JRA1 analysis framework

Raw data file

LCIO data file

DST data file

GUI
user input

Pedestal and
noise calculation

Data Reader

PixelMatrix
raw data

Pixel
over−threshold pixels

MatrixCalibration
pedestal and noise

Pedestal and noise
database

Histograms
DQM, analysis

Pedestal subtraction

Clustering

geometry database

Sensor parameters

Sensor

chip parameters

Configuration
initialization

Cluster
pixel clusters in
sensor coordinates

Position
reconstruction

DetectorPlane
sensor positioning
in the telescope

Telescope
geometry

sensor level

telescope level

TrackPoint
selected clusters in
telecscope ref. frame

Track finding
and fitting

field map

Field map

Trajectory
collection of (fitted)
track points

Alignment

Telescope and
DUT Analysis

RunInfo
date, time
beam energy

Data Writer

Result output

Output DST file

Output LCIO file

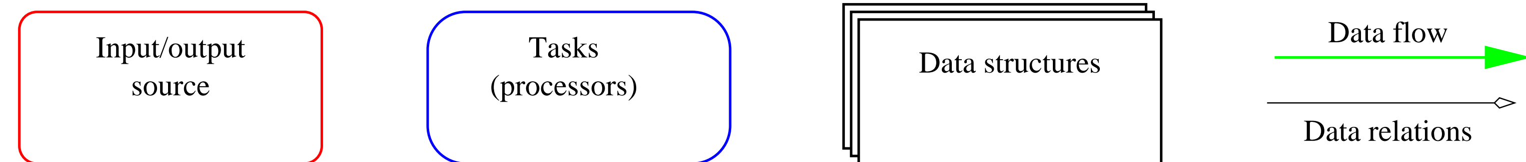
DQM histogram
file

Analysis histogram
and result file

GUI
graphic output

Event by event tasks

Run level tasks



EUTelescope

HEAD

Welcome to the EUTelescope documentation server. This is the place where you can find all the explanation and the examples you may need to run the EUTelescope processors within Marlin.

If you know already something about this software project, then you can browse the documentation clicking on the button above or putting a keyword in the search field in top right corner of this page.

If you feel you don't know enough on the EUTelescope then we encourage you take have a look at the following pages:

- **Introduction**
- **The preparation steps**
- **The analysis chain**
- **Download and installation**

If those information are not yet enough, consider contact us on the Linear Collider Forum.

Generated on Mon Jun 4 08:50:58 2007 for Eutelescope by doxygen 1.5.2

Eutelescope Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

eutelescope::EUDRBEventHeader	*This is the event file header*
eutelescope::EUDRBFileHeader	*This is the file header*
eutelescope::EUDRBTrailer	*This is the EUDRB trailer*
eutelescope::EUTelAutoPedestalNoiseProcessor	*Automatic pedestal and noise processor*
eutelescope::EUTelCalculateEtaProcessor	*Eta function calculator for the EUTelescope*
eutelescope::EUTelCalibrateEventProcessor	*Calibration processor for the EUTelescope*
eutelescope::EUTelClusteringProcessor	*Clustering processor for the EUTelescope*
eutelescope::EUTelClusterSeparationProcessor	*Cluster separation processor*
eutelescope::EUTelCopyPedestalProcessor	*Copy pedestal processor*
eutelescope::EUTELESCOPE	*Global constants used in the Eutelescope package*
eutelescope::EUTelEtaFunctionImpl	*Eta function LCIO implementation*
eutelescope::EUTelEUDRBReader	*Reads test data set written with the EUDRB board*
eutelescope::EUTelEventImpl	*Implementation of the LCEvent for the EUDET telescope*
eutelescope::EUTelFFClusterImpl	*Implementation of the a fixed frame cluster for the EUDET telescope*
eutelescope::EUTelHitMaker	*Hit maker processor*
eutelescope::EUTelOutputProcessor	*EUTelescope specific output processor*
eutelescope::EUTelPedestalNoiseProcessor	*Pedestal and noise processor for Marlin*
eutelescope::EUTelPseudo1DHistogram	*Simple class with a 1D histogram-like array w/o any display features*
eutelescope::EUTelRunHeaderImpl	*Implementation of the Run Header for the EUDET telescope*
eutelescope::EUTelSucimaImagerReader	*Reads SUCIMA Imager ASCII file*
eutelescope::EUTelTestFitter	*Analytical track fitting processor for EUDET Telescope*
eutelescope::EUTelUpdatePedestalNoiseProcessor	*Update pedestal and noise values*
eutelescope::EUTelVirtualCluster	*Virtual class to describe cluster in the EUTelescope framework*
eutelescope::IncompatibleDataSetException	*Incompatible data set*

The preparation steps

For the time being there are two preparation steps available within the EUTelescope package, for the calculation of calibration constants (**The pedestal and noise calculator**) and for eta functions (**Eta function calculation**).

