Object oriented program examples for high energy physics reconstruction

Available on CMS information server

CMS NOTE 1998/017

The Compact Muon Solenoid Experiment

CMS Note

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

25 February 1998

Objected Oriented program examples for High Energy Physics reconstruction

P.D.V. van der Stok

Eindhoven University of Technology, The Netherlands

Abstract

A number of Object Oriented example programs based on High Energy Physics event reconstruction are worked out.

1 Introduction

This report presents a set of Object-Oriented (OO) programs that should help the reader to design OO programs for High Energy Physics reconstruction. The writing of these examples seems appropriate at a moment that a large group of people is leaving the FORTRAN language and embrace the OO concepts [Boo94]. Talking with people shows that the step from FORTRAN to C++ [Str97] is easily made. Positive comments on C++ usually concern the pointer handling in C++. Most people that start with OO design are looking for a guide for choosing the appropriate classes. The creation of a class specification that later hides many implementation details is not easily understood. It is difficult to give reasons why one design is better than another (motivation of the design).

Apart from the performance and functional aspects of the software product other aspects can be important as well. OO languages and techniques are generally recognised to be a better vehicle to guarantee several non-functional aspects (e.g. adaptability, modularity).

OO languages facilitate the modelling of real-world objects. This makes it easier to verify that specifications meet the requirements. The step from loose requirements to specification is smaller. However, the accompanying design needs more thought to meet the more detailed and complex specification.

OO languages help to hide tedious implementation details. The resulting software can therefore, with less side effects, be modified at a few well defined places. Extension of the software is more easily possible because the language helps in creating objects with small interfaces thus limiting the number of places where code must be adapted or extended.

A method needs to be found to bridge the gap between the C++ programmer and the OO designer. The first knows how to make a working C++ program, the second knows how to create a set of classes and objects that meet several non functional constraints (such as: different implementations are supported, classes can be reused in other programs). The courses by Kunz are well appreciated by most and give a manual for the language. They provide a limited insight in the design process. The CERN recommended books on software design comprise the most up to date and yet accepted reading material. However, the existing literature does not seem to be convincing to the physicist. A reason can be that the cited examples and constraints are not felt to be significant to the physicist's problems. It was therefore thought advisable to create a set of examples that are recognised as relevant examples. Several implementations should be provided for these examples. Disadvantages and advantages of the different designs should be pointed out. The elaboration of examples that refer to the patterns in the book Design Patterns [GHJV94] seemed reasonable. The use of the STL library [MS96] was recommended.

The method adopted here is to start with a simple example, provide a simple solution and then complicate the problem or extend the boundary conditions. Examples were found by listening to presentations by persons from different projects within CMS.

The following examples are worked out in this report:

1. **SiBT [ea96] calibration** Good opportunities for layering and abstraction are introduced by looking at aspects of the detector that need to be hidden to the different software components.
2. **Detector layers** A complex set of complex objects with a structure that should support efficient track recognition can be proposed. The opportunity for using STL is exploited.
3. **Local pattern recognition (in progress)** An application of the open-closed principle as elaborated for CMS reconstruction.
4. **Reconstruction algorithm versions (not done)** The Ecole Polytechnique people introduced the concept of sheets to cater for different versions of the algorithms. The introduction of a pattern from [GHJV94] seems advisable.

The readers are assumed to be familiar with the Object Methodology Tool (OMT) notation used for the Rational/Rose tool. Explanations of the meaning of class diagrams and scenario diagrams are not provided. The reader is assumed to have read the Booch book on the subject.

The software described in the examples is available. Exercises are specified to assist the reader in trying different design issues. The best way to learn handling a new technique is by applying the rules and see what they do for you.

The number of comments in the code have been kept low. Quite a lot of the code is described in the text and the comments in the code only help to focus the attention of the reader. In principle every method should have a header that describes the purpose of the method and its pre- and post- conditions. This is not done for the same reasons. Entry assumptions are usually described in the text and the end results of the method as well.

The main purpose of these examples is to show possible class structures. No emphasis is placed on finding fast track fitting algorithms or complex detector structures. Therefore, the examples only use simple straight line fits and detectors with a simple geometry.

Layering is a well known and important technique to structure the software. The software is divided in layers. Each layer uses the facilities of the layer immediately below. A layer does not see the implementation of this layer or the facilities of the layers that are lower. Each layer delivers higher level abstractions than the layer below. Layering is one of the issues in the presented examples.

2 SiBT example

The first example is based on the work associated with the Silicon Beam Telescope (SiBT) that is developed by the Helsinki Institute of Physics. At the time of writing, SiBT is installed at CMS/H2 testbeam at CERN. It was chosen because at that time the conversion from Geant3 to Geant4 was undertaken and the design issues related with the associated SiBTOO software are also met in the context of CMS reconstruction software.

2.1 SiBT description

The telescope is built on an optical precision bench. The telescope consists of 8 assemblies of silicon detector strips called planes. The planes are placed in pairs such that one plane of a pair is aligned horizontally and the other plane is aligned vertically. The plane pairs are evenly spaced and the distance between the first and the last plane is approximately 50 cm.

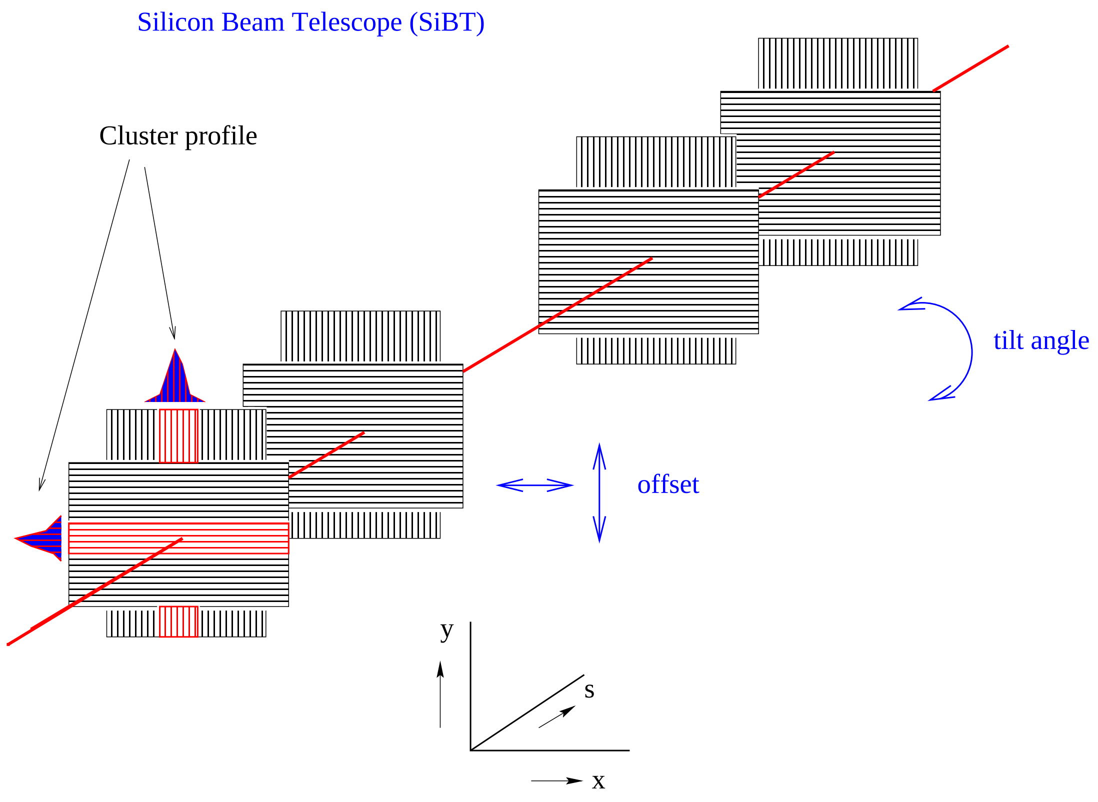

Figure 1: Diagram of SiBT layout (courtesy of Aatos Heikkinen)

Fig. 1 (not on scale) explains the basic SiBT properties. A particle passing SiBT from front to end excites the silicon strips through which it passes, thus provoking an electronic signal from the crossed strip and some of its neighbours depending on the particle energy and momentum. In Fig. 1 the first plane is vertically aligned and the second plane is horizontally aligned. The signals recorded in the vertically (horizontally) aligned planes with horizontal (vertical) strips inform about the y- (x-) coordinate of the intersection of the particle trajectory with the plane.

A strip provides two digital numbers: a noise and a signal number. The noise relates to the signal level when no particles pass. The signal is a measure for the excitation of the strip by a particle. The decision whether a particle has excited a strip is based on the signal/noise ratio that should be larger than a plane dependent constant. The criterion for the passage of a particle is given by: at least two neighbouring strips have the required signal/noise ratio. The track of a particle is determined by the measured excitations of the silicon strips. A sequence of three to four horizontal measurements is called a horizontal projection and a sequence of three to four vertical measurements is called a vertical projection.

The planes are carefully aligned. However, small misalignments may subsist. These are measured during a calibration session. A set of measurements of well defined particle tracks is used. Through each set of track measurements a straight line is fitted. For each plane the difference between the measured interaction coordinate and the point where the fitted line crosses the plane is determined. Statistics on the differences are done to establish the realignment coordinates of the individual planes.

2.2 Analysis

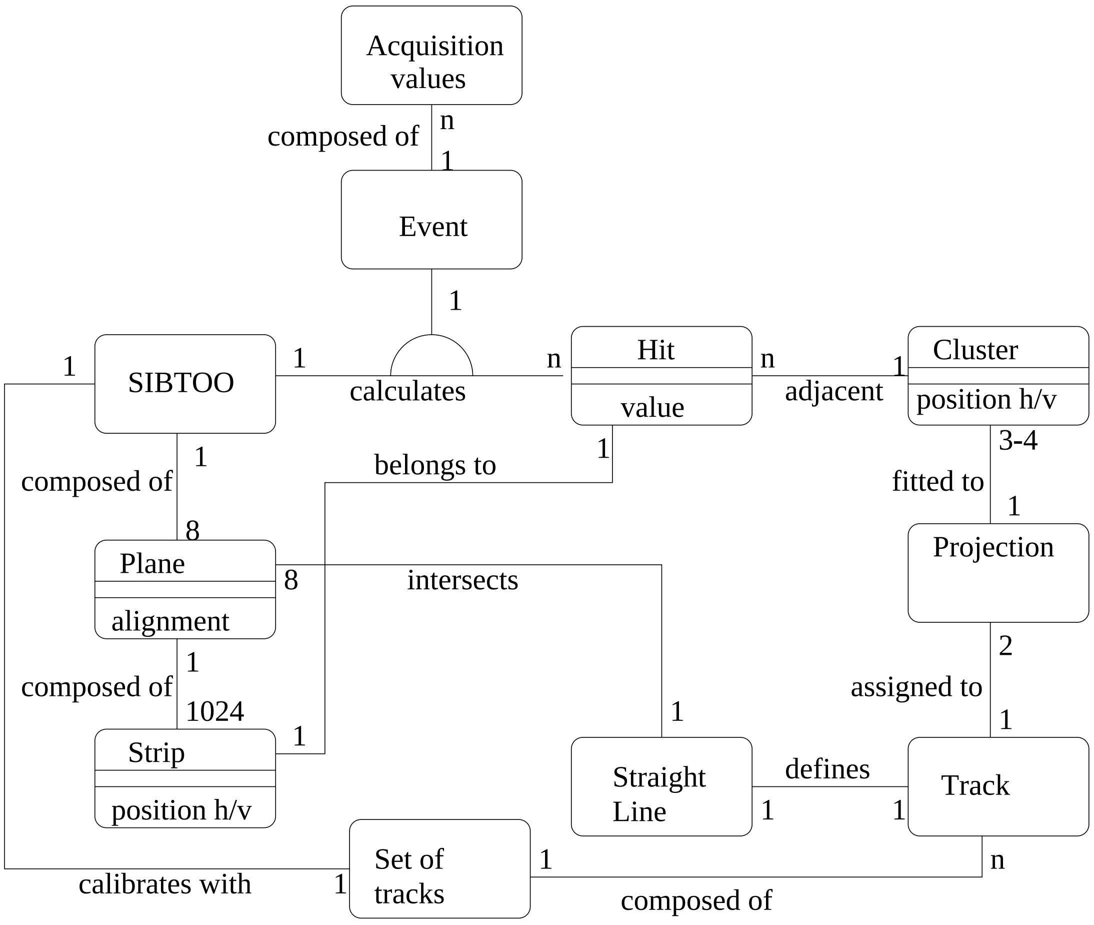

Figure 2: Class diagram based on SiBT analysis

SiBTOO is the model of the SiBT detector written in an object oriented language (C++). SiBTOO has a certain configuration. A number of planes (8) at given positions are composed of a number of strips (1024). A plane has a horizontal or vertical orientation determined by the orientation of the strips. Above, a class diagram based on the analysis of the problem domain is presented. The analysis is based on the physical lay-out of SiBT and the following two use-cases: Find tracks and Calibrate SiBT.

1. **Find tracks** An event consists of a set of acquisition values generated by SiBTOO or read from disk (off-line case, simulation) or recorded by SiBT (on-line case). All acquisition values are associated with strips that can be activated by the passage of one or more particles. Activated strips are recognised by the acquisition values. A hit is represented by a strip of which the value satisfies a number of criteria. Neighbouring hits of the same plane are combined to a Cluster. Clusters from planes with the same orientation are combined to a candidate projection. A projection is a candidate projection of three to four clusters through which a straight line can be fitted. A track is the combination of two projections, one from each orientation. Per event one or zero tracks are returned. Difficult to solve ambiguities arise when more than one track is created. For simplicity reasons this is not addressed.

2. **Calibrate SiBT** A set of tracks is used to determine the best alignment constants for the eight planes of SiBTOO. A number of times the following calibration procedure is done. A straight line is fitted through all tracks of the set. For every track from the set for every plane the difference between the cluster coordinate and the straight line coordinate is calculated. The mean and the deviation of the offsets of a given plane over all tracks are calculated. From these values an improved set of alignment constants is calculated.

The above two use-cases and the structure of SiBTOO lead to the identification of a certain number of classes and their associations shown in Fig. 2.

While defining the classes, one should keep in mind that some boundary conditions exist: the configuration of SiBTOO can change, the calculation of projections can change, the track finding and line fitting can change. The calibration can change. Instead of at most one track, several tracks can be returned. The off-line treatment of SiBTOO should resemble the on-line SiBTOO as much as possible.

2.3 Design

The scenario diagrams are used to show where the responsibilities are put for the different classes identified above. The objective is to hide as much as possible unnecessary details concerning a given class from the other classes. *Main* is the actor for the use-cases mentioned above. A first design decision is to hide the structure of SiBTOO from *Main* that provides the acquisition values received from the equipment. The SiBTOO class is made responsible for the calculation and assignment of values to the strips as defined by the acquisition values stored in Event. In this way modifications in the SiBTOO hardware are hidden to *Main*.

Track fitting To have close resemblance with the on-line case, SiBTOO is modelled to receive one event at the time and find the tracks corresponding with the event. In the off-line case a set of events can be calculated or fetched from a storage medium after which SiBTOO is invoked once for every event from the set.

Because planes differ in orientation, offsets and possibly the number and quality of strips can differ, SiBTOO distributes the unmodified acquisition values of the events over the planes. Acquisition values contain encoded results of SiBTOO. Per strip, the noise and signal values are stored. At some time in the future the event contents can be changed. For example, instead of returning the acquisition values of all strips, only activated strips are returned by the hardware and the acquisition values must contain the strip and plane identifier. The planes calculate the individual strip values and assign them to the strips. Differences between planes are localised to the plane and not visible to SiBTOO as a whole.

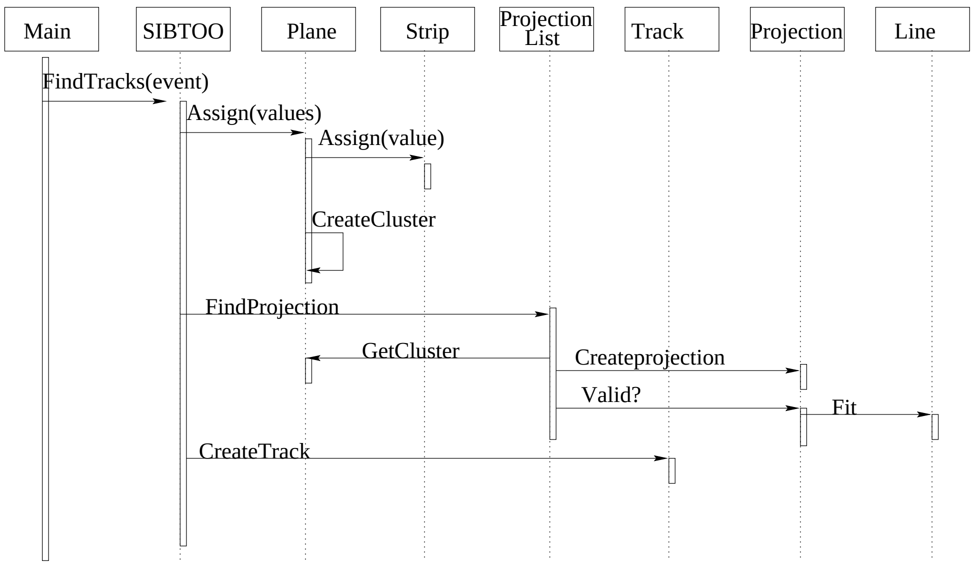

Figure 3: Scenario for track fitting

A scenario for track fitting is shown in Fig. 3. This represents one of a set of possible scenarios. This one is chosen as a first possibility based on some design criteria explained below. Design decisions concern the choice of classes that contain cluster recognition code. Clusters could be recognised by the plane or by SiBTOO. In favour of SiBTOO is: clusters from different planes need to be visible to the whole of SiBTOO to later create projections. In favour of the plane is: each plane has a specific orientation and delivers either a horizontally or vertically projected cluster; consequently, there are two types of clusters as there are two types of planes (horizontally and vertically aligned). We choose the plane for the location of the cluster calculation code. This implies that the Cluster class must be defined such that they contain sufficient information to construct a track. In this design, alignment and conversion factors are hidden in the planes responsible for creating the clusters. The location of identified clusters (in the plane or in SiBTOO) cannot be determined yet.

Clusters are calculated from the plane- and strip- characteristics and represent a point in space. The cluster contents is based on the needs for projection and track recognition in a Cartesian coordinate system independent of SiBTOO geometry. The appropriate geometry of a track, line, circle or helix can be chosen independent of the SiBTOO properties. Information about the size of the cluster and the intensity of the signal must be foreseen for later extensions when more than one track may pass through a plane and a cluster can represent the passage of one or more particles.

The SiBTOO object takes all clusters from either the horizontal or the vertical planes and creates horizontal and vertical projections. All possible combinations of clusters from four or three different planes with the same orientation are tried. Two possibilities are immediately obvious: (1) the projection finds the appropriate (vertical or horizontal) planes and selects a given cluster from each plane and (2) SiBTOO selects all posible clusters combinations from the planes and creates a projection with as parameter the selected cluster set. During the development of the program a third possibility was implemented. A projection-list is created with as parameter the set of horizontal or vertical planes. The projection-list creates as many projections as there are combinations of three or four clusters belonging to different planes with the same orientation. This choice allows to encapsulate the cluster selection for projection finding into one class and makes the cluster independent of the plane layout or plane characteristics.

The *Valid* method of the projection returns True if it can fit a line through the clusters.

To create a projection from the clusters stored inside the planes, the following design is proposed. The location of the cluster can be fixed inside the planes. The clusters in the plane are ordered in a list. By asking a plane for the current or next cluster, all possible combinations of clusters, one from each plane can be stored into projections (lists of clusters). SiBTOO creates a projection-list. The projection list creates projections to which clusters from different planes are added. Each projection tests whether the proposed cluster list is a valid one by fitting a line through them.

