/CLID.htm* to see the list of existing ones and request a new one

Appart from these three points, you have full freedom to define you own class and add whatever methods or members you may want. In addition, you could override the methods of `DetectorElement`. This is not recommanded at all except for the `initialize` method. This one is actually a kind of hook for the user. It is called by the converter just after the creation of the object and before its first use. It allows the user to initiate members, using for example the user parameters parsed from the XML.

You can now define your own converter, creating your own objects. Here are some more tricks you may want to know about :

- you must define a constructor and a destructor for the converter, even if it does nothing (well, don't forget to call the ancestor's constructor in your new constructor).
- a factory must be defined for your new converter in order that the conversion service be able to build it. It is two lines of code only that you can cut and paste from the examples, replacing Vertex by what you need. More information about converters and factories are given in the Gaudi user guide (chapter "Converters"). It is available at *http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/Gaudi/*.

3.3 Examples

Here is the minimum code to define a new converter for a specific detector element (`DeVertexDetector` in our case).

This is the definition of the new detector element :

```
#include "DetDesc/DetectorElement.h"
const CLID& CLID_DEVertex = 9999;

class DeVertexDetector : public DetectorElement {
public:
  DeVertexDetector();
  ~DeVertexDetector();

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

DeVertexDetector::DeVertexDetector() {}
DeVertexDetector::~DeVertexDetector() {}

const CLID& DeVertexDetector::clID() const {
  return classID();
}

const CLID& DeVertexDetector::classID() {
  return CLID_DEVertex;
}
```

And this is the converter :

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

class XmlVertexCnv : virtual public XmlUserDetElemCnv<DeVertexDetector> {
public:
  XmlVertexCnv (ISvcLocator* svc);
  ~XmlVertexCnv(){}
}

static CnvFactory< XmlVertexCnv > vertexdecnv_factory;
const ICnvFactory& XmlVertexCnvFactory = vertexdecnv_factory;

XmlVertexCnv::XmlVertexCnv (ISvcLocator* svc) :
  XmlUserDetElemCnv<DeVertexDetector> (svc) {
}
```

This first example defines a new detector element and its associated converter but does not add any functionnality to the default one. This is basically of no use. Here is an extension of this example, using user parameters as seen in Section 2 on page 2.

First, the XML code :

```
<detelem name="VertexDetector">
```


```
  ....
  <userParameterVector
    name="SubpartLengths"
    type="double"
    comment="this is the list of the lengths for every subpart of this element">
        10.56*mm 4.53*cm 7.23*mm 6*mm
  </userParameterVector>
  <userParameterVector
    name="SubpartNames"
    comment="this is the list of subparts of this element">
        part1  part2  part3  part4
  </userParameterVector>
</detelem>
```

Then, the corresponding detector element object in C^{++} :

```
#include "DetDesc/DetectorElement.h"
const CLID& CLID_DEVertex = 9999;

class DeVertexDetector : public DetectorElement {
public:
  DeVertexDetector();
  ~DeVertexDetector();
  const CLID& clID() const;
  static const CLID& classID();

  virtual StatusCode initialize();
  double getTotalLength();
  double getSubPartLength(std::string name);

private:
  std::map<std::string, double> subparts;
}

DeVertexDetector::DeVertexDetector() {}
DeVertexDetector::~DeVertexDetector() {}

const CLID& DeVertexDetector::clID() const {
  return classID();
}

const CLID& DeVertexDetector::classID() {
  return CLID_DEVertex;
}

StatusCode DeVertexDetector::initialize() {
  std::vector<double> subpartLengths =
    userParameterVectorAsDouble ("SubpartLengths");
  std::vector<std::string> subpartNames =
    userParameterVectorAsDouble ("SubpartNames");
  if (subpartLengths.size() != subpartNames.size()) { ERROR !!! }

  std::vector<double>::iterator itLength;
  std::vector<std::string>::iterator itName;
  for (itLength = subpartLengths.begin(), itNames = subpartNames.begin();
       itLength != subpartLengths.end(), itNames != subpartNames.end();
```


