LHCb note 2008-046

LPHE note 2008-009

Updated description of the Silicon Tracker detector element tree

M. Needham
École Polytechnique Fédérale de Lausanne

September 11, 2008

Abstract

The structure and usage of the Silicon Tracker detector element tree classes in Summer 2008 is discussed.

1 Introduction

At the time of the DC '06 [1] data challenge the XML description of the two sub-detectors that comprise the Silicon Tracker project — the Trigger Tracker (TT) and Inner Tracker (IT) was updated [2]. The corresponding update to the C++ detector element code is described in [3]. Since then the XML and code have further evolved and additional functionality has been added. This note updates [3] and describes the situation in Summer 2008 [1].

Fig. 1 shows the structure of the IT detector element tree in the XML whilst Fig 2 shows the corresponding tree for TT. For alignment studies this structure is mirrored in the structure of the C++ classes. Though there are differences between the two trees several common features can be seen:

[1] This corresponds to Brunel v33r1.

- Both trees contain elements which have common functionality: stations, layers and sectors. This suggests to develop base classes for these elements.
- Below the top level detector elements all the other elements of the tree can be expected to behave in similar ways. Therefore, a common base class has been developed.

It should also be noted that though there are many levels to the tree the user will typically only interact with the base classes representing the top level detector element and the sector. In the following sections the classes in the tree is described together with examples of their use. This note gives a snapshot of the code and its usage and complements the automatically generated doxygen documentation. All the code described can be found in the package **ST/STDet**.

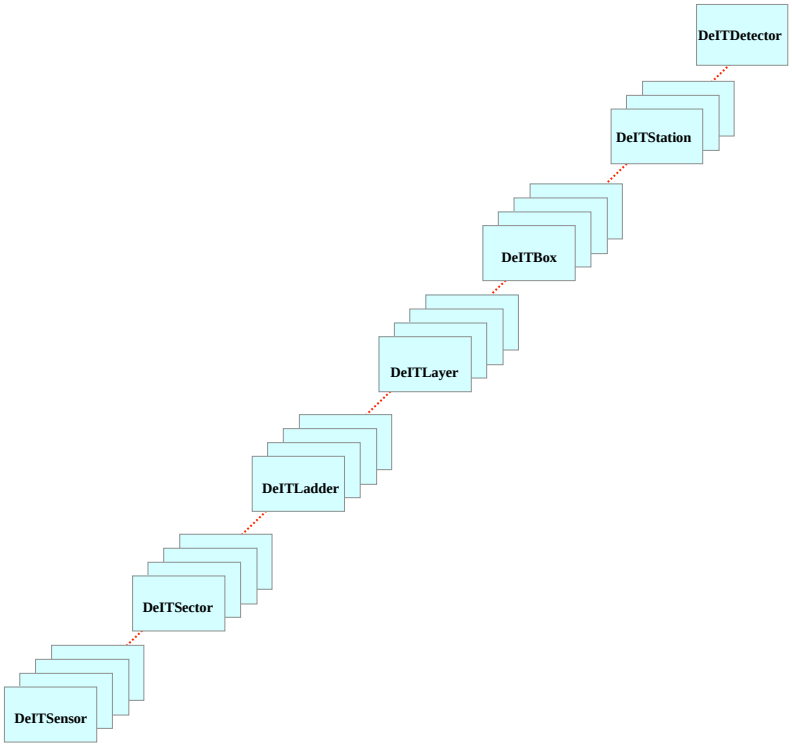

Figure 1: IT detector element structure.