All valid projections are combined to all possible tracks by selecting all possible pairs of horizontal and vertical projections. One single track is returned to *Main* if one vertical and one horizontal projection is found, otherwise a set of zero tracks is returned to *Main*. Extensions for the return of more tracks will be considered for the design but are not actually implemented.

The track finding algorithm is hidden from *Main*. Cluster determination is hidden from SiBTOO thus catering for plane dependent cluster calculations.

The current design has consequences for the initialisation. When SiBTOO is activated with new acquisition values and new strip values, references to formerly calculated clusters, projections and tracks must be removed.

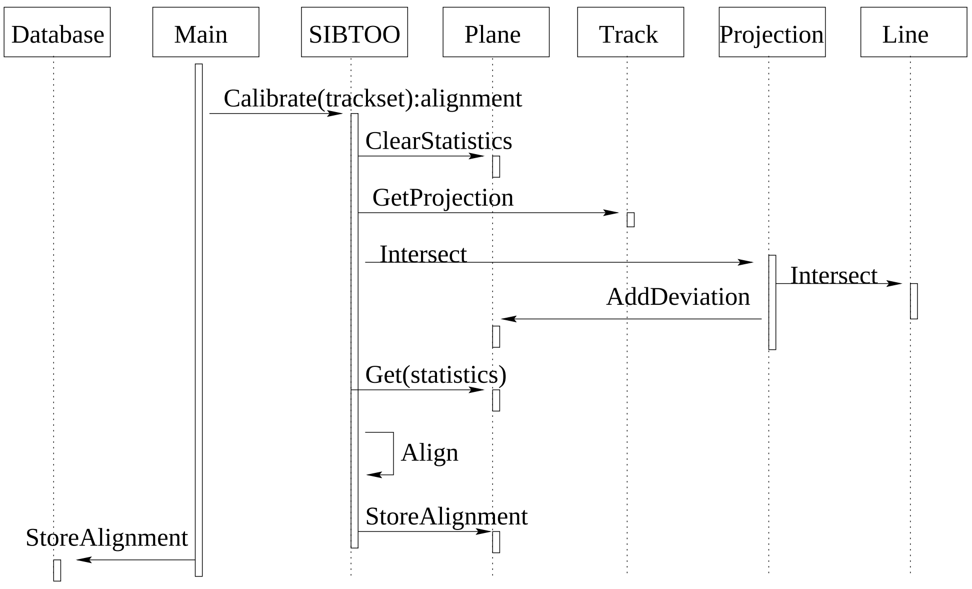

Figure 4: Scenario for calibration

Calibration A scenario for calibration is shown in Fig. 4. Main asks SiBTOO to calibrate itself with a set of tracks. The decision is taken to store statistics on differences between fitted lines and the measured clusters within the associated plane. This enables the use of plane dependent weight factors or to execute plane dependent calculations. Before calibration starts, statistics calculated during a former calibration round must be removed. For each track in the set of tracks, each projection is asked to calculate the difference between the fitted line and the clusters of which the projection is composed. The plane that corresponds with a cluster is found and the deviation is added to the plane statistics.

Once all tracks are done, SiBTOO acquires the statistic results, calculates alignment improvements and communicates them to the planes. After the calibration, *Main* stores the returned calibration constants in a database.

Another way (not shown here) to perform calibration is to split the invocation of SiBTOO in three parts. In the first part SiBTOO is initialised for calibration and all former statistics are removed. In the second part, *Main* invokes SiBTOO with *Calibrate* for every single track. In the third part, *Main* must invoke SiBTOO to determine the deviations. Based on their values, *Main* can restart the calibration for the set of tracks or store the values in the database. There are no good arguments on the basis of the available knowledge to determine whether one of the two designs is superior to the other.

2.3.1 Tentative class definitions

The above analysis and responsibility allocation permits us to do a more detailed design of the classes. In fig. 2 classes and associations are shown but no suggestion about the realization of these associations is done. From the scenarios above the association between some classes can be designed with the appropriate language facilities. During the realization of the classes, improvements and modifications are needed. The design in this section represents a first effort not hindered by implementation details. The classes of the final design presented in section 2.4 differ slightly from the here presented classes.

Within SiBTOO an array of 4 horizontal and 4 vertical Plane objects can be defined. Within a Plane an array of 1024 Strip objects can be defined. Planes are not visible outside SiBTOO and Strips are not visible outside a Plane object. Arrays are chosen because the number of planes and strips is relatively fixed and changes only after elaborate hardware modifications. The cluster is a number of contiguous strips that have a value higher than a given threshold. Although shown in the analysis (see Fig. 2), no need for the introduction of a Hit class exists. A Cluster class contains a variable number (> 1) of strips. The actual strips are not important to the cluster. The position of the strips associated with a cluster contains enough information for the track calculation.

In Fig. 5 the classes with their methods and attributes as needed so far are shown.

SiBTOO The SiBTOO class has an interface that is the interface between all classes defined within SiBTOO and the SiBTOO invoking classes (in this case Main). Three public methods are defined *Create*, *FindTracks* and *Calibrate*. Apart from *Create*, the interface is completely determined by the two scenarios. *Create* initialises SiBTOO and determines all hardware determined numbers within SiBTOO. Private attributes are the set of horizontal and vertical planes and the set of horizontal and vertical projections. The set of reconstructed tracks is returned as a parameter by *FindTracks*.

Plane The Plane is clearly the most complex class. It contains information about its position (Coordinates), the deviations calculated during the calibration efforts (Statistics), conversion factors (Alignments), equipment (StripArray) and measurement results (ClusterSet). All these attributes are hidden.

A set of public methods is delivered to manipulate the Plane contents. The *Create* routine sets all hardware and position parameters that do not change. The alignments are set to zero and are later determined by doing a calibration. The orientation of the plane is determined at the creation of SiBTOO by assigning a plane to either the set of horizontal or the set of vertical planes. The *Assign* method is used to assign a set of values to the Strips based on the measurements contained in the ChannelSet parameter. As shown in the track fitting scenario once the strip values are determined the new clusters can be calculated. It is possible that more than one cluster is measured. The clusters are stored in a certain order. The method *First* returns the first cluster and the method *Next* returns the consecutive Cluster or null when no more clusters follow the last returned one. The method *Current* returns the last Cluster returned from SiBTOO by former invocations to *Next* and *First*. Projections are made of all possible cluster combinations of three or four horizontal or vertical planes.

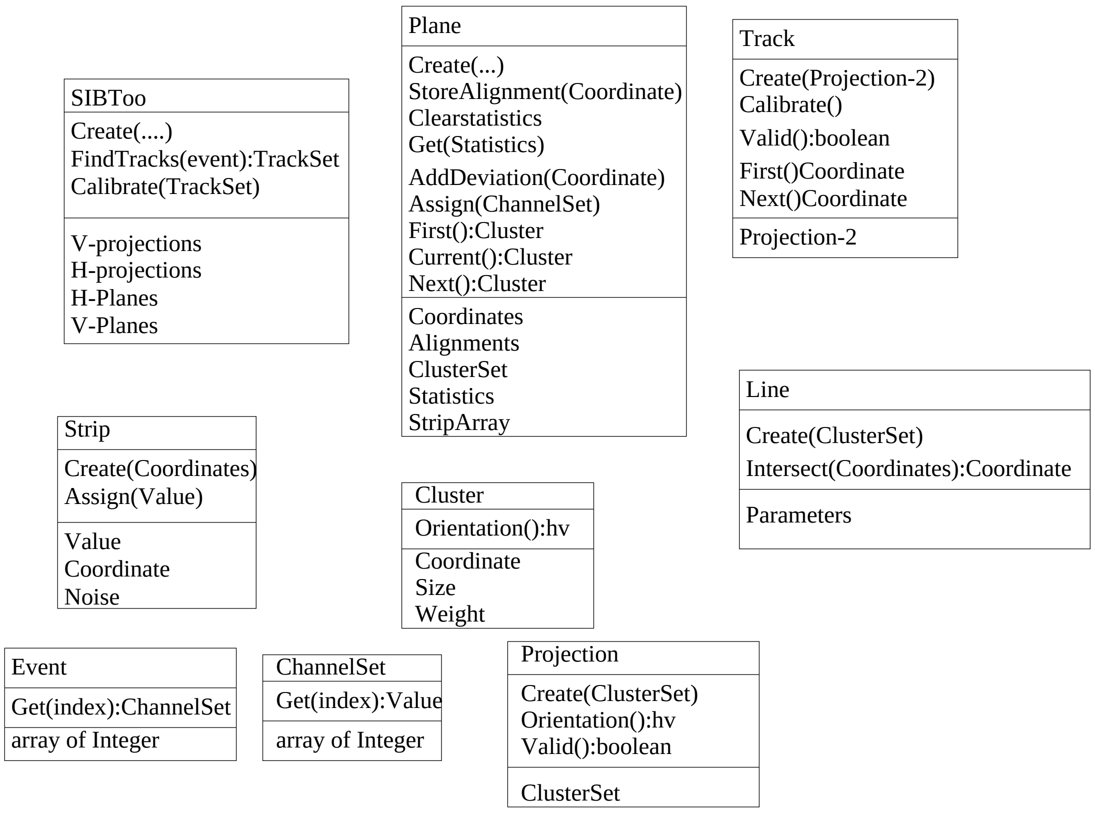

Figure 5: Association realizations

The calibrate scenario necessitates some additional methods. Statistics are gathered within the plane about the deviations calculated over a set of tracks. The routine *ClearStatistics* is needed to clear the results of a former calibration. The method *AddDeviation* adds a new deviation to the plane as calculated from the track fit. The method *Get* returns the result of all gathered statistics. The method *StoreAlignment* specifies the alignment that is calculated for this plane.

Strip The Strip class is intimately linked with the Plane class. The noise level, the value and the coordinates are implemented as public attributes for fast access within Plane. When, in a later stage, the event contents are modified, these attributes can be replaced with methods that calculate the corresponding values. A Strip needs to be created with the *Create* method. The *Assign* method takes the encoded value given as parameter, computes noise and signal value and stores them into the Strip attributes.

Event, ChannelSet Both classes are composed of arrays of integers in which strip- and noise values of the Strips are encoded. Each has a public routine *Get*. *Event.Get(index)* returns the ChannelSet belonging to the Plane identified by the index parameter. *ChannelSet.Get(index)* returns the encoded value belonging to the strip identified by the index of the earlier specified Plane.

Cluster The Cluster is a class that represents the position of the passage of a particle through a Plane. The criteria whether a certain set of strips represents the activation by a particle are the responsibility of the Plane class. Coordinate is a public attribute to be used by the Projection and Track classes. Coordinate is represented in a coordinate system that is independent of the Planes. Width gives the size of the cluster determined by the number of activated strips and Weight determines the importance of the Cluster. Weight has value 1 or 0. Zero means dummy or not applicable cluster. A public *Orientation* method returns whether the Cluster originates from a horizontal or vertical Plane.

Projection SiBTOO assigns three or four clusters from the same number of different Planes to the projection at the invocation of *Create*. After termination, the *Valid* method returns True if the clusters are aligned otherwise it returns False. The *Intersect* method determines the coordinate where the line fitted for the projection intersects a plane and adds the deviation between cluster coordinate and line coordinate to the plane statistics. ClusterSet is a public attribute that can be used by the Track.

Track At invocation of *Create*, two projections, one for each orientation, are assigned to Track. In *Create*, a line is fitted through them. If no fit can be done, the public *Valid* method returns False otherwise True. At invocation of *Calibrate*, the intersection point between each of the eight planes and the fitted line are calculated. The method *First* returns the cluster coordinate of the first plane. The method *Next* returns the deviation of the next plane.

Line The line is created from the ClusterSet communicated at invocation of *Create*. The intersection point with a plane specified by the parameters is returned after invocation of *Intersect*. To hide the plane code from the Track and vice versa, the cluster contains the plane coordinates from which it originates.

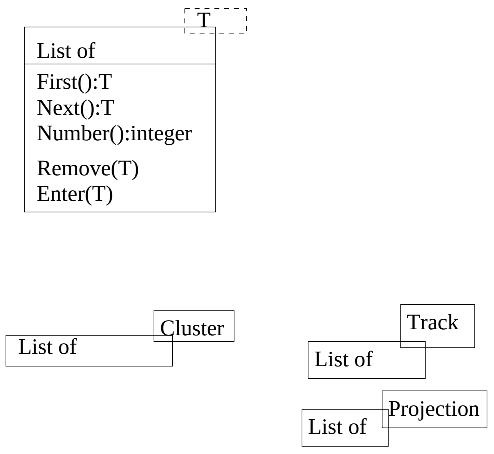

Figure 6: Lists

2.3.2 Lists

The use of lists is suggested by the above design. There are sets of Clusters in a Plane, sets of Clusters that represent a Projection, sets of Tracks and sets of Projections. The members of the sets are used in a certain order and the number of set members is variable. This observation indicates the storage of the members in a list with a variable number of items. It is not a very encouraging prospect to do all the work for a list of Clusters and then redo the same work for Tracks, etc.. The Template facility (or generic Class) offered by C++ makes it possible to specify a class ”List” of a not yet known type. Every time a list of a given type is needed a concrete instance of the list is defined. In Fig. 6 the template class list is defined with the possibility to enter and remove items from the list. Three concrete instances of List are shown as well.

The code for a Projection depends on the orientation of the Projection. This means that IF statements are needed to test the orientation and execute the appropriate part of the code. The IF statements can be avoided (thus simplifying the code) by using inheritance. An abstract Cluster and Projection are defined that are specialised to the concrete classes Cluster-H, Cluster-V, Projection-H and Projection-V. This is shown in Fig. 7. Code particular to horizontal and vertical projections are thus assigned to different classes. At this moment it is difficult to decide whether also two classes Plane-H and Plane-V are appropriate.

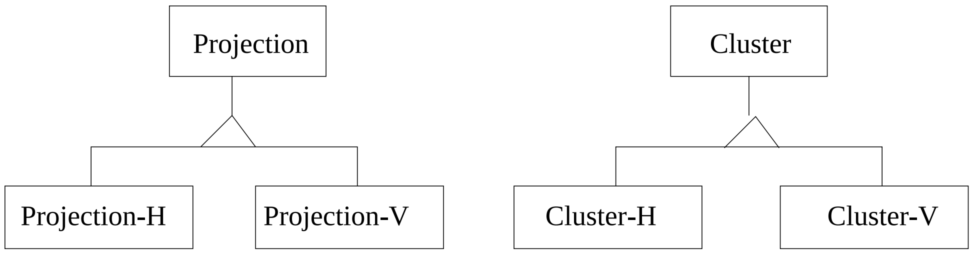

Figure 7: Projection and Cluster inheritance

2.4 Implementation

In the above we have done a top-down approach. Using the class diagrams and scenarios, a more detailed design and implementation are done in this section. The lowest level classes are implemented first and other classes are built on top, finally ending with a SiBTOO program in which tracks are recognised and calibration can be done. The final code is shown in appendix A.

List The most basic class is the list that will be used at many places. It is organised in a singly linked list. The variable *head* points to an object of class Link. Inside link, the attribute *val* is a pointer to an object of the parameter class T. The attribute *next* points to the next Link object in the list. The pointer *cur* is an attribute of list and points to the currently accessed item in list. The attribute *number* represents the number of items in the list.

```
template<class T> class list{
// parametrised class List
// singly linked list of items
// initialised with zero items, with head and cur initialised to null
protected:
        class Link{
        public:
             Link*  next;
             T*     val;
             Link(Link* n, T* v) {next = n; val =v; } ;
        };
        Link*   head;   // head of singly linked list
        Link*   cur;    // last selected item
public:
        int     number; // number of items in list
        list() {head = null; cur=null; number=0; };
        ~list(){
        while ( head != null){
            cur = head;
            delete head->val;
            head = head->next;
            delete cur;}
         };
```

A constructor and destructor are defined. After creation, *number* is equal to zero and *head* and *cur* are null (empty reference). The destructor assures that all elements in list are removed before the list descriptor is removed. Other attributes of list are :

- *enter* - enters an element into the list
- *first* - returns the pointer to the first element in the list, null if no elements are there.
- *next* - returns the next element in the list
- *current* - returns a pointer to the last accessed element in the list
- *sequel* - returns whether there are elements in the list after the last accessed one.

```
void  enter(T* item) {
      Link* temp = head;
      if (item != null){
          head = new Link( temp, item);
          number ++;}
      };
T*    first() {cur=head;
      if (cur == null) { return null;}
      else {return (cur->val);}
      };
T*    next() {
      if (cur == null){ return null;}
      else {cur = cur->next;
           if (cur == null){return null;}
           else {return (cur->val);}
           }
      };
T*    current() {
      if (cur == null){ return null;}
      else {return  (cur->val);}
      };
bool  sequel(){
      if (cur == null){ return false;}
      else {return cur->next != null;}
      };
```

The invocation of *first* initialises the value of *cur*. As long as *cur* is null, *current* returns null and *sequel* returns false. When the last element has been accessed, the invocation of *next* returns null and *cur* is set to null. Once the value of *cur* = null, the method *first* needs to be reinvoked to access again elements in the list. The use of *sequel* will become clear at a later stage in the implementation.

event, channelset Both classes are added for debugging purposes and are shown in appendix A without further comment.

cluster, strip and plane The class cluster plays a central role in SiBTOO. It uses a class coordinate that represents the cluster coordinates in a Cartesian coordinate system. A cluster represents a longitudinal coordinate (s_c) in one plane determined by the s-coordinate along the s-axis of SiBT and a transverse coordinate (t_c) that is either the x- or y- value of the cluster. The size attribute is a measure for the number of strips constituting this cluster. Remark that it is assumed that at least two strips constitute a cluster. This is refelected in the code of the plane shown later. A constructor defines the coordinates and size of the cluster. A destructor is not needed as no further structure is defined.

```
class coord{
public:
float   x,y,z;
};

class cluster{
protected:
coord   cc;         //coordinates of cluster
public:
float   size;       // number of hits (>1)
float   weight;     // weight =0 for dummy cluster
float   s_c;        // coordinate along s_axis (z_axis)
float   t_c;        // coordinate along transverse axis (x_ or y_ axis)
        cluster(float cx,float cy,float cs,float sz) {
        cc.x = cx; cc.y = cy; cc.z = cs; weight =1.; size = sz; }
};
```

The strip class is shown below and is self-explanatory.

```
class strip {
public:
int noise;      // noise measurement value
int value;      // value caused by passage of particle
float dist;     // distance from origin along x- or y-axis
float width;    // width of strip along x- or y-axis
float length;   // length of strip over s-axis
};
```

The plane class is clearly one of the more complex classes. It contains an array of strips, a list of clusters, the coordinates in space of the lower left corner of the plane, and the statistics represented by *mean* and *nr* -the number of entries over which *mean* is calculated-. In the constructor an empty list of clusters is defined and the lower hand coordinates of the plane specified in the constructor parameters. The noise of the strips is initialised to 1 as for cluster calculation the acquired value is divided by the noise and division by zero is not allowed. This way a test on zero is not needed (efficiency). Two virtual methods are defined: *storealignment* and *addcluster* that need different implementations dependent on the projection orientation (shown below). The *adddeviation* method adds a new difference value to the plane and recalculates the mean deviation.

```
const  int      nstrip = 1024;  // number of strips in plane
class plane {
protected:
list<cluster>*  clusters;      // cluster set
strip           sl[nstrip];    // array of strips
float           x,y,s;         // coordinates of lower left corner of plane
float           mean;          // statistics
int             nr;            // number of entries

public:
        plane(coord c){
        clusters = new list<cluster>;   // empty cluster list to allow
                                        // access to clusters
        x = c.x; y=c.y ; s = c.z;       // coordinates of plane
        mean = 0.; nr = 0;              //statistics for calibration
        for (int i = 0; i<nstrip; ++i) {
             sl[i].value = 0;
             sl[i].noise = 1;}          // to prevent division by zero
        }
cluster* first(){return clusters->first();};
cluster* get(){return clusters->current();};
cluster* next(){return clusters->next();};
bool     sequel(){ return clusters->sequel();};
void     assign(channelset s);
void     clearstatistics(){ mean = 0.; nr = 0;};
float    meandev(){ return mean;};
float    scoord() { return s;};
virtual  void    storealignment(float cc){  };
void     adddeviation(float dt){
         if (nr == 0) {mean = dt; nr = 1;}
         else { mean = mean*(nr/(nr+1)); nr ++; mean = mean+(dt/nr);}
         };
virtual void addcluster(float t_c, float s_c, float sz) { };
};
```

Below, the cluster calculation after assignment of the measured values is shown in the *assign* method. The parameter, *ss*, is of type channelset in which the measured values of this plane are defined. First the strip values are initialised and a new empty cluster list is set up.