The pedestal and noise calculator

The first step to be done is to calibrate the output of each pixel detector in order to remove the constant and useless signal. Together with this pedestal value also the noise figure is estimated as the width of the pedestal distribution.

The output of this step is a condition file having two TrackerData and one TrackerRawData for each detector module:

- Pedestal collection (TrackerData): one per detector with the mean output value of each pixel
- Noise collection (TrackerData): one per detector with the noise value for each pixel
- Status collection (TrackerRawData): one per detector with the status of each pixel. This can be used to hide some very bad pixels from the rest of the analysis.

Those collections are meant to be condition data and for the time being are saved to LCIO files but they can be moved to a database.
There are two ways to generate good pedestal and noise constants:

- Producing them from a specific run (**Calculating pedestal and noise**)
- Assuming a known initial value and then keep them update (**Assuming initial pedestal value**)

Calculating pedestal and noise

The standard approach for pedestal and noise calculation consists on having a special run with no or very few particles arriving on the telescope (a so called pedestal run, indeed) that should be processed in order to retrieve the mean value of the output signal of each pixel and the correponding noise figure. This task is accomplished by a specific processor named **eutelescope::EUTelPedestalNoiseProcessor**. More details about the calculation algorithms already implemented are provided within the class description.
Here below is an example of a steering file to produce the pedestal, noise and status collection using **eutelescope::EUTelPedestalNoiseProcessor**

```
<marlin>
 <global>
  <parameter name="LCIOInputFiles"> pedestal-run.slcio  </parameter>
  <parameter name="GearXMLFile" value="gear-telescope.xml"/>
  <parameter name="MaxRecordNumber" value="5001"/>
  <parameter name="SkipNEvents" value="0 "/>
  <parameter name="SupressCheck" value="false"/>
  <parameter name="Verbosity" value="MESSAGE"/>
 </global>

 <execute>
```

The analysis chain

In this page we would like to summarize which processors have to be used to go from raw data to tracks.

The histogram interface

The first processor one may want to add to the steering file is an histogram interface. This processor is making the handling of histograms very user friendly and available to all other processor. This is not a compulsory element of the analysis chain, but it is very encouraged.
Within the EUTelescope framework there are actually two available histogramming interfaces:

- the marlin::AIDAProcessor: provided by Marlin,
- the **eutelescope::ROOTProcessor**: that is instead typical of EUTelescope but still under development.

The user can decide which of one or even both should be used. Here below is a typical xml configuration the AIDAProcessor

```
 <processor name="AIDAHistogramInterface" type="AIDAProcessor">
  <!--
      Processor that handles AIDA files. Creates on directory per
      processor . Processors only need to create and fill the
      histograms, clouds and tuples. Needs to be the first
      ActiveProcessor
   -->
  <!-- compression of output file 0: false >0: true (default) -->
  <parameter name="Compress" type="int" value="1"/>
  <!-- filename without extension-->
  <parameter name="FileName" type="string" value="analysis-histo"/>
  <!-- type of output file xml (default) or root ( only OpenScientist)-->
  <parameter name="FileType" type="string" value="root"/>
</processor>
```

Loading the condition file

The second step in the analysis chain is to load the calibration file also known as pedestal file and the eta function files. Those can prepared beforehand (**The pedestal and noise calculator** and **Eta function calculation**) and the pedestal, noise and status collections should made available to the following processors using a ConditionProcessor. In fact those files can be real files or an entries of a condition database.
Here below an example of ConditionProcessor to be used to load collection from an LCIO file:

```
 <processor name="LoadPedestalFile" type="ConditionsProcessor">
  <!--ConditionsProcessor provides access to conditions data  transparently from LCIO files or a databases, using LCCD-->
  <parameter name="SimpleFileHandler" type="StringVec"> pedestalDB pedestal_db.slcio pedestalDB </parameter>
  <parameter name="SimpleFileHandler" type="StringVec"> noiseDB pedestal_db.slcio noiseDB </parameter>
  <parameter name="SimpleFileHandler" type="StringVec"> statusDB pedestal_db.slcio statusDB </parameter>
  <parameter name="SimpleFileHandler" type="StringVec"> xEtaCondition etafile_db.slcio xEtaCondition </parameter>
  <parameter name="SimpleFileHandler" type="StringVec"> yEtaCondition etafile_db.slcio yEtaCondition </parameter>
```

eutelescope::EUTelHitMaker Class Reference

Hit maker processor. More...