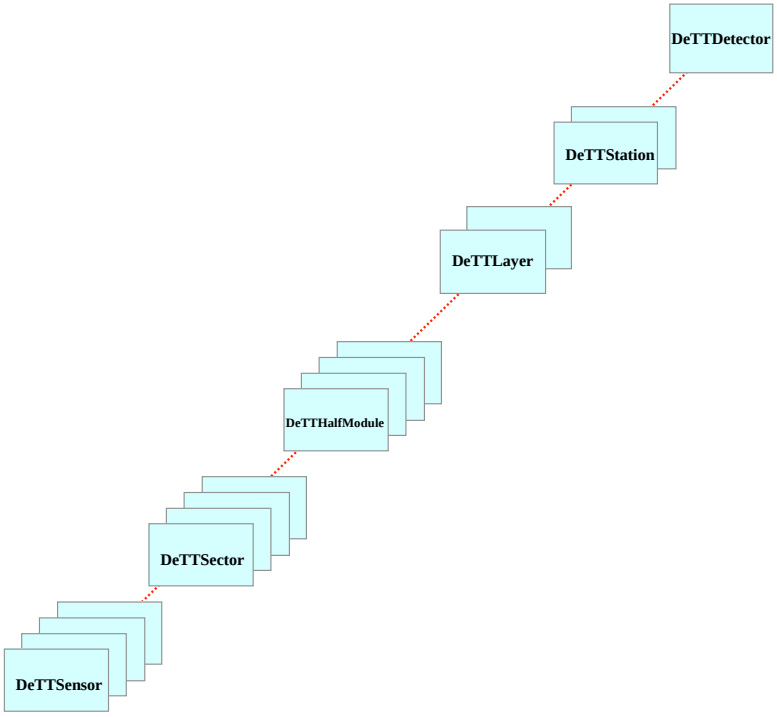

Figure 2: TT detector element structure.

2 Top Level DetectorElement

Fig. 3 shows the class structure for the top-level detector elements. A base class **DeSTDetector** expresses the common functionality between the IT and TT detectors. Concrete implementations for the two detectors inherit from this class. *Nota Bene*, most of the differences between the two classes are related to initialization. Therefore, it should not be necessary to use either the **DeITDetector** or **DeTTDetector** classes directly. The base class allows direct access to its children — the stations and also to its lowest level descendents — the sectors. The following code fragment shows how to get and use a top-level TT detector element:

```
#include "STDet/DeSTDetector.h"

// get the top-level element
DeSTDetector* tracker =
   getDet<DeSTDetector>(DeSTDetLocation::location(``TT''));
```

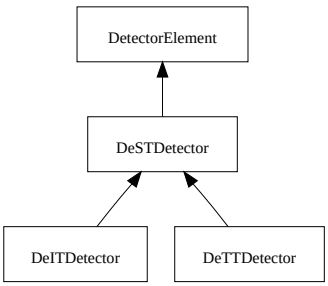

Figure 3: Class diagram for the top level detector elements.

The most common use of this class will be to find the lowest level **DetectorElement** corresponding either a detector channel or containing a given point :

```
// find the lowest detector element corresponding to a point
Gaudi::XYZPoint& aPoint;
DeSTSector* aSector = tracker->findSector(aPoint);
if (aSector !=0){
  // we found the sector
}

// find the lowest detector element corresponding to a channelid
STChannelID aChan;
DeSTSector* aSector = tracker->findSector(aChan);
if (aSector !=0){
  // we found the sector
}
```

It is possible to get a flat vector of all sectors contained in the **DeSTDetector** tree and also to select all the sectors corresponding to a list of **STChannelID**':

```
// flat vector of all descendents
const DeSTDetector::Sectors& tSectors = m_tracker->sectors();
DeSTDetector::Sectors::const_iterator iterS = tSectors.begin();
for (; iterS != tSectors.end() ; ++iterS) {
  // do something
}

// find vector of channels corresponding to list of channels
std::vector<LHCb::STChannelID> chans;
std::vector<DeSTSector*> sectors = tracker->findSectors(chans);
```

For use in the track fit a method is provided that returns the 3D trajectory [4] corresponding to a readout strip [2]:

[2]Internally, the work is delegated to the **DeSTSector** class.

```
// get a trajectory - used by tracking
double distanceToStrip;
LHCb::LHCbID chan;
std::auto_ptr<LHCb::Trajectory> traj =
    tracker->trajectory(aChan, distanceToStrip);
```

Finally, the fraction of the detector that is operational can be obtained:

```
// return working fraction
double fracActive = tracker->fractionActive();
```

3 The other DetectorElements

3.1 Base class

All other detector elements derive from the base class **DeSTBaseElement**. This provides the following functionality:

- Short-cuts for frame transformations between the local and global LHCb coordinate system.
- A short-cut to test whether a point is inside the detector element volume.
- The centre of the detector element in the global detector frame.