The clusters are calculated by doing a loop over all strips. At the line marked by (==>) a hit has been found and all consecutive strips are inspected to verify whether they also represent a hit. Two values *i* and *k* represent the first and last strip associated with the cluster. To test whether the "while loop" must end, a boolean *b* is used. The end criterium of the while loop is that *k<nstrip* and *sl[k]* contains a hit. However, because the test on *k* must be done first and only when *k<nstrip* the test on *sl[k]* can be done, the present construction with a boolean is used. When *k>i* and at least two consecutive hits have been found, a cluster is added to the attribute *clusters* -the cluster list of the plane-.

```
void plane::assign(channelset ss){
      for (int i = 0; i< nstrip; ++i) {

         // strip value and noise from channelset
              sl[i].value = ss.strip(i,1);
              sl[i].noise = ss.strip(i,2);}     // strips initialised

      delete clusters;                         // remove former clusters
      clusters = new list<cluster>;            // create empty cluster list
      for (i = 0; i< nstrip; ++i) {

         // determine clusters
          if (sl[i].value/sl[i].noise > hitquot){  // hit found in strip i
              float total =sl[i].value-sl[i].noise;
              float posit = sl[i].dist*total;
              int   k = i+1;
              bool  b = (k < nstrip);
              if (b) {b = b & sl[k].value/sl[k].noise > hitquot;}
==>            while (b) {
                  total =total+(sl[k].value -sl[k].noise);
                  posit = posit+sl[k].dist *float(sl[k].value-sl[k].noise);
                  k++;
                  b = (k < nstrip);
                  if (b) {b = b & sl[k].value/sl[k].noise > hitquot;}
                  }  // end while (b) loop

              k--;     // k always one too large
              if (k > i) {

                   //more than two consecutive hits, add cluster
                   addcluster(posit/total, s,
                                    sl[k].dist-sl[i].dist+sl[k].width);}
                   i = k+1;}        //skip already inspected strips
           }   // end if ( sl[i]........
} // end assign
```

horizontal cluster and plane Both cluster and plane need the inherited classes v_plane, h_plane, h_cluster and v_cluster to make the code independent of the orientation. Below is shown how. A cluster is defined by three coordinates of which in a particular plane only two are interesting: the transverse and the longitudinal coordinate. Therefore, a horizontal (or vertical) cluster can be defined with these two coordinates and the other coordinate *y* (*x* for the vertical cluster) is filled with 0. The code of *assign* is orientation independent as long as clusters of the right orientation are created and added to the attribute *clusters*. This is possible when the *addcluster* method of plane creates clusters with the correct orientation (vertical or horizontal). This is assured by defining *addcluster* virtual in plane and specifying it for h_plane and v_plane separately.

```
class h_cluster: public cluster{
public:
        h_cluster(float cx,float cs,float sz)
             :cluster(cx, 0., cs, sz) {          // y-coordinate =0
                      t_c = cx; s_c = cs; }
};
```

For a plane three methods are orientation dependent: the constructor, *addcluster* and *storealignment*. For the calculation of the strip coordinates different sizes for horizontal and vertical planes are possible, defined by constants *xwidth* and *ywidth*. In the constructor h_plane (v_plane) the correct values along the x-axis (y-axis) are defined.

In *addcluster* a horizontal (vertical) cluster is created with new. The right cluster coordinate along the x-axis (y-axis) is calculated by adding the x- (y-) coordinate of the plane.

The method *storealignment* is self explaining.

```
class h_plane: public plane{
public:
        h_plane(coord cc): plane(cc) {
        for (int i = 0; i<nstrip; ++i) {
             sl[i].dist = xwidth*float(i);   // x-position of strips from x=0
```

```
            sl[i].width = xwidth;           // dx width of strips
            sl[i].length = length;}         // thickness of strip over s-axis
        }
void    addcluster(float t_c, float s_c, float sz) {
     // add x coordinate of plane to position within plane
            h_cluster* h_cl = new h_cluster(t_c+x, s_c, sz);
            clusters->enter( h_cl);
        }
void    storealignment(float cc){ x = x+cc;};
};
```

line, projection A line is defined in a flat plane and is given by the formula

$$t = \text{ts\_tg} * s + \text{s\_off} \qquad (1)$$

where s_off and and ts_tg represent the offset and slope respectively. The constructor creates a line with the specified offset and slope. The *fit* method takes as parameter a list of clusters. *Fit* is independent of the orientation of the clusters. It is assumed that they are all horizontal or vertical. The following items are calculated for all clusters c: $\sum_c s_c$, $\sum_c s_c * tc$, $\sum_c t_c$, $\sum_c s_c * s_c$ and $\sum_c t_c * t_c$. In the usual way the offset and and slope are calculated from these values. The method *fit* returns false if the number of clusters is smaller than three, determinant is smaller than 0 or χ^2 is greater than some constant *linchi*. Otherwise, *fit* returns true.

```
class s_line{
public:
float   s_off;
float   ts_tg;
float   chi;
        s_line(float xt,float yt,float of){
                s_off = of; ts_tg = yt; chi = 0;}
bool    fit(list<cluster>* proj) {
        cluster* clpt = proj->first();
        int     ncl =0;                             // number of clusters
        float   S_s = 0.; float   S_ss = 0.;
        float   S_t = 0.; float   S_tt = 0.; float   S_st = 0.;

        while ( clpt != null){
                if (clpt->weight > 0){
                        ncl = ncl + 1;              //  number of entries
                        S_s = S_s+clpt->s_c;                //sum over s
                        S_t = S_t+clpt->t_c;                //sum over t
                        S_ss = S_ss+clpt->s_c*clpt->s_c;    //sum over s*s
                        S_st = S_st+clpt->s_c*clpt->t_c;    //sum over t*s
                        S_tt = S_tt+clpt->t_c*clpt->t_c;}   //sum over t*t
                clpt = proj->next();
                }
        if (ncl < 3){ return false;}    // at least three coordinates required
        float det = float(ncl)*S_ss - S_s*S_s ;
        if (det <= 0) { return false;}
        s_off = (S_ss*S_t - S_s*S_st)/det;
        ts_tg = (ncl*S_st - S_s*S_t)/det;
        chi = S_tt-s_off*S_t - ts_tg*S_st;
        return abs(chi) < linchi;
};
```

The projection contains a list of clusters that define the projection and a line as defined above. The constructor defines an empty list of clusters and a line with offset = 0, slope = 0 and χ^2 = 0. The method *add* adds a cluster to the list of clusters. The assumption is that only clusters of a given orientation are entered. This is not checked. The *valid* method invokes the line *fit* of the cluster list and returns the result of the fit to the *valid* invoking code. The method *tline* calculates the t-coordinate of the line for a given s-coordinate defined as the parameter *sc*.

The method *AddDev* needs more explanation. The assumption is that clusters are added in the plane number order, i.e. first a cluster of plane 0 then of plane 1 and finally clusters from planes 2 and 3 are entered. The first cluster to be returned is the cluster from plane 3 or failing that one from plane 2. At the start of *AddDev*, the variable *cp* contains the first cluster of the projection and the index *i* is initialised to 3. In the while loop it is checked that the cluster s-coordinate is equal to the plane s-coordinate. When this is not the case, plane i did not contribute to this projection and *i* is decreased until both s-coordinates match. For matching coordinates the t-coordinate value of the cluster is subtracted from the line coordinate and added to the deviation statistics of the plane. The loop is executed over all clusters in the projection and ends when *clp*$-$ $>$*next()* returns null, meaning that all clusters from *clp* have been used.

```
const  int      nplane = 8;    // number of planes in SiBT
class projection{
protected:
list<cluster>*  clp;    // pointer associated cluster
s_line  line;   // fitted line
public:
        projection(): line(0,0,0) { clp = new list<cluster>; }
        ˜projection(){ delete clp;}
bool    valid(){ return line.fit(clp);}
float   tline( float sc){ return line.ts_tg*sc + line.s_off;}
void    add(cluster* clref) { if (clref != null) {clp->enter(clref);}}
void    AddDev( plane* planes[]){
        cluster* cp = clp->first();
        int i = (nplane/2)-1;
        while (cp != null) { // assume that clusters come from 3,2,1,0
                while (cp->s_c != planes[i]->scoord() & i >0) { i--;}
                planes[i]->adddeviation(cp->t_c - tline(cp->s_c));
                cp = clp->next();}         // end while over clusters
        }
};
```

projlist, track The class projlst is added to construct all projections possible from a set of four horizontal or four vertical planes. The class is mainly introduced to avoid duplification of code for projection finding in the horizontal and vertical plane. A destructor and constructor are needed to initialise attribute *lp* and remove the list of projections pointed to by *lp*.

FindProj is the main method that takes an array of planes as parameter. *FindProj* can be invoked for horizontal and vertical planes. It is based on the following principle. Suppose that each plane contains two clusters, notated as: $p_x c_y$ denotes cluster y of plane x. For example: $p_0 c_1$ is cluster 1 of plane 0. The following projections are then constructed in this order:

$$p_0c_1\ p_1c_1\ p_2c_1\ p_3c_1$$

$$p_0c_2\ p_1c_1\ p_2c_1\ p_3c_1$$

$$p_0c_1\ p_1c_2\ p_2c_1\ p_3c_1$$

$$p_0c_2\ p_1c_2\ p_2c_1\ p_3c_1$$

$$p_0c_1\ p_1c_1\ p_2c_2\ p_3c_1$$

Etc....

$$p_0c_2\ p_1c_2\ p_2c_2\ p_3c_2$$

such that all combinations have been done. In the last case it can be seen that for all planes the last cluster has been used. In all other cases there is at least one plane where cluster 1 has been used. This leads to the introduction of the boolean *b* for which the value true represents: there is a plane that has not returned the last cluster. Boolean *b* = false means: all planes have returned their last cluster. Testing whether the last cluster is returned is done with the method *sequel* that returns true if there is a cluster left in the list of clusters in the plane.

The first projection, *pp*, is found by returning the first cluster from each plane. The valid projection, *pp*, is entered in the projection list *lp*. The initialisation of *lp* is discussed later. The boolean *b* is True when other projections are possible. The second part of *FindProj* is started at the sign $==>$. Another boolean *sw* is introduced. When *sw* is true, the next cluster of a plane is solicited or the first cluster in case no more clusters are left in the plane. Variable *sw* is set to true at the start of the loop over the planes to assure that always a fresh cluster is entered from plane 0. Variable *sw* is true for plane i+1 when the first cluster had to be reread from cluster i. In all other cases *sw* is set to false. When *sw* is false the current cluster is taken from the plane. When *sw* is true, the next cluster (if available or the first cluster) is inserted in the projection.

At the end of the loop over the planes, a valid projection is added to the list. The loop over all projections is ended when *b* reaches false. At the end of *FindProj* the attribute *lp* contains a list of 0 or more valid projections.

It is, of course, possible that a plane contains no clusters. Then *first* returns null. In *enter* a test on null is done. Nothing is entered in the list when the argument of *enter* is null. This caters for projections with less than four clusters.

```
class projlst {
protected:
list<projection>*  lp;
        projlst() { lp = null;}
        ˜projlst() { if (lp != null) { delete lp;} }
public:
projection*     first() { return lp->first();}
virtual projection*     newproj() { return new projection();}
int     number() { return lp->number;}
virtual void    FindProj(plane* planes[]){
// find the list of projections in array of planes

        bool b = false;
        projection*  pp = newproj();
        for (int i = 0; i<4; ++i){
            pp->add(planes[i]->first());
            b = b | planes[i]->sequel();}  // more than one cluster in plane?

        // b implies there is a plane with more than 1 cluster
        // not b implies all planes have less than 2 clusters
        if (pp->valid()){lp->enter(pp);} // if good projection add to lvp
        else {delete pp;} // else remove contents and memory

==>
// try all possible cluster combinations from vertical planes that have clusters

        cluster* cp;
        while (b){

            pp = newproj();
            b = false;
            bool sw = true;

        // sw implies get next cluster from plane or first one
        // not sw implies get current cluster from plane

             for (i = 0; i<4; ++i){
                 if (sw) {cp = planes[i]->next();}
                 else { cp = planes[i]->get();}
                 if (cp == null) { cp = planes[i]->first();}
                 else { sw = false;}

//  sw remains true when first cluster was taken from former plane
                  pp->add(cp);
                  b = b | planes[i]->sequel();}  // end for (i=0; i<4)

        // not b implies last cluster is found in all planes

              if (pp->valid()){lp->enter(pp);}
              else {delete pp;}
           }                            // end of while(b)
        }                               // end of FindProj
};                                      // end of projlst class
```

A track is composed of one valid horizontal and one valid vertical projection. Two methods, *hproj* and *vproj*, return the horizontal and vertical projection respectively. A destructor is needed to remove the horizontal and vertical projection when the track is deleted.

```
class track{
projection*   hpr;
projection*   vpr;
public:
         track(projection* h, projection* v) {hpr = h; vpr = v;}
         ~track() { delete hpr; delete vpr;}
projection* hproj() {return hpr;}
projection* vproj() {return vpr;}
};
```

horizontal projection list A horizontal (vertical) projection and projection list have been defined as well. Their reason of existence is explained in the paragraph on visualisation of data structure. The method *newproj* is used to have *pp* point to horizontal (vertical) projections. Consecutively, a horizontal projection list is filled with horizontal projections.

The method *FindProj* is the application interface to projlst. It initialises *lp* with an empty list of horizontal projections after which *FindProj* is invoked and all manipulations in *FindProj* only take place on horizontal projections.

```
class h_projlst: public projlst {
protected:
projection*     newproj(){ return new h_projection();}
public:
void    FindProj(plane* planes[]){
// find the list of projections in array of vertical planes

        delete lp;                      // remove old list
        lp = (list<projection>*) new list<h_projection>;
        projlst::FindProj( planes);
        }
};
```

SiBTOO The class SiBTOO represents the silicon beam telescope SiBT. It consist of four horizontal and four vertical planes. Additionally it maintains a list of horizontal and vertical projections in the attributes *lhp* and *lvp* respectively. A constructor is provided. Remark that *lhp* and *lvp* are initialised with h_projlst and v_projlst respectively. No destructor is provided because SiBTOO is never removed. If SiBTOO is removed in an application, a destructor is needed to remove the planes and the projection lists.

Two methods are provided *FindTracks* and *Calibrate*. *FindTracks* initialises the strip values of all 8 planes, finds the vertical and horizontal projections and if one projection is found in each direction a track is returned. The method *Calibrate* takes as input a list of tracks and loops over all tracks. For each track and for each projection in the track, deviation statistics are added to all planes. After treating all tracks, the mean deviations stored in the planes are added to the plane coordinates.

```
class SiBTOO{
plane*  hplanes[nplane/2];     // 4 horizontal planes
plane*  vplanes[nplane/2];     // 4 vertical planes
h_projlst*   lhp;              // list of horizontal projections
v_projlst*   lvp;              // list of vertical projections

public:
        SiBTOO() {
```

```
        // initialise with 0 projections
        lhp = new h_projlst();
        lvp = new v_projlst();

        //initialise plane coordinates
        for (int i = 0; i<nplane/2; ++i){
                coord cc;
                cc.z = 2*i*(sbt_ds-0.05)/nplane;// s-coordinate
                cc.x = .1;                      // x-coordinate
                cc.y = cc.x;                    // y coordinate
                vplanes[i] = new v_plane(cc);   // vertical plane
                cc.z = cc.z+.05;                // s-coordinate
                                                // (slightly different from
                                                // vertical plane s-coordinate)
                hplanes[i] = new h_plane(cc);}  // horizontal plane
        }

track*  FindTracks(event ev){
        for (int i = 0; i<nplane/2; ++i){
            vplanes[i]->assign(ev.plane(i*2));    // vertical plane strips
            hplanes[i]->assign(ev.plane(i*2+1));};// horizontal plane strips

                // clusters of event are found and stored in planes

        lvp->FindProj( vplanes);        // lvp contains vertical projections
        lhp->FindProj( hplanes);        // lhp contains horizontal projections

        // only one projection from each orientation is allowed
                if(lhp->number() == 1 & lvp->number() == 1)
                                {return new track(lhp->first(), lvp->first());}
                else { return null;}
        }

void    Calibrate(list<track>* lst){
        for (int i = 0; i<nplane/2; ++i){
                vplanes[i]->clearstatistics();
                hplanes[i]->clearstatistics();}
        track*  trp = lst->first();     // get first track from list
        while (trp != null) {           // loop over all tracks

        // horizontal projections
                trp->hproj()->AddDev( hplanes);
        // vertical projections
                trp->vproj()->AddDev( vplanes);

                trp = lst->next();} // end while over all tracks
        // realign planes
        for (i = 0; i<nplane/2; ++i){
                hplanes[i]->storealignment(-hplanes[i]->meandev());
                vplanes[i]->storealignment(-vplanes[i]->meandev());}
        }
};
```

Show contents of SiBTOO For every class a *print* function is added. Below, the *print* function of SiBTOO is shown. It prints the contents of the horizontal and vertical projections and of the individual planes. Here the inheritance of vertical planes and horizontal planes is motivated as the *print* function of the horizontal plane prints that it is a horizontal plane without any test on orientation parameters. Instead of printing, more sophisticated graphics visualisation routines can be foreseen that actually exploit the orientation of the object to visualise.

```
void    SiBTOO::print(){
        lhp->print(); lvp->print();
        for (int i = 0; i<4; i++){
                vplanes[i]->print(); hplanes[i]->print();}
        }

void    h_plane::print(){
        cout << " HORIZONTAL ";            plane::print();
        cout << "    clusters are: \n";   clusters->print();
        cout << " --------------------- H_PLANE \n \n";
        }
```

Finally, the *print* of the list is shown. For every item in the list, it invokes *val->print()*. This means that every class for which a list is defined, should provide its own *print()* method. More elaborate code is shown in the appendix A.

```
void    list::print(){
        cout << " list with " << number <<" entries \n" << flush;
        int k =0; Link* temp = head;
        while (temp != null){
                cout << " item "<< ++k << ":   "; temp->val->print();
                temp = temp->next;}    // get next element from list
        }
```

2.5 Evaluation

In a design it is always difficult to decide whether efficiency should be the most important aspect or generality. This is the case in the design of the s_line class. The *tline* code of the projection class uses the ts_tg and s_off attributes of line directly to calculate the intersection point. When other lines instead of straight lines are used, this is unacceptable. Then a mehod *t_coordinate* of line should return the transverse coordinate given the longitudinal coordinate. Substituting the line class for circle or helix class leaves the *tline* method of the projection class unmodified.

For performance reasons also the strips are arranged in an array and attributes of strips are addressed directly and not via a method.

As can be seen in the code it is assumed that the minimum number of activated strips in a cluster is 2. This shows in the design of the *assign* method where two variables *i* and *k* are used with the assumption that $k > i$ for a valid cluster. Another point is the assumption that the number of horizontal planes and vertical planes is the same as reflected by the choice of a constant *nplanes* that represents all available planes.

It is also debatable whether the inheritance for creating horizontal and vertical clusters, projections and planes is really necessary. These points are addressed in the excercises below. However, the introduction of more specialization in the planes favours the actual design choice of using inheritance.