```
       itLength++, itNames++) {
    subparts[itNames*] = itLength*;
  }
}

double DeVertexDetector::getTotalLength() {
  double result = 0.0;
  for (std::map<std::string, double>::iterator it = subparts.begin();
       it != subparts.end(); it++) {
    result += it->second;
  }
  return result;
}

double DeVertexDetector::getLength (std::string name) {
  return subparts[name];
}
```

The converter remains unchanged.

4 Extending the DTD

4.1 Why to extend the DTD

In Section 2, we learnt how to add some user specific parameters to detector elements in XML. In Section 3, we learnt how to customize the C^{++} detector element. But there is still no way to add something else than simple parameters to the XML definition of a detector element. One may for example want to define new XML children tags for the *detelem* tag in which he would put his data in a formatted way.

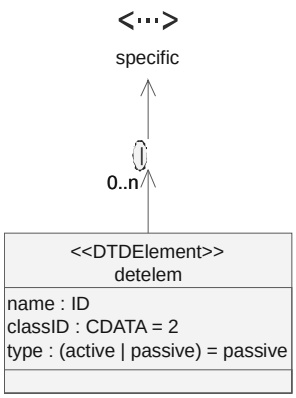
This was actually foreseen and is available through the usage of the *specific* tag. It allows the user to extend the default DTD of the XML file where his detector element resides and to use new tags. These tags will then be parsed by a user defined converter and used during the building of detector elements.

4.2 XML syntax

We will first describe how to use the *specific* tag. Then we'll deal with the definition of a user's specific DTD for a given XML file.

4.2.1 The <specific> tag

The *specific* tag is defined in the default structure DTD of the LHCb detector description as a possible child of the *detelem* tag. It can appear anywhere in the definition of a detector element and may have any number of children of any type. However, it has no attributes. This tag was thought as a hook for users who want to add their own defined tags inside a detector element. Instead of puting them directly in the detector element, they should simply be wrapped inside the tag *specific*.



Here is a small example of it :

```
<detelem classID="9990" name="MStation01">
  <author> Sebastien Ponce </author>
  <geometryinfo ... />
  <specific>
    <Al_plate_thickness value="1.1111*mm"/>
    <pad_dimensions>
      <padX length="2*mm"/>
      <padY length="4*mm"/>
    </pad_dimensions>
  </specific>
</detelem>
```

If you want more information, the full XML syntax can be found in a dedicated documentation : "The LHCb detector description DTD", by Sebastien Ponce, at *http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/DetDesc/lhcbdtd.html.*

4.2.2 Defining a new DTD

We've just seen that the user is allowed to add any new XML tag to a detector element, using the *specific* tag. But of course, the new added tags have to be inserted into the dtd of the file for the XML to be valid. There are two ways of modifying the dtd : defining an internal dtd or writing a new external DTD kind of inheriting from the old one.

4.2.2.1 Internal DTD

As you know, a valid XML file begins with these lines :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM "../DTD/structure.dtd">
```

It defines the XML versino and encoding and gives the DTD that is used in this XML file. (in our case, *../DTD/structure.dtd)*. But one could also define an internal DTD for an XML file by writing this :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB [
  <!-- This is the internal DTD -->
```


```
  <!ELEMENT myTag EMPTY>
  <!ATTLIST myTag myAttribute CDATA #REQUIRED>
]>
```

In this small example, the only tag allowed in the XML file will be *myTag* since it is the only thing present in the DTD. In our case, one may want to include *structure.dtd* as is and also define the new tag, as an extension of it. Here is the syntax :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM "../DTD/structure.dtd" [
  <!-- This is added to the external DTD -->
  <!ELEMENT myTag EMPTY>
  <!ATTLIST myTag myAttribute CDATA #REQUIRED>
]>
```

Now, you can safely use the structure DTD and use the new tag *myTag*. Of course, this supposes that the structure DTD has some tag that allows using *myTag* as a child. It is precisely the case of the *specific* tag.

4.2.2.2 External DTD

In Section 4.2.2.1, we saw how to extend the DTD for a given single file. But you may want to extend the DTD once and reuse it in several files without duplicating the new code in every file. Here is what you can do.

Define a new DTD file, say myDTD.dtd, with this code :

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

<!-- This includes the structure DTD -->
<!ENTITY % dtdForStructure SYSTEM "../DTD/structure.dtd">
%dtdForStructure;

<!-- This is added to the DTD -->
<!ELEMENT myTag EMPTY>
<!ATTLIST myTag myAttribute CDATA #REQUIRED>
```

This includes the file structure.dtd inside the new DTD file, before defining a new tag. The include mechanism works exactly as #include in C or C^{++} files. Thus, you can kind of inherit from a dtd and extend it. You can then use the new DTD as usual by writing as first lines of your XML file :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM "myDTD.dtd">
```


4.3 C++ implication : adapting the converter

We already described in Section 3 on page 5 how the C++ object representing the detector element could be customized to include new members and new methods. There is nothing new here.

However, the converter that creates this object has to be changed in order to deal with the new DTD of the XML files. This can be done at low cost using the `XmlUserDetElemCnv` templated class. However, this requires a good knowledge of the DOM interface.

4.3.1 the DOM interface

DOM is the name of an interface to XML parsers. In this interface, the XML parser is supposed to build a C++ tree representation of the XML file that is then easily browsable.