In addition, each detector element is uniquely identified by a '**STChannelID**'. This allows fast linking of detector and data items. Examples of how to use this common functionality are given in the following sections.

3.2 DeSTSector

The other important detector class is the **DeSTSector** (Fig. 4). The base class represents a readout sector, that is to say a group of sensors read out via one hybrid in the IT or TT. Internally the class manages a vector of **DeSTSensor** (see Section 3.3). Concrete implementations for the two detectors inherit from this class. Most users will only have to deal with the base class. There is one exception to this. The **DeTTSector** class has a

method DeTTSector::row() [3] which is not present in the IT class. To access this function it is necessary to make a dynamic_cast of **DeSTSector**. The

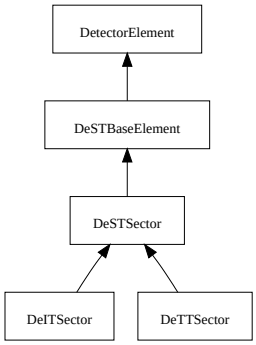

Figure 4: Class diagram **DeSTSector**.

following code fragments illustrate some of the functionality **DeSTSector** class:

```
// ...
Gaudi::XYZPoint aPoint; // point in global frame
DeSTSector* aSector;
if (aSector->isInside(aPoint) == true){
  // point is inside the sector - make global to local transformation
  Gaudi::XYZPoint lPoint = aSector->toLocal(aPoint);
}

// get the ''channel id'' corresponding to the sector
const LHCb::STChannelID chan = aSector->elementID();

// get the strip pitch
const double pitch = aSector->pitch();
```

The class provides two methods that are useful for track reconstruction. The first allows direct access to the strip trajectory [4] for the track fit. For readout sectors consisting of single sensor the trajectory is a **LineTraj**. For multi-sensor readout sectors the trajectory is a **PiecewiseTrajectory** that represents the 'broken' line of the sensors. In addition, a simplified trajectory is provided for use by the pattern recognition. This returns the parameters of a line that approximates the strip geometry. The code fragments below show how they are used:

```
// Trajectory
DeSTSector* aSector;
```

[3] A row corresponds to a set of sensors at a common y.

```
LHCb::STChannelID aChan;
double distanceToStrip;
std:auto_ptr<LHCb::Trajectory> traj =
   aSector->trajectory(aChan,distanceToStrip);

unsigned int strip;
double dxdy, double dzdy;
double xAtYEq0, double zAtYEq;
double ybegin, double yend;
trajectory(strip,distanceToStrip,dxdy,dzdy,
           xAtYEq0, zAtYEq0, ybegin,yend);
```

The class also provides a set of functions to check whether a point is within the active or dead region of the detector. For example it is possible to test whether a point in the LHCb frame is inside the active area of the sensor:

```
// in global frame
Gaudi::XYZPoint aPoint; // point in global frame
DeSTSector* aSector;
Gaudi::XYZPoint tol(0.2*mm,0.2*mm, 0.); // tolerances
if (aSector->globalInActive(aPoint, tol){
  // do something
}


// test if point is inside dead region
if (aSector->globalInBondGap(aPoint) == true){
  // in bond gap to bad no signal
}
```

Finally, the class provides information on dead and faulty strips. The following types of faults are foreseen and defined in an enumeration which is summarized in Table 1.