`#include <EUTelHitMaker.h>`

List of all members.

Public Member Functions

virtual Processor *	**newProcessor** () *Returns a new instance of* ***EUTelHitMaker****.*
	EUTelHitMaker () *Default constructor.*
virtual void	**init** () *Called at the job beginning.*
virtual void	**processRunHeader** (LCRunHeader *run) *Called for every run.*
virtual void	**processEvent** (LCEvent *evt) *Called every event.*
virtual void	**end** () *Called after data processing.*
void	**bookHistos** () *Histogram booking.*

Protected Attributes

std::string	**_pulseCollectionName** *TrackerPulse collection name.*
std::string	**_hitCollectionName** *TrackerHit collection name.*
std::vector< std::string >	**_etaCollectionNames** *Eta function collection names.*
int	**_etaCorrection** *Switch to apply eta correction.*

Detailed Description

Hit maker processor.

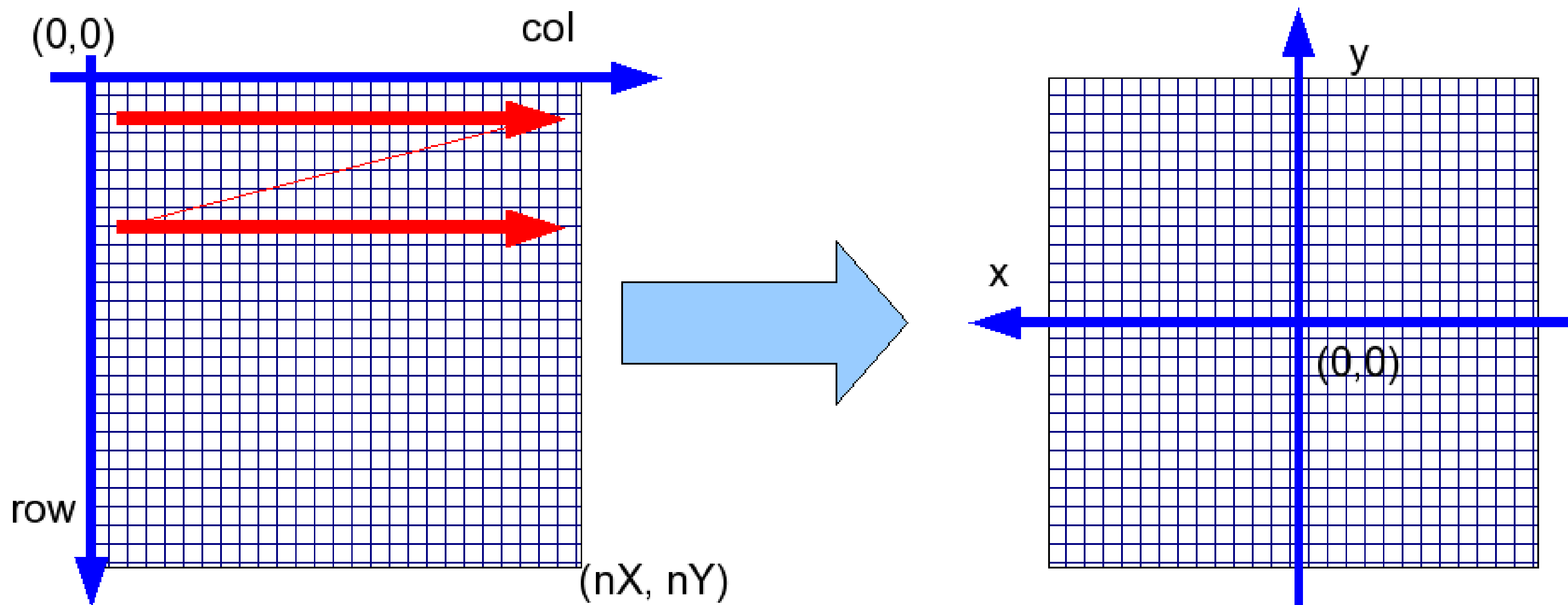
Beyond this cryptic name there is a simple as important processor. This is the place were clusters found in the clustering processor and saved into TrackerPulse objects become hits having a certain position in the outern space, a covariance matrix and all the other attributes needed to use this point into a track fitting algorithm.