A few clear advantages and general design rules need to be pointed out:

- **Information hiding** The clearest example is the list, where all pointer handling or list structure is hidden from the other classes. The structure of the list (e.g. doubly linked) can be changed without any changes to the remaining code. Other examples are:
  - **ProjLst** The algorithm to attribute clusters to a projection is completely confined to the ProjLst class.
  - **Projection** The testing of the validity of a projection and the determination of the measurement mismatch of projection clusters is confined to the projection.
  - **Plane** The calculation of clusters is confined to the plane class. Differences between horizontal planes and vertical planes can be added without modifications to the code.
  - **Line** The line fitting algorithm is confined to the s_line class.
- **Divide and Conquer** The above cited classes show that to each class a limited functionality is attributed. The development of small pieces of code with a limited functionality is easier than making large monolithic pieces of code.

- **Performance** Most classes have interfaces defined by functions. A few classes allow direct access to attributes. This is done for classes where little change is expected or/and that are used by a very limited $(, 4)$ number of classes.

- **Case dependency** Methods that contain many IF statements usually contain too much functionality. The use of inheritance to separate these functions to simplify the code is then recommended. Here, this is done for planes with horizontal and vertical orientation. Inheritance leads to more complexity due to the multiple definitions. The advisability of inheritance must be judged on the basis of the properties of the individual cases. Here, we are confronted with an unclear case.

- **Generality** When inheritance is used, the danger exists that almost identical code that differs at only a few locations is repoduced as many times as there are derived classes. It is advisable to construct methods in the base class that can be used as much as possible for all inherited classes. This is established for *FindProj*, *AddDev* and *assign*.

- **Layering** The code is built up of classes with more and more capability by exploiting the functionality of the classes from lower layers (for example: a projection that uses a cluster that is based on strips).

Not following these guidelines leads to monolithic programs with many dependent execution paths that are difficult to test and maintain. In such programs, modifications have the nasty habit not to stay localized but needing unexpected additional modifications. For example in this example responsibiliy for cluster allocation, projection finding, line fitting etc are assigned to different classes. Another possibility is to group all this functionality inside the SiBTOO class. The other classes are then reduced to simple data containers. It is left as an execercise to the reader to see the consequences for the final code.

2.6 Exercises

1. Remove all horizontal and vertical inheritance and add a boolean to the base class to assign an orientation to the class. Modify all orientation dependent methods to obtain the same results as before.

2. Change the vertical plane such that there is an empty space between any two strips.

3. Change the line fitting and add the possibility to do a circle fitting. Both fit possibilities exist simultaneously. Straight line fits are done for four cluster projections and circle fits are done for three cluster projections.

4. Change the number of planes from 8 to 10.

5. Calculate the cluster width in an orientation dependent way.

6. Treat the planes as pairs and calculate clusters on a pair basis. This should lead to quite a lot of changes. Try to identify the code that needs no change.

7. Determine the quality of the realignment by calculating the standard deviation of the plane alignment statistics during calibration.

8. Make the list of clusters an ordered list and order clusters according to s coordinate value.

9. Change the code such that the type checking of C++ verifies that vertical (horizontal) clusters are added to vertical (horizontal) projections.

3 Layers example

This example is based on the layers that can be identified in the CMS detector. The layer structure is used to do a fast track searching algorithm starting from the outside of the detector and ending at the centre. Within a detector installation different types of detectors can be found. It is shown how the SiBT can be considered as a detector type from the set of detector types that constitute the detector installation (this is actually not true but is assumed as a hypothetical example).

3.1 Layers description

In Fig. 8, a detector installation composed of several detectors with a circle symmetric layout is shown. Particles originate at the centre and the passage of the particles is recorded by the detector through which they pass. Detectors in the detector installation can have different characteristics. Several types of detectors can be discerned and usually detectors of the same type are organised in one layer around the centre. The detectors have one thing in common: the detection of the particle within the detector volume. Per particle that passes, several points are measured where the particle interacts with the detector. This is similar to SiBT where the passage of a particle is detected at eight different places. The number of places that a particle can be detected, can change from detector to detector. Also the sensitivity for a particular type of particle changes from detector to detector. Although SiBT was capable of detecting the passage of at most one particle, other detectors are capable of detecting the passage of several particles simultaneously.

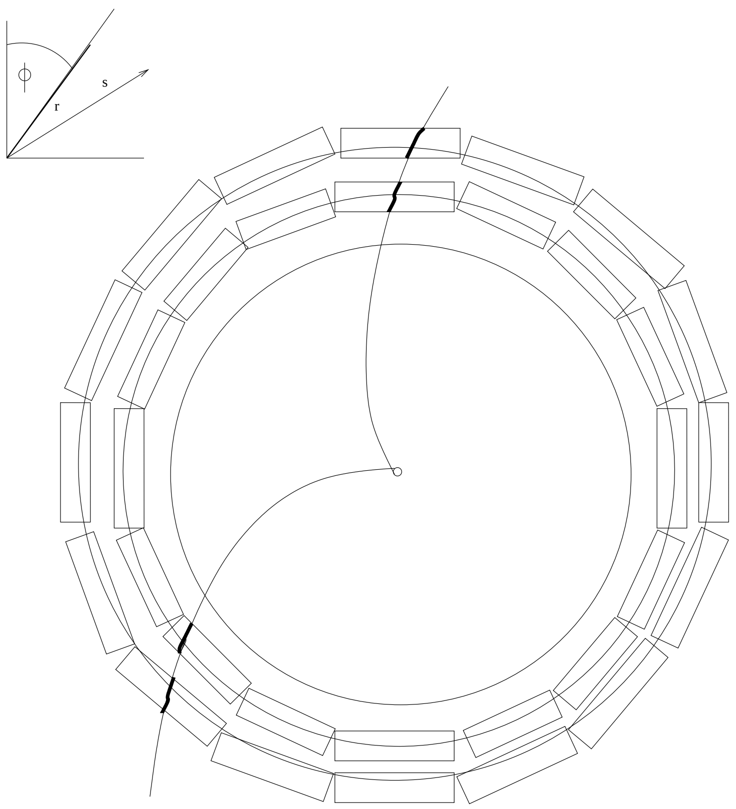

Figure 8: Diagram of detector layers

A series of points that are associated with the passage of one particle through one detector is called a track segment. The determination of a track segment inside a detector may change from detector to detector. In some detectors track segments are identified by fitting a straight line through the measurement points, in other detectors a helix or circle is fitted through the points.

In this section a series of segments from a set of detectors associated with one particle is called a track. An algorithm tries to combine all track segments to a candidate track. An example algorithm looks at the largest outer circle and determines all the points at which a particle leaves the detector installation. It associates each track segment with one particle. For all these track segments it determines the location where the particle enters the outer layer. It then inspects the next lower layer and combines the track segments that leave the lower layer at some position, with track segments of the outer layer that enter the outer layer at the same location. This goes on until the lowest layer (the centre) is reached. It is possible that a track segment can be combined with more than one lower level track segment. When n valid combinations are found, n candidate tracks are created. When all layers have been tried, a helix is fitted through all measurement points of a candidate track. Candidate tracks for which the helix fit is not good enough are rejected. The remaining candidate tracks are called valid tracks.

3.2 Analysis

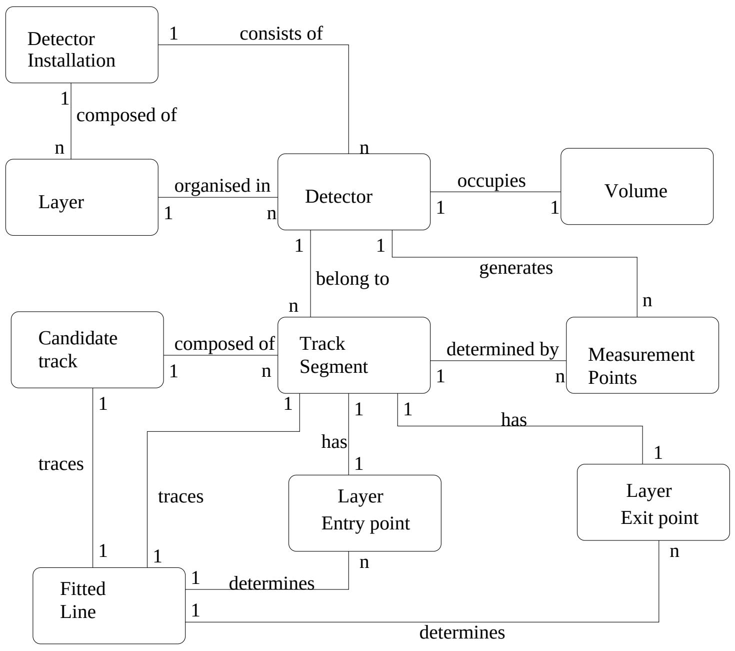

Figure 9: Class diagram based on layers analysis

The classes associated with the analysis are shown in Fig. 9. A detector installation (e.g. CMS) consists of a set of detectors that are organised in layers. Each detector occupies a volume in space where particles can be detected. The passage of a particle generates measurement points within the detector volume. These measurement points are represented in the coordinate system of the detector installation that can be different from the coordinate system of the individual detectors. For the detector installation a cylindrical coordinate system looks appropriate, while a Cartesian coordinate system seems reasonable for an individual detector (see SiBT). Measurement points are measured with a precision that is detector dependent. The measurement points inside a detector define track segments. Fitting these track segments with a line allows to extend the track segments and determine the exit and entry points of the segment from/to the detector volume. Track segments leave and enter the volumes with a certain orientation in space. Track segments are combined to candidate tracks. A track contains a given track segment only once but a segment may belong to several candidate tracks. Again, a line can be fitted through the measurement points of a track. The intersection points of this line with the different detector volumes determine the exit and entry points of the track.

This analysis is based on the description of section 3.1 and the use case described below. It is assumed that for a given event all detectors have combined the measurement points into track segments.

1. **Combine segments** It is assumed that detectors return zero or more track segments expressed in the coordinate system of the detector installation. The detector installation is constructed in concentric layers of detectors. Track segments of adjoining layers are related when their entry- and exit- point at the layer interface coincide and the orientation of the segments is identical. A set of related segments is combined to a track. Tracks can start at the centre or within the detector installation volume. Tracks can terminate in the detector installation volume or at the outside of the detector.

3.3 Design

Scenario diagrams are used to show the responsibilities for the different proposed classes. *Main* is the actor for the use case mentioned above. The same design aims as those for SiBT are applied here as well. The class Installation hides all implementation details from *Main*.

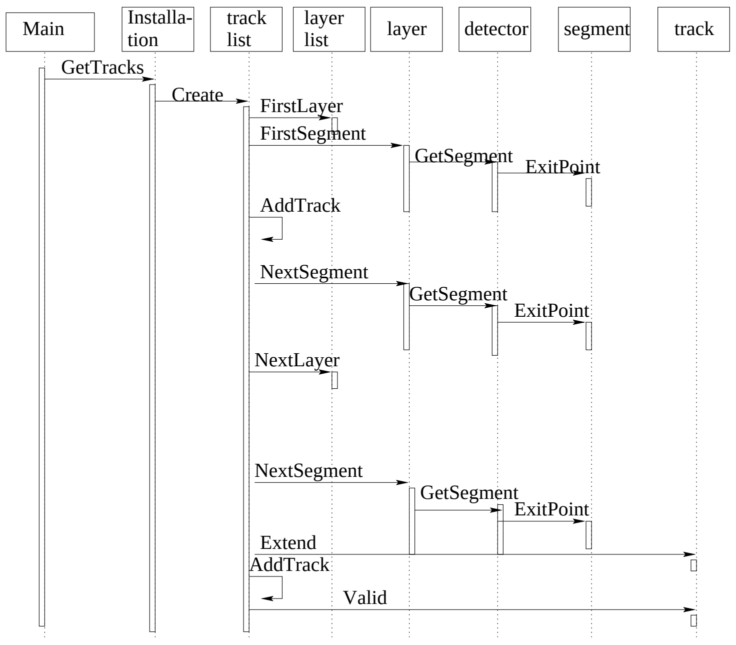

Figure 10: Scenario for determining a list of tracks

Combine segments The central idea is to visit each track segment only once during the construction of all possible tracks. Therefore, it is assumed that in each layer the detectors are ordered according to the angle in the cylindrical coordinate system and the track segments inside the detector are ordered accordingly. The tracks in construction should be ordered also according to the angle of the track at the lower layer interface. On the basis of the angle the next track segment from a layer can be searched or the next track in construction.

The scenario of Fig. 10 supports this construction. Similar to SiBT, a tracklist is constructed. The tracks in the list are created from a segment or tracks are extended with segments. The track creation and extension is started for a given layer by creating the tracklist class. The tracklist gets the first layer from the layerlist and asks the layer to return the first segment from this layer. The layer invokes the detector that returns the segment with the smallest ϕ angle. The segment is added to the list of tracks as the start of a track. This is repeated for all detectors in the layer. Invoking all layers -from the outside to the inside consecutively- leads to the construction of all possible track candidates. In the scenario, after invocation of GetTracks, the track list is created. At the start, track list retrieves the first (outer) layer from the layer list, continuing for all possible segments in the installation. When all layers have been invoked, track list asks every track to determine its validity. Invalid tracks are removed. The final implemented scenario is different from the one proposed here. This is explained in section 3.4.

3.3.1 Tentative class definitions

In this section a first effort is done to determine the required classes with their attributes and methods. The class set is a guideline for the final implementation presented in section 3.4. The following classes are needed:

Installation The Installation class is the computer model of the whole detector installation. Hidden attributes are LayerList that represents the list of layers and TrackList that represents the tracks found in the detector installation. Two public methods are needed: (1) *Create* to create the whole detector installation (2) *GetTracks* that returns the tracks found in the installation associated with an event. This is in analogy with the SiBT example.

Layer The class Layer models the set of detectors that form a layer around the inner detectors. The ordered list of detectors is a private attribute of Layer. The public methods *FirstSegment* returns the segment with the smallest ϕ coordinate detected within a detector in the layer. The public method *NextSegment* returns the segments with a ϕ coordinate that is larger than the former returned segment but with the smallest ϕ coordinate of all not yet returned segments in this layer. The public method *Assign* assigns a set of measurement values to the detectors in the layer.

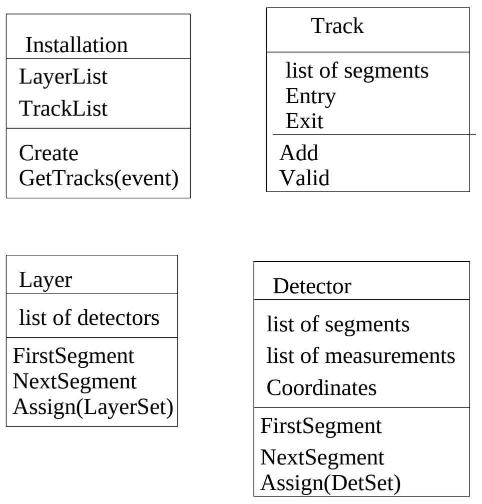

Figure 11: Class diagram based on scenario

Detector This class models a piece of equipment called a detector. The private attribute *Coordinates* represents the position of the detector in the installation's coordinate system. Within the detector, measurements are stored that can be combined to segments. These are represented with the ordered private attributes: ”list of segments” and ”list of measurements”. The public method *Assign* assigns a set of measurement values to the detector. The public methods *FirstSegment* and *NextSegment* have the same functionality as the corresponding methods of the layer.

TrackList The class TrackList represents the tracks that are being recognised on the basis of the measurements contained in the detector objects. The ordered private attribute ”list of tracks” represents the list of already identified tracks. The public method *Extend* adds new tracks to TrackList or extends tracks within the tracklist. The public *Create* method creates an object of this class.

Track An object of the Track class represents one candidate track. The private attribute ”list of segments” represents all segments that constitute this track. The private attributes *Entry* and *Exit* represent the entry and exit points of the track respectively. The public attribute *Add* extends the track with a segment. The public attribute *Valid* tests whether the track is a valid one.

Segment This class represents the measurements in one detector associated with one particle. The private attribute ”list of coordinates” represents the track measurements expressed in the coordinate system of the installation. The public attributes *Exit* and *Entry* represent the points where the particle respectively leaves and enters the layer associated with this detector. The public methods *Enter* and *Valid* add measurements to the segment and test whether the segment is a valid one, respectively. Validity is tested by fitting a straight line.

Event, LayerSet These classes (not shown in the Fig. 11) represent the measurements associated with the whole installation, a layer and a detector respectively.

3.3.2 Discussion

The same design strategy has been used as for the SiBT example. There is a clear analogy between installation and SiBT, planes and layers, strips and detectors, clusters and segments and projections and tracks. Analogous to the ProjList class a TrackList has been defined. A TrackList object combines the measurements of different layers to create complete tracks. The particulars of a specific track recognition algorithm are confined to the TrackList class. The fitting of a line with a particular geometry is confined to the Segment class. As in the SiBT example, lists of segments, tracks, measurements and detectors are needed.

3.3.3 Ordered lists

Similar to the lists of SiBT, the lists of the layered installation needs to be ordered according to the ϕ coordinate of the segment, track or detector. The ordering can be added to the list defined for SiBT, but this is deemed to much work considering that ordered lists are supported by the STL library [MS96]. Sorted lists are wanted from which elements can be retrieved ordered according to a key. The key here is the ϕ angle of the installation's coordinate system. No random access is wanted. The order in the list of tracks is changed infrequently. This leads us to the choice of the simplest STL container: the vector. The list is another good candidate but the syntax allowed by the STL library made it a poor choice.

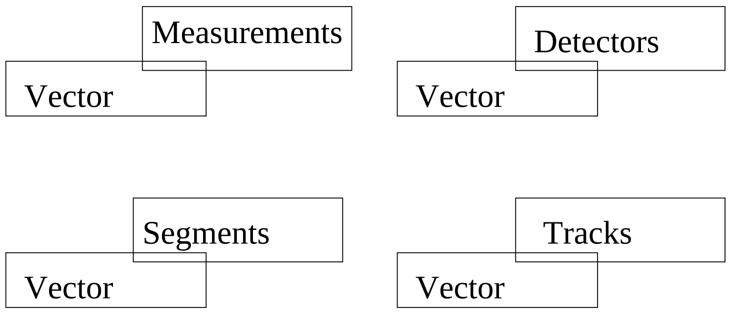

Figure 12: Concrete vectors based on STL

Fig. 12 shows the possible vectors that are used throughout this example.

3.4 Implementation

In the above we have done a top-down approach. Using the class diagrams and scenarios a more detailed design and implementation is done in this section. The lowest level classes are started first and other classes are built on top, finally ending with a program in which tracks are recognised. The final code is shown in appendix B.

cylpoint, measur In contrast to the SiBT example, a cylindrical coordinate system is needed. A coordinate is defined by the radius, angle and cylinder axis. The class *cylpoint* has been defined for that purpose. A measurement is not associated with a point only but occupies a volume in space. The size of this volume is given by the individual sizes in the three coordinates. The *measur* class is derived from the *cylpoint* class. Another possibility for defining *measur* is the definition of an attribute of class *cylpoint*. This not used as a too complex naming scheme is the consequence.

```
class cylpoint{
// point in cylindrical coordinate system
public:
float   r;              // r coordinate
float   phi;            // phi angle coordinate
float   s;              // cylinder axis coordinate
};

class measur: public cylpoint{
// measurement point in cylindrical coordinate system
public:
float      dr,ds,dphi;  // measurement uncertainties in above
};
```

The vector of *measur* is used to show some vector syntax.