A hierarchical approach is used to store and access this information as the following code fragments illustrate.

```
// check if sector is OK
DeSTSector* aSector;
if (aSector->sectorStatus() == DeSTSector::OK ){
  // sector is OK
}

// instead of checking the sector check the beetle
LHCb::STChannel achan;
if (aSector->beetleStatus(achan) == DeSTSector::OK ){
  // beetle is OK
}

// or strip
LHCb::STChannel achan;
if (aSector->stripStatus(achan) == DeSTSector::OK ){
  // strip is OK
}
// short cut for ok strip
```

Type	Value	Definition
OK	0	No problems
Open	1	Strip is interupted
Short	2	Two strips are shorted
Pinhole	3	Strip has pinhole between Al and implant
ReadoutProblems	4	Not Readout
NotBonded	5	Not connected to hybrid
LowGain	6	Low Signal
Noisy	7	High noise
OtherFault	9	—
Dead	10	As a doornail [5]

Table 1: Dead strip enumeration definitions.

```
if (isOKStrip(achan) == true){
 // short cut for strip is ok
}

const double fracWorking = aSector->fractionActive();
```

The status of all strips or beetles on a module can also be obtained:

```
DeSTSector* aSector;
std::vector<DeSTSector::Status> beetles = aSector->beetleStatus();
std::vector<DeSTSector::Status> strips = aSector->stripStatus();
```

3.3 DeSTSensor

Fig. 5 shows the class diagram for the **DeSTSensor**. Again a base class expresses the commonality between the IT and TT. This class represents a single silicon wafer. It can be accessed via the **DeSTSector**:

```
DeSTSector* aSector;
const DeSTSector::Sensors& sensors = aSector->sensors();
```

Typically, most geometry questions can be answered by the **DeSTSector** class either directly or via the Trajectory method and hence the user will not need to use this class. The main exceptions to this are related to finding where a particle entered and exited a readout sector:

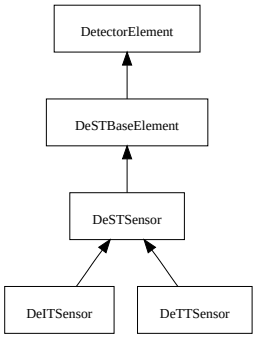

Figure 5: Class diagram **DeSTSensor**.

```
DeSTSensor* aSensor;
Gaudi::Plane3D entry = aSensor->entryPlane();
Gaudi::Plane3D exit = aSensor->exitPlane();
```

In addition, several methods are provided to convert between the measurement coordinate in the local frame and the strip number:

```
DeSTSensor* aSensor;
Gaudi::XYZPoint globalPoint;
Gaudi::XYZPoint point = aSensor->toLocal(globalPoint);
double x = point.x(); // What the strips measure
const unsigned int strip = aSensor->localUToStrip(x);

// or the other way
const double u = localU(strip,0.5); // offset between 0 and 1
```

These methods are used in the digitization procedure where the Monte Carlo truth information (entry and exits points from Geant4) is converted to the amount of charge deposited on each readout strip.

3.4 DeSTStation

Fig. 6 shows the class diagram for the **DeSTStation**. Again a base class expresses the commonality between the IT and TT. Typically, the user will not interact with these classes. Since they inherit from **DeSTBaseElement** they have all the common functionality that that class provides. For example it trivial to test whether a point is inside a station:

```
// ...
Gaudi::XYZPoint aPoint; // global point
```

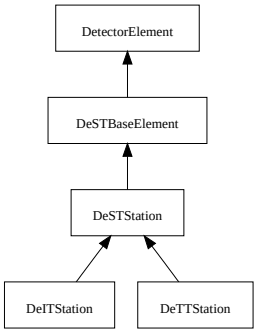

Figure 6: Class diagram **DeSTStation**.

```
DeSTStation* station;
if (station->isInside(aPoint) == true){
  // point is inside station
}
```

It is also possible to check if a **STChanneID** is inside the station:

```
LHCb::STChannelID chan;
bool isInside = station->contains(chan);

// internally use the fact that each detector element has STChannelID
LHCb::STChannelID chan = station->elementID();
```