This is the place where the cluster center in the local detector frame of reference is translated into a geometrical position according to the telescope description provided in the GEAR environment. For this reason, a cross check on the availability of a GEAR instance and of a GEAR geometry description XML file is done during the init phase.

The center of the cluster can be calculate both using a standard linear procedure, or using the Eta function. In this case the user must activate the "Apply Eta correction" function and provide the name of a suitable collection containing Eta information for each of the detector.

Frame of reference conversion

Data coming from the DAQ and from the previous analysis steps are organized into Tracker(Raw)Data objects with pixel signals saved into a 1D array in a row by row arrangement. Which is the first pixel to be readout depends on the hardware. For the MimoTel, the first read pixel is the one occupying the top left most position as shown in the picture here below. This means that the last pixel will be the one on the bottom right corner. Generally speaking this local frame of reference has to be converted in the telescope frame of reference that is right-handed with the z axis parallel to the beam directions, the y axis going from the bottom to the top side and the x axis from the right to the left hand side.



Frame of references: the

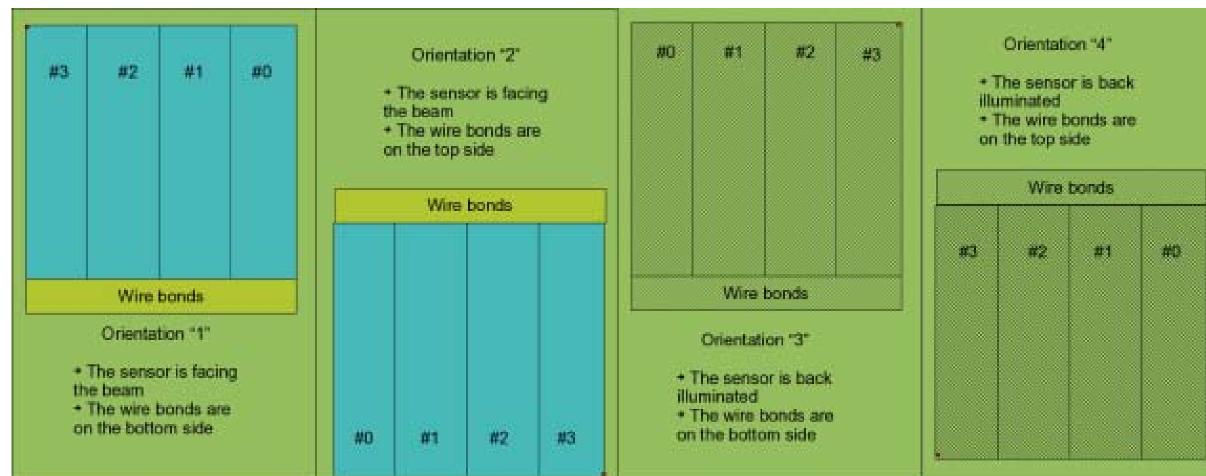
detector on the left and the telescope one on the right."

Moreover, the detector FoR can be rotated with respect to the telescope one. This rotation in the x-y plane is expressed by the two 2D vectors xPointing[2], yPointing[2].

Some examples:

- The two FoRs are perfectly aligned: xPointing = { 1 , 0 } yPointing = { 0 , 1 }
- The detector is oriented as in the figure above with the x axis horizontal going left to right and the y axis vertical going top to bottom. xPointing = { -1 , 0 } yPointing = { 0 , -1 }

Among all possible rotations, four are particularly important because of the way the sensor support can be inserted into the telescope mechanics. Those are described into the figure here below:



Four important orientations

Orientation 1 represents the case in which the sensor is inserted up right in the telescope setup and the beam is hitting the front surface of the sensor. This configuration corresponds to:
xPointing = { -1 , 0 } yPointing = { 0 , -1 }

Orientation 2 represents the case the sensor is inserted up side down facing the beam. This is coded as:
xPointing = { 1 , 0 } yPointing = { 0 , 1 }

Orientation 3 represents the case of the sensor in up right position but back illuminated. This is coded as:
xPointing = { 1 , 0 } yPointing = { 0 , -1 }

Orientation 4 represents the sensor up side down and back illuminated. This is obtained with:
xPointing = { -1 , 0 } yPointing = { 0 , 1 }

Other possible rotations are not currently available in the geometry description.