Vector The list of *measur* is introduced to show how vectors are used in this example. An ordered list is required which means that the ordering needs to be defined. For the purpose of this example measurements are ordered according to their ϕ value. The *equal* and *lessthan* operators are defined for the *measur* class. Once they are defined, ordered vectors can be defined.

```
// Define == on measur objects
bool operator==(const measur&  m1, const measur& m2)
{ return m1.phi == m2.phi;}

// Define < on measur objects
bool operator<(const measur&  m1, const measur& m2)
{ return m1.phi < m2.phi;}
```

Below, the object *points* is defined as a vector of *measur*. The iterator *m* is used to access elements in the vector. In the example, *m* is started by initialising it with the begin of points. For the next element in the vector it is sufficient to state (++m) *m++*. This is done at the end of the for loop. The for loop is ended when *m* is equal to value returned by *points.end()*. Remark that *points.end()* is a pointer to the last element plus 1. The operation *m++* can be executed on *m* when it has passed the value *points.end()* but leads to invalid iterators that point to nowhere.

```
vector<measur>  points;
vector<measur>::iterator m;
for (m= points.begin(); m != points.end(); ++m){
        cout << "point with radius: " << m->r;
        cout <<"  and angle: " << m->phi << "\n";}
```

In the for loop, the phi- and r- attributes of all elements in *points* are printed in the order they are stored. To be sure that elements in points are ordered according to phi, the order function needs to be invoked, as shown later.

layerset, event The class *event* consists of an array of *nlayer* layersets where the values returned from the equipment are stored. For the example some possible values are stored (see Appendix B). The large difference with SiBT, where the number of strips is known, is that the number of measurements per detector are not known beforehand. This has some consequences for the design of the *detector* class, as shown later.

```
const int nlayer = 8;     // number of layers in installation
class event{
layerset        set[nlayer];    // 1 event: nlayer layersets
public:
        event(){ initialise  }
layerset layer(int pl){ return set[pl];}
};
```

The rest of the event and layerset class is self explanatory (see Appendix B). However, the structure of the event data has a large consequence for the whole design, as shown later as well.

segment The segment class uses the same line fitting method as used fro SiBT. A segment has the attribute *points* that represents the measurement points associated with this segment. Measurement points are added piecemeal with the method *enter*. The choice of adding them one by one is suggested by the assumption made on the detector and the event layout (i.e. variable number of measurements per detector and event structiure not known to detector). In *enter* the new measurement is added to *points* with the *push_back* method delivered by the STL library.

The most complex method is *valid*. It returns True when a straight line can be fitted through the points projected to the plane perpendicular to the s-axis. In contrast to SiBT, the line is not entered as a special class because the line fit is only needed for the segment determination and not for other purposes. If other fit algorithms are needed, the whole *valid* method needs to be replaced. It can be envisaged to inherit different segments from the base class segment differentiated according to the type of fit that is applied. Examples are helix-segment or circle-segment (not done here).

A segment enters the layer at the inside and exits at the outside. An addition to the SiBT method *valid* is added here. The fitted line is intersected with the layer border. The intersection points *entry* and *exit* are calculated and stored in the segment. These two points are used to determine if two segments can belong to one track.

```
const float     linchi  = 10.;
```

```
class segment{
public:
vector<measur>  points;
cylpoint        entry;
cylpoint        exit;

void    enter(measur* m){ points.push_back(*m);}
bool    valid(float r,float phi) {
float   ts_tg ,s_off, chi;     // line fitting variables
float   of, tg;                // variables for detector surface
float   s,t;                   // coordinates after transformations
int     nms =0;                // number of measurements
float   S_s = 0., S_ss = 0., S_t = 0., S_tt = 0., S_st = 0.;

        if (points.begin() != points.end()){
                vector<measur>::iterator m;
                for (m= points.begin(); m != points.end(); ++m){
                        nms = nms + 1;          //  number of entries
                        s = m->r*cos(m->phi);
                        t = m->r*sin(m->phi);
                        S_s = S_s + s;          //sum over s
                        S_t = S_t + t;          //sum over t
                        S_ss = S_ss+s*s;        //sum over s*s
                        S_st = S_st+t*s;        //sum over t*s
                        S_tt = S_tt+t*t;}       //sum over t*t
                }
        if (nms < 3){ return false;}    // at least three coordinates required
        float det = float(nms)*S_ss - S_s*S_s ;
        if (det <= 0) { return false;}
        s_off = (S_ss*S_t - S_s*S_st)/det; ts_tg = (nms*S_st - S_s*S_t)/det;
        chi = S_tt-s_off*S_t - ts_tg*S_st;

        // calculate entry and exit points of line
        entry.s = 0; exit.s = 0;                // default values
        tg = (sin(phi)-sin(phi+2.*dphi))/(cos(phi)-cos(phi+2.*dphi));
        // inner layer, entry point
        of = r*sin(phi)-tg*r*cos(phi);
        t = tg*s+of; s = (s_off-of)/(tg-ts_tg);
        entry.r = sqrt(s*s + t*t); entry.phi = atan2(t,s);

        // outer layer exit point
        r = r+detcdr+rspace; of = r*sin(phi)-tg*r*cos(phi);
        s = (s_off-of)/(tg-ts_tg); t = tg*s+of;
        exit.r = sqrt(s*s + t*t); exit.phi = atan2(t,s);

        return abs(chi) < linchi;
        }
};
```

One can test whether a vector is empty by comparing the begin and end iterator. When they are unequal, the vector is filled. This test at the start of *enter* is not really necessary because the for loop over *m* terminates directly when the vector is empty. In that case *nms* is not increased and remains smaller than 3.

The ordering of the segments is important for the track recognition algorithm. By looking at Appendix B it can be seen that for the segment also the *equal* $(==)$ and the *lessthan* $(<)$ operators are defined with as key the *phi* value of the *exit* attribute.

track A track consists of a vector of segments. A point to make is that the ordering for segments is already defined according to the value of the exit point. Segments in a track are preferably ordered according to their distance, *r*, from the intstallation center. This ordering of segments within a track is realized by entering the segments in a given order and not applying the STL order method defined on the angle.

The track has an entry and exit point. A track enters from the inner layer and exits at the outer layer. The first entered segment is found in the outer layer and defines the exit point of the track. The last entered segment is from the innermost layer and determines the entry point of the track. This is reflected in the code of *enter*.

It is assumed that segments from the outer layer are entered to the track first, followed by segments from inner layers according the decreasing segment entry radius value (defined by *entry.r*). This assumption is a logic consequence of the chosen track finding algorithm.

```
class track{
vector<segment> trk;    // segments that constitute track
public:
cylpoint        entry;
cylpoint        exit;
void    enter(segment* s){
        entry = s->entry;
        if (trk.begin() == trk.end()){ exit = s->exit;}
        trk.push_back( *s);
        }
}
```

Ordering of tracks is defined on the *phi* value of their entry point. Remark that the segments are ordered according to the *phi* value of the exit point.

detector Detectors are assumed to be flat boxes that are defined by eight corner points: *pt[8]*. Two lists are defined in the detector: the list of measurement points and the list of segments composed of these measurement points. For simplicity it is assumed that a detector can produce only one segment composed of all measurements in the detector. *Detector* is the first class that needs a constructor. The detector class, like all former classes, does not need a destructor as no pointers to objects need to be initialised with *new* thus avoiding the destruction of these objects. All the objects inside vectors are deleted automatically by the destructor of the vector class.

The constructor defines a detector volume with the eight pt[8] coordinates calculated from the *r* and *phi* parameters of the constructor. One measurement at the time is added to the detector with the *add* method. The reason for this is explained when the tracklist is discussed. Before any measurements are added the *clean* method must be invoked to remove all former measurement points and associated segments.

Once all measurement points are added to the detector the *FindSegment* method determines whether the measurements belong to a segment. As already mentioned the segment finding algorithm is a very simple one. All measurements are taken from the vector of measurements and stored in a newly created vector called *seglst*. Remark that the same local object *sg* is used to construct the segment. This is allowed because *push_back* makes a copy of the item before it is stored into the vector. The *valid* method of the segment *sg* is invoked. Its parameters *pt[0].r* and *pt[0].phi* serve to calculate the entry and exit points of the segment. A valid segment is added to the segment list *seglst*. Only one segment is added in this case but the introduction of *seglst* leaves the possibility to add more segments in a later stage.

```
class detector{
public:
cylpoint        pt[8];   // detector defined by 8 points in space
vector<segment> seglst;  // list of segments
vector<measur>  measlst; // list of measurements

        detector() { };
        detector(float r, float phi){
        for (int i = 0 ; i< 4; i++) {pt[i].s = 0; pt[i+4].s=detcds;}
        pt[0].r = r; pt[1].r = r;
        pt[4].r = r; pt[5].r = r;
        pt[2].r = r+detcdr; pt[3].r = r+detcdr;
        pt[6].r = r+detcdr; pt[7].r = r+detcdr;
        pt[0].phi = phi - dphi; pt[4].phi = phi- dphi;
        pt[3].phi = phi - dphi; pt[7].phi = phi- dphi;
        pt[1].phi = phi + dphi; pt[5].phi = phi+ dphi;
        pt[2].phi = phi + dphi; pt[6].phi = phi+ dphi;
        }  // end of detector constructor
void    add(measur m){measlst.push_back( m);}
void clean(){
        // remove all segments and measurements
        seglst.erase(seglst.begin(),seglst.end());
        measlst.erase(measlst.begin(),measlst.end());
        } // end of clean
void    FindSegment(){
        if (measlst.begin() != measlst.end()){ // measurements present
                segment sg;
                vector<measur>::iterator m;
                for (m= measlst.begin(); m != measlst.end(); ++m){
                      sg.enter(m);} // all measurements in one segment
                if (sg.valid(pt[0].r,pt[0].phi)){seglst.push_back(sg);}
                                                // add sg to segment list
                } // end of if (measlst....
        } // end of FindSegment
};
```

Ordering of detectors is defined on the *phi* value of their *pt[0]* coordinate (see appendix B.

layer The layer contains a set of detectors. The constructor of the layer creates the detectors and stores them into the detector vector *dts*. The detectors in *dts* are ordered according to the ϕ value of one of their corner points (see Appendix B).

The design of the layer class *assign* method deviates from the ones suggested earlier. These deviations have an impact on the design of the *detector* and *tracklist* classes. The assumption is that for each layer, the event data may have a different structure. This means that each individual layer is made responsible for the assignment of data to the detector. The amount of data per detector varies. In the layerset structure it is assumed that the ϕ angle of the measurement point can be used to attribute a data-item to a detector. Hiding the layerset structure to the detector means that the layerset method *detect* is asked to return a pointer to the next measurement corresponding with a given ϕ range. The result of these design decisions is that detectors cannot decide whether the last measurement point is added to the detector. It is the layer that knows when the last measurement point is furnished to the detector. This means that the layer first must clean the data in the detector by invoking the *clean* method of the detector before invoking the *add* method of the detector for each individual measurement point. After the insertion of all measurement points into a given detector, the *FindSegment* method of the detector can be invoked.

```
class layer{
public:
vector<detector>        dts;
        layer(float ri) {
// create a layer with inner circle defined by ri
        for (int i=0; i < ndetct; i++){
                dts.push_back( * new detector(ri, dphi*i*2));}
        sort(dts.begin(), dts.end(),less<detector>() );
        }  //end of layer constructor
void    assign(layerset ls){
        float*  pm;
        measur  m;
        m.dr =0.; m.dphi = 0.; m.ds =0.;
        vector<detector>::iterator d;
        for (d= dts.begin(); d != dts.end(); ++d){
                d->clean();             // remove segments and measurements
                pm = ls.detect(d->pt[0].phi);
                while (pm != null){
                        m.phi = pm[0]; m.r = pm[1]; m.s = pm[2];
                        d->add(m);
                        pm = ls.detect(d->pt[0].phi);} // end of while
                d->FindSegment(); // construct segments
                } // end of for
        } // end of assign
};
```

In the discussion section 3.5 alternatives for the detector and event design are mentioned.

tracklist Like in the SiBT example, tracklist is the heart of the layers example. In this class the relation between the individual detectors and layers is made. The algorithm constructs tracks layer by layer starting at the outside and ending at the inside of the installation. The loop over all layers can be made inside tracklist or inside the invoking installation. To reduce the complexity of tracklist code, it is chosen to do one layer at the time inside tracklist. This is in contradiction with our original idea mentioned in section 3.3.1. A vector of tracks is maintained in the attribute *tklst*. This vector contains the zero tracks, the tracks under construction or the finished tracks after the treatment of the last layer. No destructor is needed. It is assumed that every time a new set of tracks is calculated a new tracklist object is created. Therefore, no *clean* method is needed to remove all former tracks from *tklst*.

Only one method *extend* with parameter the current layer is defined. The local object *cr_ent* maintains the entry point of the track under consideration. When no tracks are available, its value is set to a value larger than 2π. The method *extend* first copies the segments of all detectors in the layer into the local object *sglst* and orders the list.

In the next stage, segments are added to tracks when the entry point of the track is sufficiently near the exit point of the segment. When no appropriate track can be found for the segment a new track, composed of this one segment, is inserted into *tklst*.

A loop is done over all segments. As long as the exit ϕ value of the segment is less than *cr_ent*, the segment is entered as a new track. A local object *tk* is created, the segment *s* is entered and *tk* is pushed on *tklst*. The scope of *tk* is left at the end of the while loop and *tk* is automatically destroyed. This does not affect the track recently added to *tklst* as a copy of *tk* is inserted.

When the exit ϕ value is close enough to *cr_ent*, the segment is added to the current track. When the *cr_ent* value is less than the exit ϕ value of the current segment, the next track is considered until *cr_ent* is larger than *s*$-$ *>exit.phi-phidev*. Under the assumption that only one segment is found per detector and the detectors are sufficiently widely spaced, this code works well. However, if segments are close, a track may end up with two or more segments of one layer. This is in conflict with the track definition.

```
class tracklist{
vector<track>   tklst;  // list of tracks
public:
void    extend  (layer* ly){
        float   cr_ent;                 // entry point of current track
        vector<track>::iterator t = tklst.begin();
        if (t == tklst.end()){ cr_ent = 2*M_PI+dphi;}  // empty track list
        else { cr_ent = t->entry.phi;}

        // store all segments of all detectors in layer ly into sglst
        vector<segment>  sglst;         // list of segments in layer ly
        vector<detector>::iterator d;
        for (d= ly->dts.begin(); d != ly->dts.end(); ++d){
             copy(d->seglst.begin(), d->seglst.end(), back_inserter(sglst));}
        sort(sglst.begin(),sglst.end());        // sorted segments of layer ly

        // start extension of creation of tracks
        vector<segment>::iterator s= sglst.begin();
        while (s != sglst.end()){
                while ((s->exit.phi < cr_ent - phi_dev) & (s != sglst.end())){
                        track tk;               // new track
                        tk.enter( s); s++;      // track with one segment
                        tklst.push_back( tk);}  // add new track to list
                if (s != sglst.end()){
                        while ( cr_ent < s->exit.phi-phi_dev){
                            t ++;                       // next track
                            if (t == tklst.end()){ cr_ent = 2*M_PI+dphi;}
                            else { cr_ent = t->entry.phi;}
                            } // end while ...
                        if (abs(s->exit.phi - cr_ent) < phi_dev) {
                            t->enter( s); s++;}         // extend track
                        } // end of if (s != ....
                } // end while (s!=sglst ...
        sort(tklst.begin(),tklst.end());
        } //end extend
};
```

The STL convention to copy items and then store them into the vector has a profound influence on the creation and deletion of objects. Suppose no copies were made by STL. Every time an object of class *tracklist* is removed, all segments and the measurements in the segments are removed as well. Consequences are many fold: (1) possibly all segments in the detectors are removed and a subset of all measurements are removed in the detectors, (2) to prevent a double removal of some segments and or measurement points, measurements must be removed from the measurement list in the detector before being put in a segment (3) same thing is true for a segment and (4) each time a segment is moved to a track, it must be removed from the segment list in the corresponding detector.

When large elements are manipulated in the vector, it is advisable to store pointers to the elements into the vector. This will increase the performance as much unwanted copying is suppressed. However, a lot of thought must be put in the removal and creation of elements, as shown above.

installation The installation contains an array of layers and the tracks associated with an event. The constructor defines the layers by invoking the layer constructors *nlayer* times with an appropriate radius value.

The method *GetTracks* determines the tracks associated with a given event passed as parameter. The former tracks are removed by deleting the object pointed at by *tklist*. For all layers the measurements are assigned to the detectors after which the tracks in *tklist* are extended with the segments of this layer.

```
class installation{
layer*          ly[nlayer];     // nlayer layers in installation
tracklist*      tklist;         // list of tracks
public:
        installation(){
        tklist = new tracklist();
        for (int i = 0; i <nlayer; i++) {
                ly[i]= new layer(i*(router-rinner)/nlayer + rinner); }
        }
tracklist* GetTracks(event ev){
        delete tklist;                  // remove old tracks
        tklist = new tracklist();       // new empty track list

        // find tracks by going from outer layer to inner layer
        for (int i = nlayer-1; i>-1; i--) {
                ly[i]->assign(ev.layer(i));  // measurements into layer i
                tklist->extend(ly[i]);       // tracks include layer i segments
                }
        // measurements are assigned to layers
        return tklist;
        }
};
```

3.5 Evaluation

Having done the detector installation, it is clear that the track returned by the SiBTOO example is not correct. Too much information is still included in the track that is composed of two projections. SiBTOO should return a segment. each measurement in a segment should conform with a cluster in SiBTOO converted to cylindrical coordinates. The y-value of the horizontal clusters and the x-value of the vertical clusters need to be determined.

An adequate point plus deviations need to be determined. Knowing that several type of detectors are possible, a base class detector with virtual methods needs to be defined. Other detector types can then be derived from detector with inheritance. A good example is the derivation of SiBTOO (see Fig. 13).

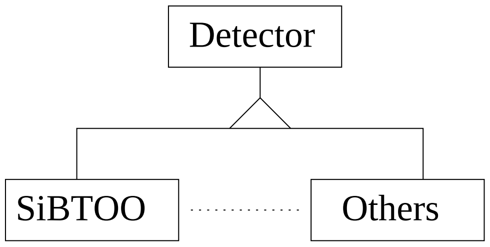

Figure 13: Specialisation of detectors

The use of the STL library has introduced the possibility to return the complete vector of segments from the detector. The TrackList class exploits this by copying all segment lists of all detectors in a layer in one go to another ordered segment list. Therefore, the *NextSegment* and *FirstSegment* methods defined in section 3.3 have not been used.

The structure of the event has a large influence on the design. When it can be assumed that for every event the number of data per detector is constant, the event data can be stored in a large array with pointers to the array portion that is relevant for a given detector. Every detector knows where its data area starts and how much data it is supposed to treat. Determining tracks can then be done in one go. The detector removes old results, reads in the data and constructs accompanying tracks. This is similar to what is done for SiBT but was not done in the layer example. Another assumption may be that the amount of data for a given detector can change from event to event. Then a structure needs to be added to the event. The question is whether this structure is known to the individual detectors. If the structure is unknown to the detectors (because it changes all the time) a design must be done like was done here with *clean*, *assign* and *FindSegment* methods. If the structure is stable but changes from layer to layer, then it is best to have each layer assign the values to the detectors.

The ”Combine Segments” scenario suggested in section 3.3 is different from the one that is implemented. The scenario for the implementation discussed in section 3.4 is shown in Fig. 14.

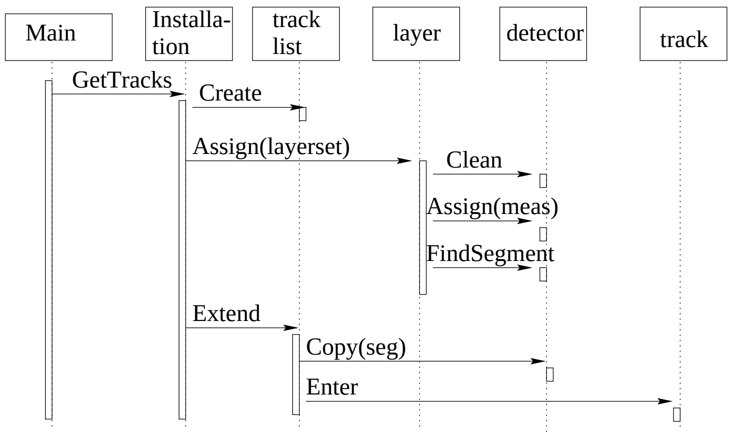

Figure 14: Implemented scenario for determining a list of tracks

The design of the layers has followed the same guidelines as for the SiBT example. However, a few things can be said about the layer design not covered by the SiBT design

- **Performance** should especially be considered in relation with the STL library. Another design step is needed to improve the performance. As suggested in the text, vectors of pointers to items instead of vectors of items can be used. Additionally, the copying of the segments of the detectors into a one vector can be avoided to diminish the copying overhead. This will render the code less comprehensible.