The fraction of the station that is active can also be obtained:

```
bool activeFraction = station->fractionActive();
```

3.5 DeSTLayer

Fig. 7 shows the class diagram for the **DeSTLayer**. Again, a base class expresses the commonality between the IT and TT. During the track reconstruction the user may have to interact with this class. For example to test whether a point is inside a layer or to extrapolate to the plane corresponding to the centre of the layer. Examples of how to do this are given below:

```
// ...
Gaudi::XYZPoint aPoint;
DeSTLayer* layer;
if (layer->isInside(aPoint) == true){
  // point is inside layer
}
```

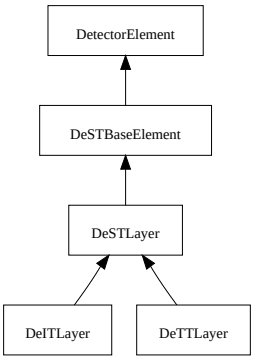

Figure 7: Class diagram **DeSTLayer**.

```
// Equation of plane at midpoint of layer mother volume
Gaudi::Plane3D thePlane = layer->plane();
```

3.6 DeITBox

This class is part of the IT detector element tree. It is needed by the alignment framework. The user should never interact with this class directly. Some examples of how this class is used are given below:

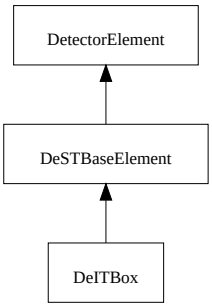

Figure 8: Class diagram **DeITBox**.

```
// ...

// get the top-level IT Detector element
DeITDetector* tracker =
   getDet<DeSTDetector>(DeSTDetLocation::location(''IT''));
Gaudi::XYZPoint aPoint;
DeITBox* aBox = tracker->findBox(aPoint);
```

```
// get the box nickname
std::string name = aBox->nickname();
```

3.7 DeITLadder

This class is part of the IT detector element tree. It is needed by the alignment framework. In general, the user should never interact with this class directly.

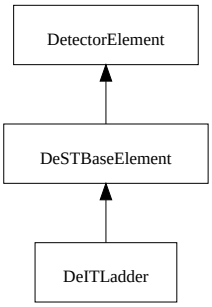

Figure 9: Class diagram **DeITLadder**.

3.8 DeTTHalfModule

This class is part of the TT detector element tree. It is needed by the alignment framework. In general. the user should not interact with this class directly. The code below illustrates some of the special functionality of this class:

```
// ...

DeTTHalfModule* halfModule;
std::string type = halfModule->type(); // KLM or LM
std::string location = halfModule->position(); // Top or Bottom
unsigned int col = halfModule->column();
```

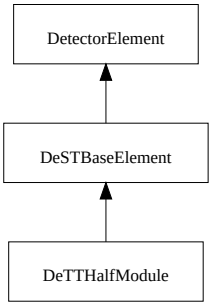

Figure 10: Class diagram **DeTTHalfModule**.

4 Debug Tools

For debugging all the detector classes are equipped with streamers that allow the properties of the object to be printed out at the command line:

```
DeITBox* aBox; // for example
std::cout << *aBox << std::endl;
```

In addition, the algorithm **STDumpGeom** in the STCheckers package allows to printout information on the entire IT or TT detector tree.

References

[1] Gauss v25r7, Boole v12r10, Brunel v30r14, XmlDDDB v30r14.

[2] M. Needham and Volyanskyy. Updated geometry description for the LHCb Trigger Tracker. LHCb-Note 2006-032.

[3] M. Needham. The Silicon Tracker Detector element tree. LHCb-Note 2006-034.

[4] E. Bos. The Trajectory Model for Track Fitting and Alignment. LHCb-note 2007-008.

[5] C. Dickens. *A Christmas Carol*. Bradbury and Evans Printers, Whitefriars, 1843.