Control histograms

eutelescope::EUTelTestFitter Class Reference

Analytical track fitting processor for EUDET Telescope. More...

`#include <EUTelTestFitter.h>`

List of all members.

Public Member Functions

virtual Processor *	**newProcessor** () *Returns a new instance of EUTelTestFitter.*
	EUTelTestFitter () *Default constructor.*
virtual void	**init** () *Called at the job beginning.*
virtual void	**processRunHeader** (LCRunHeader *run) *Called for every run.*
virtual void	**processEvent** (LCEvent *evt) *Called every event.*
virtual void	**check** (LCEvent *evt) *Check event method.*
virtual void	**end** () *Called after data processing for clean up.*

Protected Member Functions

double	**MatrixFit** () *Find track in XZ and YZ.*
double	**SingleFit** () *Find track in XZ and YZ assuming nominal errors.*
double	**NominalFit** () *Find track in all planes assuming nominal errors.*
int	**DoAnalFit** (double *pos, double *err) *Fit particle track in one plane (XZ or YZ).*

double *	**_fitEy**
double *	**_fitArray**
double *	**_nominalFitArray**
double *	**_nominalError**

Detailed Description

Analytical track fitting processor for EUDET Telescope.

This processor was designed for fitting tracks to hits reconstructed in the telescope sensor planes. Analytical approach is used, taking into account multiple scattering in the telescope planes.

Method

Track fitting is performed separately in XZ and YZ planes (Z is defined along the beam axis direction). Track position in each telescope plane is found by solving matrix equation resulting from Chi^2 minimum condition. The following approximation is used:

- all telescope planes are parallel to each other
- the incoming beam is perpendicular to the telescope planes
- the incoming beam has a small angular spread
- particle scattering angles in subsequent telescope layers are also small
- thicknesses of all material layers are very small compared to the distances between planes
- particle energy losses in telescope layers can be neglected

Algorithm

- Read measured track points from input TrackerHit collection and copy to local tables
- Prepare lists of hits for each active sensor plane
- Count hit numbers, return if not enough planes fired
- Calculate number of fit hypothesis (including missing hit possibility)
- Fit each hypotheses and calculate Chi^2 (including ``penalties'' for missing hits or skipped planes)
- Select the Chi^2
- Write fitted track position at DUT to output TrackerHit collection
- Remove best track hits from hit list and repeat procedure

Geometry description

This version of the processor does not use GEAR input yet! Needed geometry information are stored in a dedicated ASCII file. The format of the file is as follows.

First line consists of two int numbers:

- number of telescope layers N (sensors and passive layers taken into account in the fit) and
- position of DUT i_DUT (i_DUT = 1 ... N). i_DUT=0 corresponds to telescope without DUT.

Following N lines include description of telescope planes. Planes have to be ordered in position along the beam line ! For each plane following details have to be given:

- position of the plane along beam axis in mm (float)
- thickness of the plane in mm (float)
- radiation length in the plane material in mm (float)
- flag indicating sensitive planes (int; 1 for sensitive, 0 for passive plane)
- nominal position resolution of sensitive planes in mm (float)

Output

Fitted particle positions in all telescope planes are stored as `TrackerHit` collection. In addition fit results are written in a `Track` collection. Following `Track` variables are filled:

- Chi2 of the fit
- number of measured hits used in the track fit (as Ndf)
- reconstructed position at DUT (as a track reference point)
- vector of hits (fitted particle positions in all planes)

Parameters:

InputCollectionName	Name of the input TrackerHit collection
OutputHitCollectionName	Name of the output collection of fitted particle positions in telescope planes (hits)
OutputTrackCollectionName	Name of the output Track collection
DebugEventCount	Print out debug and information messages only for one out of given number of events. If zero, no debug information is printed.
GeometryFileName	Name of the geometry description file. This version of the processor does not use GEAR input yet! Needed geometry information are stored in a dedicated ASCII file.
AllowMissingHits	Allowed number of hits missing in the track (sensor planes without hits or with hits removed from given track)
AllowSkipHits	Allowed number of hits removed from the track (because of large Chi^2 contribution)
MaxPlaneHits	Maximum number of hits considered per plane. The algorithm becomes very slow if this number is too large.
MissingHitPenalty	"Penalty" added to track Chi^2 for each missing hit (no hits in given layer).
SkipHitPenalty	"Penalty" added to track Chi^2 for each hit removed from the track because of large Chi^2 contribution.
Chi2Max	Maximum Chi2 for accepted track fit.
UseNominalResolutio	Flag for using nominal sensor resolution (as given in geometry description) instead of hit position errors.
UseDUT	Flag for including DUT measurement in the track fit.
Ebeam	Beam energy in [GeV], needed to estimate multiple scattering.
UseBeamConstraint	Flag for using beam direction constraint in the fit. Can improve the fit, if beam angular spread is small.
BeamSpread	Assumed angular spread of the beam [rad]
SearchMultipleTracks	Flag for searching multiple tracks in events with multiple hits

Warning:

Algorithm is very fast when there are only single (or double) hits in each sensor layer. With increasing number of hits per layer (HpL) fitting time increases like HpL^N, where N is number of sensors. To protect from "freezing" `MaxPlaneHits` parameter is introduced (default value is 5 and should rather not be increased).

Todo:

- Interface to GEAR geometry description
- More detailed track parameter output (currently only reconstructed hit positions are stored)
- Implement new track selection algorithm, which would avoid testing all track hypothesis (to improve efficiency for events with many hits)

Author:

A.F.Zarnecki, University of Warsaw

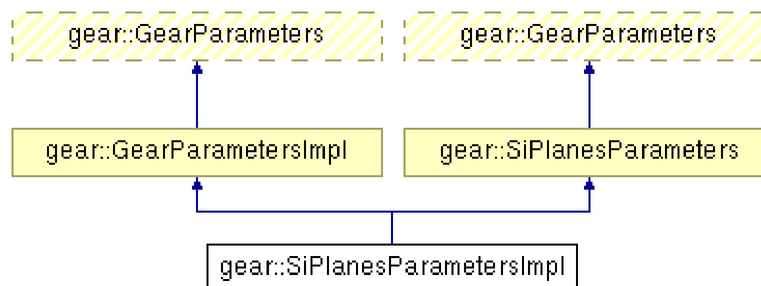
Version:

gear::SiPlanesParametersImpl Class Reference

Abstract description of layers in beam telescope with or without DUT. More...

`#include <SiPlanesParametersImpl.h>`

Inheritance diagram for gear::SiPlanesParametersImpl:



List of all members.

Public Member Functions

	SiPlanesParametersImpl (int siplanesType, int siplanesNumber) *C'tor.*
virtual void	**addLayer** (int layerID, double layerPositionX, double layerPositionY, double layerPositionZ, double layerSizeX, double layerSizeY, double layerThickness, double layerRadLength, double sensitivePositionX, double sensitivePositionY, double sensitivePositionZ, double sensitiveSizeX, double sensitiveSizeY, double sensitiveThickness, double sensitiveRadLength) *adding a Layer to the Si planes*
virtual void	**addDUT** (int dutID, double dutPositionX, double dutPositionY, double dutPositionZ, double dutSizeX, double dutSizeY, double dutThickness, double dutRadLength, double dutsensitivePositionX, double dutsensitivePositionY, double dutsensitivePositionZ, double dutsensitiveSizeX, double dutsensitiveSizeY, double dutsensitiveThickness, double dutsensitiveRadLength)
virtual const **SiPlanesLayerLayout** &	**getSiPlanesLayerLayout** () const *Returns the layer layout in the SiPlanes detector.*
virtual int	**getSiPlanesType** () const *The type of SiPlanes detector: SiPlanesParametersImpl.TelescopeWithDUT or SiPlanesParametersImpl.TelescopeWithoutDUT.*
virtual int	**getSiPlanesNumber** () const

Number of Si planes.

Static Public Attributes

const int	**TelescopeWithDUT** = 1
const int	**TelescopeWithoutDUT** = 2

Protected Attributes

SiPlanesLayerLayoutImpl	**_layer**
int	**_siplanesType**
int	**_siplanesNumber**

Detailed Description

Abstract description of layers in beam telescope with or without DUT.