- **Information hiding** The layer structure is hidden from *Main*. The detector structure in a layer is hidden to the installation. The calculation of a track is hidden within the TrackList class. The fitting of a line through a segment is hidden inside the segment class. The measurement determination is hidden inside the vector. A point of debate is what part of the structure of the event should be hidden.

- **Layering** The abstractions needed are much clearer after this example. An installation returns tracks. An installation is composed of layers that are composed of detectors. A detector returns segments. A track is composed of segments such that per layer at most one segment occurs. Inside a detector measurements are done and measurement points are calculated. A segment is composed of measurement points.

3.6 Exercises

1. Change the event structure such that the *assign* of a detector is invoked with a pointer to the event data that concern the detector. The detector should be able to recognize the end of event or know the total amount of event data that concern it. The methods *clean* and *FindSegment* are suppressed.
2. Use inheritance to define points with Cylindrical coordinates, Cartesian coordinates and Polar coordinates. Define the constructors such that for a coordinate of a given type, the values in the two other coordinate systems are defined as well. Inherit Cylindrical, Cartesian and Polar measurements from the appropriate point classes.
3. The same as above, but first inherit measurement from point and then inherit the three types of measurements. Which of the two inheritance hierarchies is the more appropriate one?
4. Change SiBTOO such that it returns segments.
5. Instead of a vector of *meas*, use a vector of pointer to *meas*. Carefully consider all ceation and removals of *meas* objects. Do the same with vectors of segments.
6. Change the *insert* method in the *tracklist* class such that the copying of segments is not needed any more at the beginning.
7. Change the *insert* code such that a segment can be attributed to two tracks. This means that the track has to be copied every time another segment of the same layer is a candidate for addition to this same track.

4 Tracing example

This example builds on the layers example of section 3. It assumes the same simplified layer structure. Particles are traced through the detectors to determine their trajectory.

4.1 Tracing description

In this section the tracing of a particle trajectory is based on local decisions inside the detector. In the former example of section 3.1, tracks were determined on a layer basis. This is not done for the trajectory. A particle enters a detector at a given coordinate of a wall. It traces a trajectory until it hits the opposite or neighbouring wall of the entered detector. The intersection of the detector wall with the particle trajectory leaving the detector is called the exit point. At that point the particle enters the neighbouring detector. The exit and entry points of the particle passing through neighbouring detectors are assumed to be identical for simplicity reasons.

In Fig. 15 the additional structure required of the detectors is shown. For simplicity reasons only the plane perpendicular on the s-axis of the installation is considered. Each detector has a pointer to its upper, lower, former and forward detector. The outer and inner layer detectors have no upward or forward detector respectively.

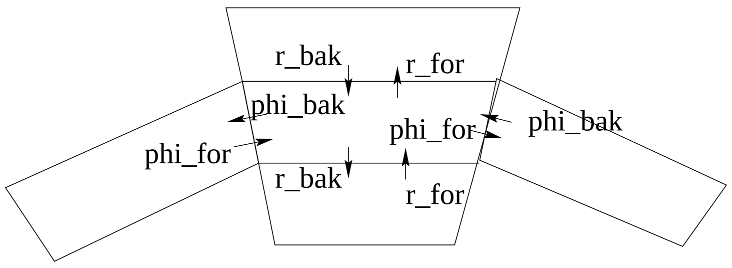

Figure 15: Diagram of detector neighbours

In Fig. 15 the phi_bak pointer points to the neighbour in the same layer with a smaller ϕ angle, the phi_for pointer points to the neighbour in the same layer with a larger ϕ angle, r_for and r_bak point to neighbours with a larger respectively smaller layer radius.

4.2 Analysis

The classes associated with tracing, excluding the ones defined already in Fig. 9, are shown in Fig. 16. A neutral particle traces one straight trajectory through a detector from entry point to exit point. A charged particle traces one curved trajectory through a detector. A particle does not interact with the detectors. A detector has 3-4 neighbours. A particle traces as many trajectories as it passes detectors. There is a relation between the exit point of one trajectory with the entry point of the consecutive trajectory.

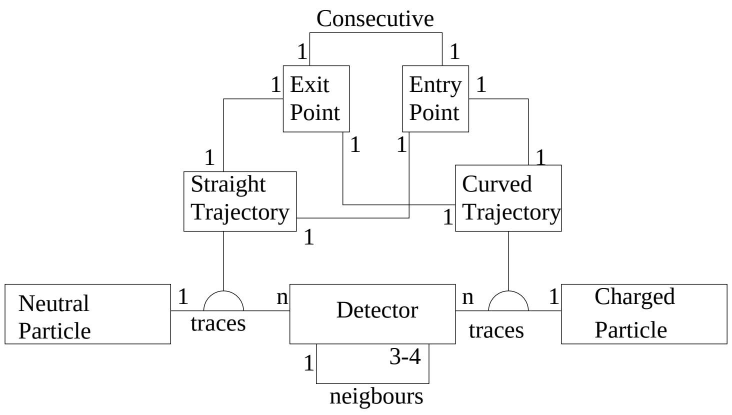

Figure 16: Class diagram based on layers analysis

The following use case is considered:

1. **Particle tracing** A particle with a certain mass passes through a magnetic field that pervades the whole installation. Charged particles trace curved trajectories and neutral particles trace straight lines. A particle

enters the installation at a detector in the innermost layer. It traces a trajectory until it passes the detector wall. At the exit point it leaves the detector and enters the neighbouring detector at its entry point. The entry point of the neighbour is equal to the exit point. The particle continues its way from the entry point of the neighbouring detector until the exit point of the neighbouring detector. This continues until the particle leaves a detector at the outer layer without entering a neighbour

4.3 Design

The scenario diagrams are used to show where the responsibilities are put for the different classes identified above. A first design decision is to hide the structure of the installation from *Main* that provides the entry point of a particle.

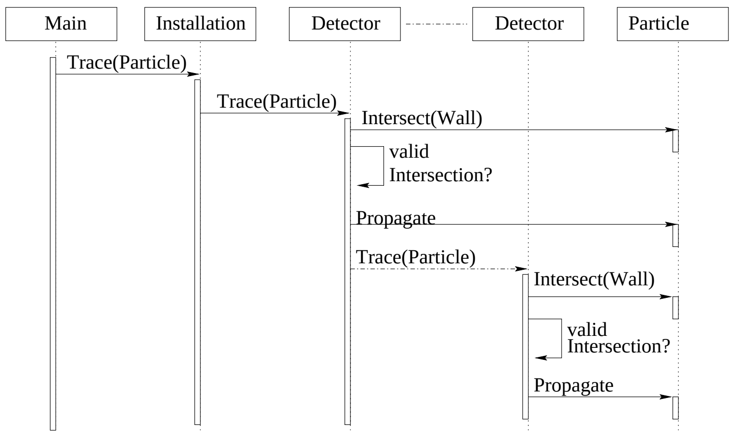

Figure 17: Scenario diagram for particle tracing

Particle tracing The central idea is to decide inside each detector which neighbouring detector is entered when the particle leaves the detector under consideration. *Main* provides a particle with some initial values that enters the first detector. Dependent on the mass, charge, momentum and orientation of the particle, the particle traces a curve. This curve is particle dependent and therefore the detector asks the particle to determine the intersection of the particle trace with one of the four walls. The detector determines whether the calculated intersection point lies within the detector boundaries. If not, the intersection with the next wall is calculated. When the exit point is calculated, the particle is propagated to the exit point. At the exit point, the neighbour can be found. The neighbour is asked to continue the tracing of the particle. When the particle exits the installation, the coordinates and kinetic parameters of the particle at the exit point of the installation are returned to Main.

4.3.1 Tentative class definitions

Additions to the layers example are mentioned.

detector The *Detector* class is extended with local attributes that are references to the detector's neighbour. The B-field is a local attribute that represents the magnetic field in this detector. The method *Trace* starts the tracing of a particle in the detector.

Particle A *Particle* class is introduced. It has coordinates and a direction. The private attributes charge, mass and momentum determine the curvature of the particle. When the exit point of the particle's trajectory is known, the particle's coordinates are updated with the method *Propagate* to continue the tracing in the next detector. Because charged particles have a different trajectory as neutral particles, inheritance is used to differentiate the *Intersect* methods. The charged particle's *Intersect* intersects the specified line with a curved trajectory, while the neutral particle intersects it with a straight trajectory.

Installation The method *Trace* is added to trace a particle through the installation. The particle starts at a detector in the inner layer and terminates at the outer layer. After invocation the particle's coordinate is the exit point coordinate.

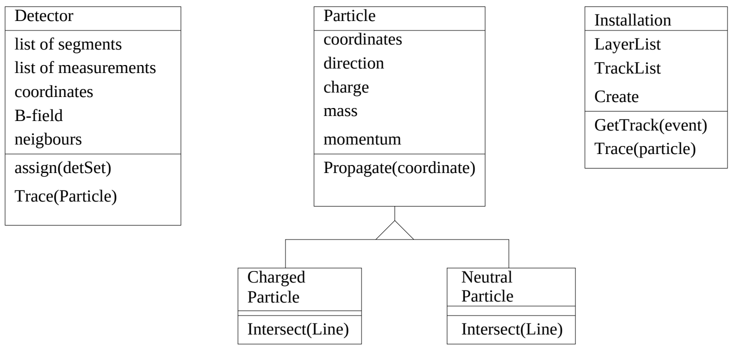

Figure 18: Association realizations

4.4 Implementation

In the above, we have discussed an extension to the layers example. In this section the same approach is followed. First the particle tracing is considered followed by the geometry and initialisation of the detectors. The final total code is shown in appendix C and is an extension to the code in appendix B.

vect, particle The particle has a direction and a coordinate. In addition to *cylpoint*, the class *vect* is introduced that is an extension of *cylpoint*. The *angle* of direction in the plane perpendicular to the s-axis is added to the coordinate. The class *vect* inherits from the class *cylpoint*.

```
class vect: public cylpoint{
// point with direction in plane perpendicular to s
public:
float   angle;          // orientation in plane perpendicular to s
};
```

The *particle* class inherits from the *vect* class. This follows from the observation that the location and direction of the particle are intrinsic properties of the particle. Another more practical argument to use inheritance is the simplified code that is obtained to access the *vect* values of the particle. The method *Propagate* advances the particle to the coordinates and direction specified in the parameter *vc*. The method *Intersect* is virtual as the propagation of a particle trough a medium and magnetic field depends on the properties of the particle. In the particle class it is assumed that a particle needs to be described by its mass and momentum. The methods *Intersect* and *Propagate* are two separate methods as a particle can intersect with the four walls of a detector. First, it needs to be determined where the particle trajectory intersects with the detector wall. Intersection with the wall does not necessarily take place inside the detector volume. A correct intersection takes place within the detector volume. Consequently the intersection of the particle with one given wall can be chosen by invoking *Propagate* with the chosen intersection point as parameter.

```
class particle: public vect{
// physical characteristics of particle
public:
float           mass;           // particle’s mass
float           p;              // particle’s momentum
void    Propagate(vect vc) {
                phi = vc.phi; r = vc.r; s = vc.s; angle = vc.angle;}
virtual vect Intersect(cylpoint cc1, cylpoint cc2, float B) {
        vect vc;
        vc.phi = 4*M_PI; vc.angle =0;
        vc.r = 2*router; vc.s =0;
        return vc; }
};
```

neutral particle The neutral particle is clearly a refinement of a particle. The class *neutral_particle* inherits from particle. A concrete *Intersect* method is defined for this class. The particle is assumed to move along a straight trajectory. The particle does not interact with the surrounding material. The wall line is defined by the two parameters *cc1* and *cc2* that represent two corners of a detector. Some assumptions are made in this method: (1) the particle trajectory intersects the wall line within a reasonable distance (no parallel lines) and (2) the local variables *tg1* and *tg2* never evaluate to ∞. For production code these assumption cannot be made and necessitate additional coding.

```
class neutral_particle: public particle{
// neutral particle trajectory is straight line
vect    Intersect(cylpoint cc1, cylpoint cc2, float B) {
        double tg1 = tan( angle);
        double of1 = r*sin(phi)-r*cos(phi)*tg1;

        double tg2 = (cc1.r*sin(cc1.phi) - cc2.r*sin(cc2.phi)) /
                     (cc1.r*cos(cc1.phi) - cc2.r*cos(cc2.phi));
        double of2 = cc1.r*sin(cc1.phi)-cc1.r*cos(cc1.phi)*tg2;

        double x = (of2-of1)/(tg1-tg2);
        double y = tg1*x + of1;
        vect vc;
        vc.s = 0;
        vc.phi = atan2(y,x);
        vc.r = sqrt(x*x + y*y);
        vc.angle = angle;
        return vc;
        }
};
```

charged particle The class *charged_particle* inherits from particle. The attribute *charge* represents the charge of the particle. It can have two values: -1 or +1. A concrete *Intersect* method is defined for this class. The particle is assumed to move along a circle trajectory. The particle does not interact with the surrounding material. Like for the neutral particle case, the wall line is defined by the two parameters *cc1* and *cc2*. A simple quadratic equation determines 0, 1 or 2 intersection points. When no intersection point exists a intersection point value is returned outside the detector volume. When one of the two possible intersection points intersects the wall outside the detector volume the second intersection point is returned. Of course, this is not necessarily a point within the detector volume. A check on the correctness of the returned intersection point is done within the detector. Again, an assumption is made on the trajectory and wall line parameters: the variable *tg* should never evaluate to ∞.

```
class charged_particle: public particle{
// charged particle trajectory is circle
public:
int     charge;                 // particle’s charge (+1 or -1)
vect    Intersect(cylpoint cc1, cylpoint cc2, float B) {
        double x1 = cc1.r*cos(cc1.phi);
        double x2 = cc2.r*cos(cc2.phi);

        // calculate centre coordinates of circle
        double Radius = curv*p/B;
        float xm = Radius*cos(angle+charge*M_PI/2.) + r*cos(phi);
        float ym = Radius*sin(angle+charge*M_PI/2.) + r*sin(phi);
        // calculate intersection point of line with circle
        double tg = (cc1.r*sin(cc1.phi) - cc2.r*sin(cc2.phi)) /
                    (cc1.r*cos(cc1.phi) - cc2.r*cos(cc2.phi));
        double of = cc1.r*sin(cc1.phi)-cc1.r*cos(cc1.phi)*tg;
        double a = 1+tg*tg;
        double b = 2.*tg*(of -ym)-2*xm;
        double c = xm*xm+(of-ym)*(of-ym)-Radius*Radius;
        double det = b*b - 4*a*c;
        vect vc; vc.s = 0;
        if (det < 0) {          // no intersection point
                vc.phi = 4*M_PI; vc.r = 2*router;
                return vc;}
        double x = (-b + sqrt(det))/(2*a);
        if (x < min(x1,x2) | x > max(x1,x2)) { // no intersection
                x = (-b - sqrt(det))/(2*a);}
        double y = tg*x + of;
        vc.s = 0; vc.phi = atan2(y,x); vc.r = sqrt(x*x + y*y);
        vc.angle = atan2(ym-y,xm-x)-charge*M_PI/2.;
        return vc;
        }
};
```

detector::trace Within a detector, a particle is traced from entry wall to exit wall. The method *trace* traces a particle defined by the *ptk* parameter. Origin and direction of trace are defined by *ptk*. It is assumed that a particle intersects with the lines defined by corner pairs (pt[2],pt[3]), (pt[1],pt[2]) and (pt[0],pt[3]). The intersection point with the line defined by (pt[2],pt[3]) is done first under the assumption that the outer wall is the most likely wall to be intersected. Notice that the parameter *ptk* can be of class *charged_particle* or of class *neutral_particle*. Once an intersection point has been found (the algorithm assumes the existence of such a point), the particle is propagated to this intersection point. By setting the detector pointer *dtp* equal to one of the pointers *r_fw*, *ph_bk* or *ph_fw*, *dtp* points to the appropriate neighbour detector that shares the found intersected wall. Accordingly, the detector invokes the *trace* method of the neighbour detector.

```
virtual void trace(particle& ptk) {
        detector* dtp;
        vect cc = ptk.Intersect(pt[2],pt[3], B);
        dtp = r_fw;
        if (cc.phi > pt[2].phi | cc.phi < pt[3].phi){
                cc = ptk.Intersect(pt[0],pt[3], B);
                dtp = ph_bk;
                if (cc.r < pt[0].r | cc.r > pt[3].r){
                        cc = ptk.Intersect(pt[1],pt[2], B);
                        dtp = ph_fw;}
                } // end if(cc.phi < pt[2].phi.....
        ptk.Propagate(cc);              // advance particle
        dtp->trace(ptk);                // next detector
        } // end of trace
```

dummy detector To terminate the tracing, the class *dummy_detector* is introduced. The *trace* method of this class returns without doing anything and consequently terminates the tracing through all neighbouring detectors.

```
class dummy_detector: public detector{
public:
void trace(particle& ptk) {   }
};
```

layer connections The wanted connection of neighbouring detectors is done in the layer. Two options are created with as many *connect* methods (overloading of *connect*).

For the method *connect(layer* ly)* it is assumed that the *ly* parameter points to the neighbouring outer layer. Two vectors of detectors are used: *dts* the vector of the executing layer and *ly->dts* the vector of the outer neighbouring layer. It is assumed (by construction) that the same number of detectors are found in both layers and that the nth detector of the one layer is the upper neighbour of the nth detector in the other layer. Connecting detectors within a layer and between two neighbouring layers is then rather straightforward.

For the method *connect(detector* dup)* it is assumed that the *dup* parameter points to the neighbouring outer detector. For the outer layer this is a pointer to a *dummy_detector*. The result is that when the trace is invoked of the outer neighbour of a detector in an outer layer, the tracing stops.

```
void    connect(layer* ly){
        vector<detector>::iterator dup = ly->dts.begin();
        vector<detector>::iterator d,dbk = dts.begin();
        for (d= dts.begin(); d != dts.end(); ++d){
                dbk->ph_fw = d; d->ph_bk = dbk;
                d->r_fw = dup; dup->r_bk = d; ++dup;
                dbk = d;}
        // dbk points to last detector
        d = dts.begin(); dbk->ph_fw = d; d->ph_bk =dbk;
        } // end of connect(layer)
void    connect(detector* dup){
        vector<detector>::iterator d,dbk = dts.begin();
        for (d= dts.begin(); d != dts.end(); ++d){
                dbk->ph_fw = d; d->ph_bk = dbk;
                d->r_fw = dup;
                dbk = d;}
        // dbk points to last detector
        d = dts.begin(); dbk->ph_fw = d; d->ph_bk =dbk;
        } // end of connect(detector)
```

installation::Trace The tracing of a particle is started by invoking the *Trace* method of the installation for a suitably chosen particle. It is assumed that such a particle is chosen to be on the inner wall of a detector in the inner layer. The detector with the right angle is found by invoking the STL *find* function that acts on the vector of detectors till it finds a detector that is equal to *key*. The equal operator on detectors is defined to operate on the *phi* attribute of the pt[0] corner of the detector. The detector *key* is defined to be the detector with the wanted pt[0].phi value. The consequence is that after invocation of find, the *where* variable points to the specified detector. When no such detector is found, where points to dts.end().