The tree is built from the following classes :

- `DOM_Document` : this is the root of each XML document. It corresponds to the <? xml ...> first line of the file
- `DOM_Element` : this represents one XML element, ie one tag of the file and its content. It thus includes the list of children, the list of attributes and the value of the tag itself represented as a `DOM_Text` child.
- `DOM_Text` : this is used for values of tags, that are written in plain text inside the XML file.
- `DOM_Attr` : this represents an attribute and its value.
- `DOM_Comment` : this represents a comment.

Here is an example of the tree built for a very simple XML file.

Here is the XML file :

```
<A>
  <B1 myAt="bla">
    <C/>
  </B1>
  blabla
  <B2/>
</A>
```

And here is the resulting tree :

```
DOM_Document
  DOM_Element A
    DOM_Element B1
      DOM_Attr "myAt", value is "bla"
      DOM_Element C
    DOM_Text "blabla"
    DOM_Element B2
```

As you can see, nothing very complicated here. The point is that you can then easily retrieve any data from the XML file using the methods defined on the nodes of the tree. We'll describe some of them but you should go to the xerces documentation to have all

details (see *http://xml.apache.org/xerces-c/apiDocs/index.html*) Take care that some of the methods are defined in the DOM_Node class that is the common ancester for every node of the tree.

4.3.1.1 Methods on `DOM_Node` :

- `DOM_NamedNodeMap getAttributes ()` : this returns the list of all attributes for a given node. This list may be empty. The return value is a kind of list object where items can be retrieved by index (`item` method) or by name (`getNamedItem`). The length of the list can be obtained with `getLength`.
- `DOM_NodeList getChildNodes ()` : this returns the list of all children of a given node. This list may be empty. It contains a mix of elements, attributes and text children. The return value is a kind of list object where items can be retrieved only by index (`item` method). The length of the list can be obtained with `getLength`.
- `DOM_Node getParentNode ()` : this returns the parent of a given node in the tree, as a `DOM_Node`.
- `DOMString getNodeName ()` : this returns the name of a given node.
- `DOMString getNodeValue ()` : this returns the value of a given node.
- `short getNodeType ()` : this returns the type of a given node. Correct values are ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, COMMENT_NODE or DOCUMENT_NODE among others.

4.3.1.2 Methods on `DOM_Element` :

- `DOM_Attr getAttributeNode (const DOMString &name)` : this retrieves the node of an attribute of the element for a given attribute's name.
- `DOMString getAttribute (const DOMString &name)` : this retrieves the value of an attribute of the element for a given attribute's name.
- `DOM_NodeList getElementsByTagName (const DOMString &name)` : this returns a list of all descendant elements with a given tag name.

4.3.1.3 Methods on `DOM_Document` :

- `DOM_Element getElementById (const DOMString &elementId)` : this returns the `DOM_Element` whose ID is given by elementId.

4.3.1.4 The `DOMString` object

You way have realized in the previous sections that the DOM interface has its own string type, called `DOMString`. The problem is that there is no default converter from `DOMString` to the regular `std::string`. This is fixed by the method :

```
static const std::string dom2Std (DOMString domString);
```

of the converter classes.

4.3.1.5 Expression evaluator

You may also have realized that values of attributes or elements are given in the DOM interface as strings even when they are numbers. This is the normal way XML works. These strings can however be evaluated by using one of the 2 methods of the `XmlCnvSvc` object :

```
virtual double eval (const char* expr, bool check = true) const;
virtual double eval (const std::string& expr, bool check = true) const;
```

The `XmlCnvSvc` object can be obtained in any converter by a call to `xmlSvc()`. Note that the boolean parameter specifies whether the evaluator should check that the expression has a dimension or not. If it is true, it checks it and you may get a warning.

4.3.2 Hooks for converting specific elements

We've just learnt how to access information contained in an XML file using a tree provided by the xerces parser. Fine. But where should we use it in the converters ?

The answer is in the following method :

```
StatusCode i_fillSpecificObj (DOM_Element childElement, DeType* dataObj);
```

This one is defined as virtual in class XmlUserDetElemCnv and thus should be overridden in every user defined converter. Its principle is simple : it is called by the XML parser service everytime a tag is found directly inside a *specific* tag. The parameters are then the `DOM_Element` corresponding to the tag found and the C^{++} detector element object being built when this occured.