This assumes a number of silicon layers, arranged perpendicular to the beam

Author:
T Klimkovich, DESY

Version:
$Id:

Constructor & Destructor Documentation

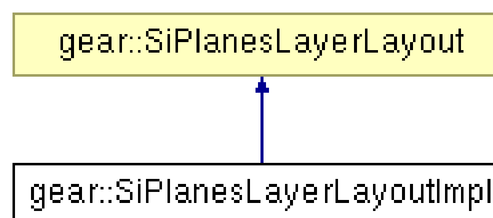
gear::SiPlanesParametersImpl::SiPlanesParametersImpl (**int** *siplanesType*,
int *siplanesNumber*
)

gear::SiPlanesLayerLayoutImpl Class Reference

Abstract description of layers in beam telescope or FTD detector. More...

`#include <SiPlanesLayerLayoutImpl.h>`

Inheritance diagram for gear::SiPlanesLayerLayoutImpl:



List of all members.

Public Types

typedef std::vector< **Layer** >	**LayerVec**
typedef **Layer**	**DUT**

Public Member Functions

virtual int	**getNLayers** () const *The total number of layers.*
virtual int	**getID** (int layerIndex) const *The ID of the layer.*

virtual double **getLayerRadLength** (int layerIndex) const
The radiation length in the support structure of layer layerIndex - layer indexing starts at 0 for the layer closest to the beam source.

virtual double **getLayerPositionX** (int layerIndex) const
The position of layer layerIndex.

virtual double **getLayerPositionY** (int layerIndex) const
virtual double **getLayerPositionZ** (int layerIndex) const
virtual double **getLayerSizeX** (int layerIndex) const
The size in mm of layer layerIndex.

virtual double **getLayerSizeY** (int layerIndex) const
virtual double **getLayerThickness** (int layerIndex) const
virtual double **getSensitiveRadLength** (int layerIndex) const
The radiation length in sensitive volumes in layer layerIndex - layer indexing starts at 0 for the layer closest to the beam source.

virtual double **getSensitivePositionX** (int layerIndex) const
The position of sensitive area in layer layerIndex.

virtual double **getSensitivePositionY** (int layerIndex) const
virtual double **getSensitivePositionZ** (int layerIndex) const
virtual double **getSensitiveSizeX** (int layerIndex) const
The size in mm of the sensitive area in layer layerIndex.

virtual double **getSensitiveSizeY** (int layerIndex) const
virtual double **getSensitiveThickness** (int layerIndex) const
virtual void **addLayer** (int layerID, double layerPositionX, double layerPositionY, double layerPositionZ, double layerSizeX, double layerSizeY, double layerThickness, double layerRadLength, double sensitivePositionX, double sensitivePositionY, double sensitivePositionZ, double sensitiveSizeX, double sensitiveSizeY, double sensitiveThickness, double sensitiveRadLength)
Add a new layer at the given positon.

virtual int **getDUTID** () const

	The ID of the DUT.
virtual double	**getDUTRadLength** () const *The radiation length in the support structure of the DUT.*
virtual double	**getDUTPositionX** () const *The position of DUT.*
virtual double	**getDUTPositionY** () const
virtual double	**getDUTPositionZ** () const
virtual double	**getDUTSizeX** () const *The size in mm of DUT.*
virtual double	**getDUTSizeY** () const
virtual double	**getDUTThickness** () const
virtual double	**getDUTSensitiveRadLength** () const *The radiation length in the sensitive volume of the DUT.*
virtual double	**getDUTSensitivePositionX** () const *The position of sensitive area of DUT.*
virtual double	**getDUTSensitivePositionY** () const
virtual double	**getDUTSensitivePositionZ** () const
virtual double	**getDUTSensitiveSizeX** () const *The size in mm of the sensitive area of DUT.*
virtual double	**getDUTSensitiveSizeY** () const
virtual double	**getDUTSensitiveThickness** () const
virtual void	**addDUT** (int dutID, double dutPositionX, double dutPositionY, double dutPositionZ, double dutSizeX, double dutSizeY, double dutThickness, double dutRadLength, double dutsensitivePositionX, double dutsensitivePositionY, double dutsensitivePositionZ, double dutsensitiveSizeX, double dutsensitiveSizeY, double dutsensitiveThickness, double dutsensitiveRadLength)