Invoking the *trace* method of the found detector starts the tracing. Tracing ends when the particle enters a dummy_detector on leaving the outer layer.

```
void    Trace(particle& ptk){
        float phip = ptk.phi+dphi;
        phip = int(phip/(2*dphi))*2*dphi;
        detector key = detector(rinner,phip);
        vector<detector>::iterator where = find(ly[0]->dts.begin(),
                                ly[0]->dts.end(), key);
        if (where != ly[0]->dts.end()) {where->trace(ptk);}
        else { cout << " COULD NOT FIND DETECTOR for Tracing \n" << flush;}
        }
```

Invocation from Main The invocation from *Main* is shown here. In *trlst->tklst* a set of tracks is returned by *GetTracks*. The inner intersection point of each track is used as starting point for the particle tracing. Particles with different momenta and masses can be traced to determine whether their exit points are in accordance with the particle trajectory exit point contained in *cptk* after termination of *Trace*.

```
        tracklist*  trlst = instal.GetTracks(ev);
        vector<track>*   vt = &(trlst->tklst);
        vector<track>::iterator t;

        charged_particle        cptk;
        cptk.charge = -1;
        cptk.mass = 1;
        cptk.p = 1;
        for (t = vt->begin(); t != vt->end(); t++){
                cptk.phi = t->entry.phi;
                cptk.r = t->entry.r;
                cptk.s = t->entry.s;
                dx = t->exit.r*cos(t->exit.phi)-cptk.r*cos(cptk.phi);
```

```
dy = t->exit.r*sin(t->exit.phi)-cptk.r*sin(cptk.phi);
cptk.angle = atan2(dy,dx);
instal.Trace(cptk);}
```

4.5 Evaluation

The layer design was such that the extension for particle tracing could be added without problems. No modifications to the general structure were needed. A problematic area is the design of the particles class in relation with the detector class. The trajectory of the particle is determined by the physical properties of the particle and the material of the detector. Assuming that there are more particle variants than detector materials, the detector characteristics are taken as parameter for the particle trace trajectory. This way particles can be added for the known set of detector textures.

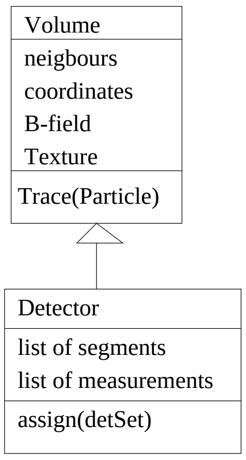

Figure 19: Volume and detector class

In the above, the empty space between detectors was neglected. In Fig. 15 it is assumed that detectors touch each other. This is not the case. Between a layer of detectors, there is a layer of air of a given thickness represented by the constant *rspace* in the code. In Fig. 19, a more appropriate design is presented. A volume is defined by its corner points. A particle is traced through a volume with a certain magnetic field and of a given texture. A detector inherits from the volume and contains a number of measurements and segments. The installation consists of layers of detectors separated by a layer of air volumes.

Some points need special attention:

- **Coordinate system** Maintaining a cylindrical coordinate system proves to be rather calculation intensive. Using cylindrical coordinates only to define the layers and detectors is sufficient. A Cartesian coordinate system looks better in all other cases.
- **Information hiding** The *installation* class and the *layer* class do not know about the propagation of a particle. A particle can pass from a detector in one layer to a detector of another or same layer without awkward case sensitive statements.
- **Performance** Performance can be improved by looking at the chosen coordinate system and chosen tracing algorithm. This can be done without any modification to the total structure.

4.6 Exercises

1. Store the detector coordinates in a Cartesian coordinate system. What other classes need to be modified.
2. Modify the *Intersect* methods such that they work under any circumstance without overflow problems.
3. Change the finding of the wanted detector in the *Trace* method in the *installation* class:
   - by doing a loop over all detectors and testing their corner pt[0] angle,
   - by using *find* and specifying another equal operator.
4. Add the volume class as suggested above and change the installation to include layers of air.

Acknowledgements

Martti Pimia and Werner Jank encouraged me to prepare these examples. I am very grateful to Veikko Karimaki, Aatos Heikkinen and Jukka Klem who explained SiBT to me. Ian Willers and Veronique Lefebure have commented on first manuscripts and suggested many improvements. Comments by Matthias Schroder, David Francis, Teddy Todorov and Vincenzo Innocente have contributed to a better presentation.

References

[Boo94] G. Booch. *Object Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Co, 1994.

[ea96] G.L. Bencze et al. An Accurate Telescope for Beam Position Monitoring and Spatial Resolution Studies. *Nuclear Instruments and Methods*, A(368):283–287, 1996.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[MS96] D.R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[Str97] B. Stroustrup. *The C++ programming Language*. Addison-Wesley, third edition, 1997.

A SiBTOO code

```
#include <stdlib.h>
#include <iostream.h>
#include <math.h>

// standard definition of NULL and bool
const int null = 0;

#define bool int
#define true 1
#define false 0


//definition of contants for SiBT geometry
//

const int       nplane = 8;                     // number of planes in SiBT
const float     sbt_ds = 200.;                  // length of SiBT over s-axis
const float     sbt_dx = 40.;                   // width of SiBT over x-axis
const float     sbt_dy = 38.;                   // height of SiBT over y-axis
const float     length = .003;                  // length of strip over s-axis
const int       nstrip = 1024;                   // number of strips
const float     xwidth = .9*sbt_dx/nstrip;      // width of strip in x direction
const float     ywidth = .9*sbt_dy/nstrip;      // width of strip in y direction

const int       hitquot = 3;                    // hit is represented by val/noise > hitquot
const float     linchi  = 10.;                  // chi of accepted line fit

// definition of classes
//

template<class T> class list{
// parametrized class List
// singly linked list of items
// initialized with zero items, with head and cur initialized to null
protected:
        class Link{
        public:
                Link*  next;
                 T*     val;
                Link(Link* n, T* v) {next = n; val =v; } ;
        };
        Link*   head;   // head of singly linked list
        Link*   cur;    // last selected item
public:
        int     number; // number of items in list
        list() {head = null; cur=null; number=0; };
        ~list(){
        while ( head != null){
                cur = head;
                delete head->val;
                head = head->next;
                delete cur;}
        };
void    enter(T* item) {
        Link* temp = head;
        if (item != null){
                head = new Link( temp, item);
                number ++;}
        };
T*      first() {cur=head;
```

```
        if (cur != null) {return (cur->val);}
        else { return null;}
        };
T*      next() {
        if (cur != null){
                cur = cur->next;
                if (cur != null){return (cur->val);}
                else {return null;}}
        else {return null;}
        };
T*      current() {
                if (cur != null){ return (cur->val);}
                else {return null;}
         };
bool    sequel(){
        if (cur == null){ return false;}
        else {return cur->next != null;}
        };
void    remove(T* item) {
        Link**    temp = &head;  // temp contains the location of the pointer to the item
                                // temp is initialized with the location of head
        for (Link** temp = &head; *temp == null; temp = &((*temp)->next)){
                if ((*temp)->val == item){
                        Link*  rm = *temp;
                        delete item;
                        delete rm;
                        temp = &((*temp)->next);
                        if (cur == *temp) cur = null;}  // cur points to removed item,
                                                        // so cur is set to null
                }
        };
#ifdef DEBUG
void    print(){
        cout << " list with " << number <<" entries \n" << flush;
        int k =0;
        Link* temp = head;
        while (temp != null){
                if ( cur == temp) { cout << "current ";}
                else { cout << "        ";};
                cout << " item "<< ++k << ":   " << flush;
                temp->val->print();
                temp = temp->next;}
        };
#endif

};

class coord{
public:
float   x,y,z;
};

class channelset{
int  items[50];
int  as;
public:
        channelset(){
                as = 0;
                for( int i =0; i<50; ++i) {items[i] = 0;}
        }

int     strip(int pl, int st) {
                for (int i = 0; i<as ; i = i+3){
```

```
                if (items[i] == pl){
                        return items[i+st];} // hit of channel pl found
        }
        return 1;}                      // no hit found
                                        // (minimum noise of 1 to prevent div by zero

void    assign( int strip, int value, int noise){
        items[as] = strip; as++;
        items[as] = value; as++;
        items[as] = noise; as++;}
#ifdef DEBUG
void    print(){
        int i=0;
        while (i < as){
                cout << "          strip " << items[i] <<";  value " <<
items[i+1] <<";   noise " << items[i+2] << "\n" << flush;
                i = i+3;}
        }
#endif
};


class event{
channelset      set[8]; // 1 event: 8 channel sets for 8 planes
public:
        event(){
        for (int pl = 0; pl <8 ; ++pl){

// if (pl != 2) {
                set[pl].assign(40+abs(pl-4),20,4);
                set[pl].assign(41+abs(pl-4),21,5);
                set[pl].assign(42+abs(pl-4),35,6);
//              set[pl].assign(254+abs(pl-4),59,8);
                set[pl].assign(255+abs(pl-4),63,9);}
//           }
/*
if (pl != 2){
                set[pl].assign(40,20,4);
                set[pl].assign(41,21,5);
                set[pl].assign(42,35,6);
//              set[pl].assign(254,59,8);
                set[pl].assign(255,63,9);}
             }
*/
        }

channelset plane(int pl){ return set[pl];}
#ifdef DEBUG
void    print(){
        cout << " EVENT print \n";
        for (int i =0; i<8; ++i){
                cout << "   Plane " << i+1 << " \n" << flush;
                set[i].print();}
        cout<< " --------- End Event \n \n" << flush;
        }
#endif
};

class strip {
public:
int noise;
int value;
float dist;    //distance from origin
float width;    // width of strip
float length;    // length of strip over s-axis
};

class cluster{
protected:
coord   cc;             //coordinates of cluster
public:
float   size;           // number of hits (>1)
float   weight;         // weight =0 for dummy cluster
float   s_c;            // coordinate along s_axis (z_axis)
float   t_c;            // coordinate along transverse axis (x_ or y_ axis)
        cluster(float cx,float cy,float cs,float sz)
                { cc.x = cx; cc.y = cy; cc.z = cs; weight =1.; size = sz; }
#ifdef DEBUG
virtual void    print(){
        cout << "Cluster " << "t: " << t_c << "   s: "
                        << s_c << "  size: " << size << " \n" << flush;
        }
#endif
};

class h_cluster: public cluster{
public:
        h_cluster(float cx,float cs,float sz)
                :cluster(cx,0.,cs,sz) {         // y-coordinate =0
                t_c = cx; s_c = cs; }
#ifdef DEBUG
void    print(){
        cout << "Horizontal "  << flush; cluster::print();
        }
#endif
};


class v_cluster: public cluster{
public:
        v_cluster(float cy,float cs,float sz)
                :cluster(0.,cy,cs,sz) {         // x-coordinate = 0
                t_c = cy; s_c = cs; }
#ifdef DEBUG
void    print(){
        cout << "  Vertical "  << flush; cluster::print();
        }
#endif
};

class plane {
protected:
list<cluster>*  clusters;       // cluster set
strip           sl[nstrip];     // array of strips
float           x,y,s;          // coordinates of lower left corner of plane
float           mean;              // statistics
int             nr;             // number of entries

public:
        plane(coord c){
        clusters = new list<cluster>;   // empty cluster list to allow access to clusters
        x = c.x; y=c.y ; s = c.z;       // coordinates of plane
        mean = 0.; nr = 0;              //statistics for calibration
        for (int i = 0; i<nstrip; ++i) {
                sl[i].value = 0;
                sl[i].noise = 1;}               // to prevent division by zero
        }
cluster* first(){return clusters->first();};
cluster* get(){return clusters->current();};
cluster* next(){return clusters->next();};
bool    sequel(){ return clusters->sequel();}
void    assign(channelset s);
void    clearstatistics(){ mean = 0.; nr = 0;};
float   meandev(){ return mean;};
float   scoord() { return s;};
virtual void    storealignment(float cc){  };
void    adddeviation(float dt){
        if (nr == 0) {mean = dt; nr = 1;}
        else { mean = mean*(nr/(nr+1)); nr ++; mean = mean+(dt/nr);}
        };
virtual void addcluster(float t_c, float s_c, float sz) { };
#ifdef DEBUG
virtual void    print(){
        cout << "PLANE coordinates:: " << "x: " << x << "  y: " << y <<
                                      "  s: " << s << " \n";
        for (int i = 0; i<nstrip; ++i) {
           if (sl[i].value > 1){
                cout << "     strip: " << i << " value: " << sl[i].value  <<
                         "  noise: " << sl[i].noise << "  distance: " <<
                                sl[i].dist << " \n" << flush;}
           }
        }
#endif
};


void plane::assign(channelset ss){
        for (int i = 0; i< nstrip; ++i) {  // strip value and noise from channelset
                sl[i].value = ss.strip(i,1);
                sl[i].noise = ss.strip(i,2);}   // strips initialized

        delete clusters;                        // remove former clusters
        clusters = new list<cluster>;           // empty cluster list
        for (i = 0; i< nstrip; ++i) {  // determine clusters
                if (sl[i].value/sl[i].noise > hitquot){
                        float   total =sl[i].value-sl[i].noise;
                        float   posit = sl[i].dist*total;
                        int  k = i+1;
                        bool b = (k < nstrip);
                        if (b) {b = b & sl[k].value/sl[k].noise > hitquot;}
                        while (b) {
                            total =total+(sl[k].value -sl[k].noise);
                            posit = posit+sl[k].dist *float(
                                        sl[k].value-sl[k].noise);
                            k++;
                            b = (k < nstrip);
                            if (b) {b = b & sl[k].value/sl[k].noise > hitquot;}
                            }
                        k--;            // k always one too large
                        if (k > i) {
                                //more than two consecutive hits, add cluster
                                addcluster(posit/total, s,
                                        sl[k].dist-sl[i].dist+sl[k].width);}
                        i = k+1;}               //skip already inspected strips
                }
}


class h_plane: public plane{
```

```
public:
        h_plane(coord cc): plane(cc) {
        for (int i = 0; i<nstrip; ++i) {
                sl[i].dist = xwidth*float(i);   // x-position of strips from x=0
                sl[i].width = xwidth;           // dx width of strips
                sl[i].length = length;}         // thickness of strip over s-axis
        }
void    addcluster(float t_c, float s_c, float sz) {
                // add x coordinate of plane to position within plane
                h_cluster* h_cl = new h_cluster(t_c+x, s_c, sz);
                clusters->enter( h_cl);
        }
void    storealignment(float cc){ x = x+cc;};
#ifdef DEBUG
void    print(){
        cout << " HORIZONTAL ";
        plane::print();
        cout << "     clusters are: \n";
        clusters->print();
        cout << " --------------------- H_PLANE \n \n" << flush;
        }
#endif
};



class v_plane: public plane {
public:
        v_plane(coord cc): plane(cc) {
        for (int i = 0; i<nstrip; ++i) {
                sl[i].dist = ywidth*float(i);   // y-position of strips from y=0
                sl[i].width = ywidth;           // y-width of strips
                sl[i].length = length;}  //length of strip over s-axis
        }
void    addcluster(float t_c, float s_c, float sz) {
                // add y coordinate of plane to position within plane
                v_cluster* v_cl = new v_cluster(t_c+y, s_c, sz);
                clusters->enter( v_cl);
        }
void    storealignment(float cc){ y = y+cc;};
#ifdef DEBUG
void    print(){
        cout << "   VERTICAL ";
        plane::print();
        cout << "     clusters are: \n";
        clusters->print();
        cout << " --------------------- V_PLANE \n \n" << flush;
        }
#endif

};


class s_line{
public:
float   s_off;
float   ts_tg;
float   chi;
        s_line(float xt,float yt,float of){
                s_off = of; ts_tg = yt; chi = 0;}
bool    fit(list<cluster>* proj) {
        cluster* clpt = proj->first();
        int     ncl =0;                                 // number of clusters
        float   S_s = 0.;
        float   S_ss = 0.;
        float   S_t = 0.;
        float   S_tt = 0.;
        float   S_st = 0.;
        while ( clpt != null){
                if (clpt->weight > 0){
                        ncl = ncl + 1;                  //  number of entries
                        S_s = S_s+clpt->s_c;                    //sum over s
                        S_t = S_t+clpt->t_c;                    //sum over t
                        S_ss = S_ss+clpt->s_c*clpt->s_c;        //sum over s*s
                        S_st = S_st+clpt->s_c*clpt->t_c;        //sum over t*s
                        S_tt = S_tt+clpt->t_c*clpt->t_c;}       //sum over t*t
                clpt = proj->next();
                }
        if (ncl < 3){ return false;}    // at least three coordinates required
        float det = float(ncl)*S_ss - S_s*S_s ;
        if (det <= 0) { return false;}
        s_off = (S_ss*S_t - S_s*S_st)/det;
        ts_tg = (ncl*S_st - S_s*S_t)/det;
        chi = S_tt-s_off*S_t - ts_tg*S_st;
        return abs(chi) < linchi;
        }
#ifdef DEBUG
void    print(){
        cout << "LINE  ts_tg: " << ts_tg << "  s_off: "
                << s_off << "  chi: " << chi << " \n" << flush;}
#endif
};


class projection{
protected:
list<cluster>*  clp;    // pointer associated cluster
s_line  line;   // fitted line
public:
        projection(): line(0,0,0) { clp = new list<cluster>; }
        ~projection(){ delete clp;}
bool    valid(){ return line.fit(clp);}
float   tline( float sc){ return line.ts_tg*sc + line.s_off;}
void    add(cluster* clref) { if (clref != null) {clp->enter(clref);}}
void    AddDev( plane* planes[]){
        cluster* cp = clp->first();
        int i = (nplane/2)-1;
        while (cp != null) { // assume that clusters come from 4,3,2,1
                while (cp->s_c != planes[i]->scoord() & i >0) { i--;}
                planes[i]->adddeviation(cp->t_c - tline(cp->s_c));
                cp = clp->next();} // end while over clusters
        }

#ifdef DEBUG
virtual void    print(){
        cout << " projection \n" << flush;
        clp->print();
        line.print();}
#endif
};

class v_projection: public projection{
public:

#ifdef DEBUG
void    print () {
        cout <<"   VERTICAL ";
        projection::print();
        }
#endif
};

class h_projection: public projection{
public:

#ifdef DEBUG
void    print () {
        cout <<" HORIZONTAL ";
        projection::print();}
#endif
};

class projlst {
protected:
list<projection>*  lp;
        projlst() { lp = null;}
        ˜projlst() { if (lp != null) { delete lp;} }
public:
projection*     first() { return lp->first();}
virtual projection*     newproj() { return new projection();}
int     number() { return lp->number;}
virtual void    FindProj(plane* planes[]){
// find the list of projections in array of planes

        bool b = false;
        projection*  pp = newproj();
        for (int i = 0; i<4; ++i){
                pp->add(planes[i]->first());
                b = b | planes[i]->sequel();}  // more than one cluster in plane?
                // b implies there is a plane with more than 1 cluster
                // not b implies all planes have less than 2 clusters
        if (pp->valid()){lp->enter(pp);} // if good projection add to lvp
        else {delete pp;} // else remove contents and memory

// try all possible cluster combinations from vertical planes that have clusters

        cluster* cp;
        while (b){

                pp = newproj();
                b = false;
                bool sw = true;

        // sw implies get next cluster from plane or first one
        // not sw implies get current cluster from plane

                for (i = 0; i<4; ++i){
                        if (sw) {cp = planes[i]->next();}
                        else { cp = planes[i]->get();}
                        if (cp == null) { cp = planes[i]->first();}
                        else { sw = false;}

// sw remains true when when first cluster was taken from former plane
                        pp->add(cp);
                        b = b | planes[i]->sequel();}  // end for (i=0; i<4)

        // not b implies last cluster is found in all planes

                if (pp->valid()){lp->enter(pp);}
                else {delete pp;}
             }                              // end of while(b)
          }                                 // end of FindProj

#ifdef DEBUG
virtual void    print() {
        cout << "projection list \n";
        if (lp==null) {cout << " no list present \n";}
        else {lp->print();}
        }
#endif
};

class v_projlst: public projlst {
protected:
projection*     newproj(){ return new v_projection();}
public:
void    FindProj(plane* planes[]){
// find the list of projections in array of vertical planes

        delete lp;                      // remove old list
        lp = (list<projection>*) new list<h_projection>;
        projlst::FindProj( planes);
        }

#ifdef DEBUG
void    print() {
        cout << "\n   Vertical ";
        projlst::print();
        }
#endif
};


class h_projlst: public projlst {
protected:
projection*     newproj(){ return new h_projection();}
public:
void    FindProj(plane* planes[]){
// find the list of projections in array of vertical planes

        delete lp;                      // remove old list
        lp = (list<projection>*) new list<h_projection>;
        projlst::FindProj( planes);
        }

#ifdef DEBUG
void    print() {
        cout << "\n Horizontal ";
        projlst::print();
        }
#endif
};


class track{
projection*   hpr;
projection*   vpr;
public:
         track(projection* h, projection* v) {hpr = h; vpr = v;}
         ˜track() { delete hpr; delete vpr;}
projection* hproj() {return hpr;}
projection* vproj() {return vpr;}
```

```
#ifdef DEBUG
void    print(){
        cout << " TRACK \n" << flush;
        if (hpr != null) {hpr->print();}
        else { cout << " no horizontal projection \n" << flush;}
        if (vpr != null) {vpr->print();}
        else { cout << " no vertical projection \n" << flush;}
        }
#endif
};


class SiBTOO{
plane*  hplanes[nplane/2];      // 4 horizontal planes
plane*  vplanes[nplane/2];      // 4 vertical planes
h_projlst*   lhp;               // list of horizontal projections
v_projlst*   lvp;               // list of vertical projections



public:
        SiBTOO() {

        // initialize with 0 projections
        lhp = new h_projlst();
        lvp = new v_projlst();

        //initialize plane coordinates
        for (int i = 0; i<nplane/2; ++i){
                coord cc;
                cc.z = 2*i*(sbt_ds-0.05)/nplane;// s-coordinate
                cc.x = .1;                      // x-coordinate
                cc.y = cc.x;                    // y coordinate
                vplanes[i] = new v_plane(cc);   // vertical plane
                cc.z = cc.z+.05;                // s-coordinate
                                                // (slightly different from
                                                // vertical plane s-coordinate)
                hplanes[i] = new h_plane(cc);}  // horizontal plane
        }

track*  FindTracks(event ev){
        for (int i = 0; i<nplane/2; ++i){
                vplanes[i]->assign(ev.plane(i*2));   // vertical plane strips assigned
                hplanes[i]->assign(ev.plane(i*2+1));};// horizontal plane strips assigned
                // clusters of event are found and stored in planes

        // VERTICAL projections
        lvp->FindProj( vplanes);        // lvp contains vertical projections

        // HORIZONTAL projections
        lhp->FindProj( hplanes);        // lhp contains horizontal projections

        // only one projection from each orientation is allowed
                if(lhp->number() == 1 & lvp->number() == 1)
                        {return new track(lhp->first(), lvp->first());}
                else { return null;}
        }

void    Calibrate(list<track>* lst){
        for (int i = 0; i<nplane/2; ++i){
                vplanes[i]->clearstatistics();
                hplanes[i]->clearstatistics();}
        track*  trp = lst->first();     // get first track from list
        while (trp != null) {           // loop over all tracks

        // horizontal projections
                trp->hproj()->AddDev( hplanes);
        // vertical projections
                trp->vproj()->AddDev( vplanes);

                trp = lst->next();} // end while over all tracks
        // realign planes
        for (i = 0; i<nplane/2; ++i){
                hplanes[i]->storealignment(-hplanes[i]->meandev());
                vplanes[i]->storealignment(-vplanes[i]->meandev());}
        }


#ifdef DEBUG
void    print(){
        lhp->print();
        lvp->print();
        for (int i = 0; i<nplane/2; i++){
                vplanes[i]->print();
                hplanes[i]->print();}
        }
#endif
};


int main()
{
SiBTOO  sbt;
event   ev;             // some well chosen values
list<track>*    lst = new list<track>;
                        //initialize SiBTOO
#ifdef DEBUG
        ev.print();             // print event contents
        sbt.print();            // print all plane contents of SiBTOO
        cout << "\n Track list first time \n\n\n" << flush;
        lst->print();           // print all tracks
#endif
        lst->enter( sbt.FindTracks(ev));        // find one track
#ifdef DEBUG
        sbt.print();
        cout << "\n Track list 2nd time\n\n\n" << flush;
        lst->print();
#endif
        sbt.Calibrate(lst);             // calibrate with one track
#ifdef DEBUG
        sbt.print();
#endif

        lst->enter( sbt.FindTracks(ev));        // find same track in calibrated SiBTOO
#ifdef DEBUG
        sbt.print();
        cout << "\n Track list 3rd time\n\n\n" << flush;
        lst->print();
#endif

cout << " main stopped\n";
}
```