Thus, for the following XML file :

```
<detelem classID="9990" name="MStation01">
  <author> Sebastien Ponce </author>
  <geometryinfo ... />
  <specific>
    <Al_plate_thickness value="1.1111*mm"/>
    <pad_dimensions>
      <padX length="2*mm"/>
      <padY length="4*mm"/>
    </pad_dimensions>
  </specific>
</detelem>
```

The method will be called twice, once for *Al_plate_thickness* and once for *pad_dimensions*. It will not be called for *padX* or *padY.*

4.4 Examples

There is only one example here that deals with the following XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM "../DTD/structure.dtd" [
  <!-- This is added to the external DTD -->
  <!ELEMENT Al_plate_thickness EMPTY>
  <!ATTLIST Al_plate_thickness value CDATA #REQUIRED>
  <!ELEMENT pad_dimensions (padX, padY)>
  <!ELEMENT padX EMPTY>
  <!ATTLIST padX length CDATA #REQUIRED>
  <!ELEMENT padY EMPTY>
  <!ATTLIST padY length CDATA #REQUIRED>
]>
<detelem classID="9990" name="MStation01">
  <author> Sebastien Ponce </author>
  <geometryinfo ... />
  <specific>
    <Al_plate_thickness value="1.1111*mm"/>
    <pad_dimensions>
      <padX length="2*mm"/>
      <padY length="4*mm"/>
    </pad_dimensions>
  </specific>
</detelem>
```

Here is the corresponding C++ detector element :

```
#include "DetDesc/DetectorElement.h"
const CLID& CLID_DEVertex = 9999;

class DeStation : public DetectorElement {
public:
  DeStation();
  ~DeStation();
  const CLID& clID() const;
  static const CLID& classID();

  double getX();
  double getY();
  double getThickness();

  void setX (double x);
  void setY (double y);
  void setThickness (double t);

private:
  double x, y, t;
}

DeVertexDetector::DeStation() : x(0), y(0), t(0) {}
DeVertexDetector::~DeStation() {}

const CLID& DeStation::clID() const {
  return classID();
}

const CLID& DeStation::classID() {
  return CLID_DEVertex;
```


```
}

double DeStation::getX() {
  return x;
}

double DeStation::getY() {
  return y;
}

double DeStation::getThickness() {
  return t;
}

void DeStation::setX (double newX) {
  x = newX;
}

void DeStation::setY (double newY) {
  y = newY;
}

void DeStation::setThickness (double newT) {
  t = newT;
}
```

And finally, here is the converter :

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

class XmlStationCnv : virtual public XmlUserDetElemCnv<DeStation> {
public:
  XmlStationCnv (ISvcLocator* svc);
  ~XmlStationCnv(){}

  virtual StatusCode i_fillSpecificObj (DOM_Element childElement,
                                        DeMuonStation* dataObj);
}

static CnvFactory< XmlStationCnv > stationcnv_factory;
const ICnvFactory& XmlStationCnvFactory = stationcnv_factory;

XmlStationCnv::XmlStationCnv (ISvcLocator* svc) :
  XmlUserDetElemCnv<DeStation> (svc) {
}

StatusCode XmlStationCnv::i_fillSpecificObj (DOM_Element childElement,
                                             DeMuonStation* dataObj) {
  MsgStream log (msgSvc(), "XmlMuonStationCnv");

  std::string tagName = dom2Std (childElement.getNodeName());
  // We expect here two possible tags : Al_plate_thickness and pad_dimensions
  if ("Al_plate_thickness" == tagName) {
    // get a value of the 'value' attribute
    const std::string value = dom2Std (childElement.getAttribute ("value"));
```


```
      if (!value.empty()) {
        dataObj->setThickness (xmlSvc()->eval(value));
      }
    } else if ("pad_dimensions" == tagName) {
      // get value attributes from tags 'padX' and 'padY'
      DOM_NodeList padXList = childElement.getElementByTagName ("padX");
      DOM_NodeList padYList = childElement.getElementByTagName ("padY");
      if (padXList.getLength() > 0) {
        if (padXList.getLength() > 1) {
          log << MSG::ERROR
              << "Too many padX tags in pad_dimensions. Invalid XML !!!"
              << endreq
        }
        DOM_Element padXElement = (DOM_Element) padXList.item(0);
        const std::string value = dom2Std (padXElement.getAttribute ("value"));
        if (!value.empty()) {
          dataObj->setX (xmlSvc()->eval(value));
        }

      if (padYList.getLength() > 0) {
        if (padYList.getLength() > 1) {
          log << MSG::ERROR
              << "Too many padY tags in pad_dimensions. Invalid XML !!!"
              << endreq
        }
        DOM_Element padYElement = (DOM_Element) padYList.item(0);
        const std::string value = dom2Std (padYElement.getAttribute ("value"));
        if (!value.empty()) {
          dataObj->setY (xmlSvc()->eval(value));
        }
      }
    } else {
      // Unknown tag, a warning message is issued here
      log << MSG::WARNING << "Unknown tag found in specific : "
          << tagName << endreq;
    }
    return StatusCode::SUCCESS;
  }
```