B Layer code

```
#include <stdlib.h>
#include <iostream.h>
#include <math.h>
#include <vector.h>
#include <algo.h>

// standard definition of NULL and bool
const int null = 0;

// constants for detector installation
const int nlayer = 8;     // number of layers in installation
const int ndetct = 16;    // numer of detectors in layer
const float rinner = 20.; // inner radius of installation
const float router = 40.; // outer radius of installation
const float rspace = .2;  // space between two layers
const float detcdr = (router - rinner)/nlayer - rspace;
                           // thickness of detector
const float detcds = 6.2;   // length of detector over s-axis
const float dphi = M_PI /ndetct; // angle over which detector extends

// constants for event layout
const int nr_det = 3;      // 3 entries per detector in event

// acceptance limits and allowed deviations
const float     linchi  = 10.;                 // chi of accepted line fit
const float     phi_dev = dphi/100.;           // tolerated segment deviations


class cylpoint{
// point in cylindrical coordinate system
public:
float   r;              // r coordinate
float   phi;            // phi angle coordinate
float   s;              // cylinder axis coordinate
};

class measur: public cylpoint{
// measurement point in cylindrical coordinate system
public:
float      dr,ds,dphi;  // measurement uncertainties in above
};

// Define == on measur objects
bool operator==(const measur&  m1, const measur& m2)
{
        return m1.phi == m2.phi;
}
// Define < on measur objects
bool operator<(const measur&  m1, const measur& m2)
{
        return m1.phi < m2.phi;
}


class layerset{
float  items[50];
int  as,k;
public:
        layerset(){
                k = 0;
                as = 0;
                for( int i =0; i<50; ++i) {items[i] = 0.;}
        }

float*  detect(float phi ) {
                for (int i = k; i<as ; i = i+nr_det){
                if (items[i] > phi & items[i] < phi + 2*dphi){
                        k = i+nr_det;
                        return &items[i];} // coordinate in detector with
                                             // angle phi
                }
        k = 0;                          // start from beginning
        return null;}                   // no hit found

void    assign( float phi, float r, float s){
        items[as] = phi; as++;
        items[as] = r; as++;
        items[as] = s; as++;}
#ifdef DEBUG
void    print(){
        int i=0;
        while (i < as){
                cout << "          angle: " << items[i] <<"    r: " << items[i+1]
                << "   s: " << items[i+2] << "\n" << flush;
                i = i+nr_det;}
        }
#endif
};


class event{
layerset        set[nlayer];    // 1 event: nlayer layersets
public:
        event(){
        for (int pl = 0; pl <nlayer ; ++pl){
                set[pl].assign(M_PI/4+.5*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.25,0);
                set[pl].assign(M_PI/4+.5*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.3,0);
                set[pl].assign(M_PI/4+.5*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.5,0);
                set[pl].assign(M_PI/4+.5*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.66,0);
                set[pl].assign(M_PI/4+.5*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.75,0);
             }

        for ( pl = 0; pl <nlayer ; ++pl){
                set[pl].assign(M_PI/3+dphi*.01+pl*.023*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.25,0);
                set[pl].assign(M_PI/3+dphi*.015+pl*.023*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.3,0);
                set[pl].assign(M_PI/3+dphi*.017+pl*.023*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.5,0);
                set[pl].assign(M_PI/3+dphi*.02+pl*.023*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.66,0);
                set[pl].assign(M_PI/3+dphi*.022+pl*.023*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.75,0);
             }

        }

layerset layer(int pl){ return set[pl];}
```

```
#ifdef DEBUG
void    print(){
        cout << " EVENT print \n";
        for (int i =0; i<nlayer; ++i){
                cout << "   Layer " << i << " \n" << flush;
                set[i].print();}
        cout<< " --------- End Event \n \n" << flush;
        }
#endif
};

class segment{
public:
vector<measur>  points;
cylpoint        entry;
cylpoint        exit;

void    enter(measur* m){ points.push_back(*m);}
bool    valid(float r,float phi) {
float   s_off, of;
float   ts_tg, tg;
float   chi;
float   s,t;            // coordinates after transformations
int     nms =0;                         // number of measurements
float   S_s = 0.;
float   S_ss = 0.;
float   S_t = 0.;
float   S_tt = 0.;
float   S_st = 0.;
        if (points.begin() != points.end()){
                vector<measur>::iterator m;
                for (m= points.begin(); m != points.end(); ++m){
                        nms = nms + 1;          //  number of entries
                        s = m->r*cos(m->phi);
                        t = m->r*sin(m->phi);
                        S_s = S_s + s;          //sum over s
                        S_t = S_t + t;          //sum over t
                        S_ss = S_ss+s*s;        //sum over s*s
                        S_st = S_st+t*s;        //sum over t*s
                        S_tt = S_tt+t*t;}       //sum over t*t
                }
        if (nms < 3){ return false;}    // at least three coordinates required
        float det = float(nms)*S_ss - S_s*S_s ;
        if (det <= 0) { return false;}
        s_off = (S_ss*S_t - S_s*S_st)/det;
        ts_tg = (nms*S_st - S_s*S_t)/det;
        chi = S_tt-s_off*S_t - ts_tg*S_st;

        // calculate entry and exit points of line
        entry.s = 0; exit.s = 0;                // default values
        tg = (sin(phi)-sin(phi+2.*dphi))/(cos(phi)-cos(phi+2.*dphi));
        // inner layer, entry point
        of = r*sin(phi)-tg*r*cos(phi);
        s = (s_off-of)/(tg-ts_tg);
        t = tg*s+of;
        entry.r = sqrt(s*s + t*t);
        entry.phi = atan2(t,s);

        // outer layer exit point
        r = r+detcdr+rspace;
        of = r*sin(phi)-tg*r*cos(phi);
        s = (s_off-of)/(tg-ts_tg);
        t = tg*s+of;
        exit.r = sqrt(s*s + t*t);
        exit.phi = atan2(t,s);

        return abs(chi) < linchi;
        }

};

// Define == on segment objects
bool operator==(const segment&  s1, const segment& s2)
{
        return s1.exit.phi == s2.exit.phi;
}
// Define < on segment objects
bool operator<(const segment&  s1, const segment& s2)
{
        return s1.exit.phi < s2.exit.phi;
}

class track{
vector<segment> trk;    // segments that constitute track
public:
cylpoint        entry;
cylpoint        exit;
void    enter(segment* s){
        entry = s->entry;
        if (trk.begin() == trk.end()){ exit = s->exit;}
        trk.push_back( *s);
        }
#ifdef DEBUG
void    print(){
        cout << "TRACK with entry.phi: " <<  entry.phi << "  exit.phi: " << exit.phi << "\n";
        cout << "           entry.r:   " << entry.r << " exit.r:  " << exit.r << "\n";
        vector<segment>::iterator sg;
        for (sg= trk.begin(); sg != trk.end(); ++sg){
                cout << "          segment exit phi: " << sg->exit.phi;
                cout << "  exit r: " << sg->exit.r << "\n" << flush;
                cout << "          segment entry phi: " << sg->entry.phi;
                cout << "  entry r: " << sg->entry.r << "\n";}
        }
#endif
};

// Define == on track objects
bool operator==(const track&  t1, const track& t2)
{
        return t1.entry.phi == t2.entry.phi;
}
// Define < on track objects
bool operator<(const track&  t1, const track& t2)
{
        return t1.entry.phi < t2.entry.phi;
}



class detector{
public:
cylpoint        pt[8];  // detector defined by 8 points in space
vector<segment> seglst;  // list of segments
vector<measur>  measlst; // list of measurements

        detector() { };
        detector(float r, float phi){
        for (int i = 0 ; i< 4; i++) {pt[i].s = 0; pt[i+4].s=detcds;}
        pt[0].r = r; pt[1].r = r;
        pt[4].r = r; pt[5].r = r;
        pt[2].r = r+detcdr; pt[3].r = r+detcdr;
        pt[6].r = r+detcdr; pt[7].r = r+detcdr;
        pt[0].phi = phi - dphi; pt[4].phi = phi- dphi;
        pt[3].phi = phi - dphi; pt[7].phi = phi- dphi;
        pt[1].phi = phi + dphi; pt[5].phi = phi+ dphi;
        pt[2].phi = phi + dphi; pt[6].phi = phi+ dphi;
        }  // end of detector constructor
void    add(measur m){measlst.push_back( m);}
void    clean(){
        // remove all segments and measurements
        seglst.erase(seglst.begin(),seglst.end());
        measlst.erase(measlst.begin(),measlst.end());
        } // end of clean
void    FindSegment(){
        if (measlst.begin() != measlst.end()){ // measurements present
                segment sg;
                vector<measur>::iterator m;
                for (m= measlst.begin(); m != measlst.end(); ++m){
                        sg.enter(m);} // all measurements in one segment
                if (sg.valid(pt[0].r,pt[0].phi)){seglst.push_back(sg);}
                                                // add sg to segment list
                } // end of if (measlst...
        } // end of FindSegment

#ifdef DEBUG
void    print(){
        cout << " Detector ";
        cout << " pt[0].phi: " <<  pt[0].phi  << " pt[0].r: " << pt[0].r;
        cout << " pt[2].phi: " <<  pt[2].phi  << " pt[2].r: " << pt[2].r;
        cout << " \n" << flush;

        if (seglst.size() !=0){
           vector<segment>::iterator sg;
           for (sg= seglst.begin(); sg != seglst.end(); ++sg){
                cout << "          segment entry phi: " << sg->entry.phi;
                cout << "  entry r: " << sg->entry.r << "\n";
                cout << "          segment exit phi: " << sg->exit.phi;
                cout << "  exit r: " << sg->exit.r << "\n" << flush;}
           } // end of if (seglst.size ...

        if (measlst.size() !=0){
                vector<measur>::iterator m;
                for (m= measlst.begin(); m != measlst.end(); ++m){
                        cout <<"      measurement r: " << m->r <<
                        "   phi: " << m->phi << " \n" << flush;}
                } // end of if
        } // end of print
#endif
};

// Define == on detector objects
bool operator==(const detector&  d1, const detector& d2)
{
        return d1.pt[0].phi == d2.pt[0].phi;
}
// Define < on detector objects
bool operator<(const detector&  d1, const detector& d2)
{
        return d1.pt[0].phi < d2.pt[0].phi;
```

```
};


class layer{
public:
vector<detector>        dts;
        layer(float ri) {
// create a layer with inner circle defined by ri
        for (int i=0; i < ndetct; i++){
                dts.push_back( * new detector(ri, dphi*i*2));}
        sort(dts.begin(), dts.end(),less<detector>() );
        }  //end of layer constructor
void    assign(layerset ls){
        float*  pm;
        measur  m;
        m.dr =0.; m.dphi = 0.; m.ds =0.;
        vector<detector>::iterator d;
        for (d= dts.begin(); d != dts.end(); ++d){
                d->clean();             // remove segments and measurements
                pm = ls.detect(d->pt[0].phi);
                while (pm != null){
                        m.phi = pm[0];
                        m.r   = pm[1];
                        m.s   = pm[2];
                        d->add(m);
                        pm = ls.detect(d->pt[0].phi);} // end of while
                d->FindSegment(); // construct segments
                } // end of for
        } // end of assign
#ifdef DEBUG
void    print(){
        int i =1;
        vector<detector>::iterator d;
        for (d= dts.begin(); d != dts.end(); ++d){
                cout << i; i++; d->print();}
        }
#endif
};

class tracklist{
vector<track>   tklst;  // list of tracks
public:
void    extend  (layer* ly){
        float   cr_ent;                 // entry point of current track
        vector<track>::iterator t = tklst.begin();
        if (t == tklst.end()){ cr_ent = 2*M_PI+dphi;}  // empty track list
        else { cr_ent = t->entry.phi;}

        // store all segments of all detectors in layer ly into sglst
        vector<segment>  sglst;         // list of segments in layer ly
        vector<detector>::iterator d;
        for (d= ly->dts.begin(); d != ly->dts.end(); ++d){
             copy(d->seglst.begin(), d->seglst.end(), back_inserter(sglst));}
        sort(sglst.begin(),sglst.end());        // sorted segments of layer ly

        // start extension of creation of tracks
        vector<segment>::iterator s= sglst.begin();
        while (s != sglst.end()){
                while ((s->exit.phi < cr_ent - phi_dev) & (s != sglst.end())){
                        track tk;               // new track
                        tk.enter( s); s++;      // track with one segment
                        tklst.push_back( tk);}  // add new track to list
                if (s != sglst.end()){
                        while ( cr_ent < s->exit.phi-phi_dev){
                            t ++;                       // next track
                            if (t == tklst.end()){ cr_ent = 2*M_PI+dphi;}
                            else { cr_ent = t->entry.phi;}
                            } // end while ...
                        if (abs(s->exit.phi - cr_ent) < phi_dev) {
                            t->enter( s); s++;}         // extend track
                        } // end of if (s != ....
                } // end while (s!=sglst ...
        sort(tklst.begin(),tklst.end());
        } //end extend

#ifdef DEBUG
void    print(){
        cout << "TRACK LIST \n";
        int     cnt = 1;
        vector<track>::iterator  t;
        for (t= tklst.begin(); t != tklst.end(); ++t){
                cout << cnt << "  " ; cnt++;
                t->print();}
        }
#endif
};


class installation{
layer*          ly[nlayer];     // nlayer layers in installation
tracklist*      tklist;         // list of tracks
public:
        installation(){
        tklist = null;
        for (int i = 0; i <nlayer; i++) {
                ly[i]= new layer(i*(router-rinner)/nlayer + rinner); }
        }
tracklist*      GetTracks(event ev){
        delete tklist;                  // remove old tracks
        tklist = new tracklist();       // new empty track list

// find tracks by going from outer layer to inner layer
        for (int i = nlayer-1; i>-1; i--) {
                ly[i]->assign(ev.layer(i));  // measurements into layer i
                tklist->extend(ly[i]);       // tracks include layer i segments
                }
        // measurements are assigned to layers
        return tklist;
        }
#ifdef DEBUG
void    print(){
        for (int i=0; i < nlayer; i++){
                cout << " Layer " << i << " \n";
                ly[i]->print();
                cout << " \n" << flush;}
        tklist->print();                // print found tracks
        }
#endif
};


int main(){
installation    instal;
event   ev;
tracklist*      trlst;
#ifdef DEBUG
```

```
        ev.print();
        cout << "debug \n";
//      instal.print();
#endif
        trlst = instal.GetTracks(ev);
#ifdef DEBUG
        instal.print();
#endif
cout << " main stopped\n";
}
```

C Layer code

```
#include <stdlib.h>
#include <iostream.h>
#include <math.h>
#include <vector.h>
#include <algo.h>

// standard definition of NULL and bool
const int null = 0;

// constants for detector installation
const int nlayer = 8;     // number of layers in installation
const int ndetct = 16;    // numer of detectors in layer
const float rinner = 20.; // inner radius of installation
const float router = 40.; // outer radius of installation
const float rspace = .2;  // space between two layers
const float detcdr = (router - rinner)/nlayer - rspace;
                           // thickness of detector
const float detcds = 6.2;   // length of detector over s-axis
const float dphi = M_PI /ndetct; // angle over which detector extends

// constants for event layout
const int nr_det = 3;      // 3 entries per detector in event

// acceptance limits and allowed deviations
const float     linchi  = 10.;          // chi of accepted line fit
const float     phi_dev = dphi/100.;    // tolerated segment deviations
const float     curv = 100.;            // curvature of standard particle


class cylpoint{
// point in cylindrical coordinate system
public:
float   r;              // r coordinate
float   phi;            // phi angle coordinate
float   s;              // cylinder axis coordinate
};

class measur: public cylpoint{
// measurement point in cylindrical coordinate system
public:
float      dr,ds,dphi;  // measurement uncertainties in above
};

// Define == on measur objects
bool operator==(const measur&  m1, const measur& m2)
{
        return m1.phi == m2.phi;
}
// Define < on measur objects
bool operator<(const measur&  m1, const measur& m2)
{
        return m1.phi < m2.phi;
}

class vect: public cylpoint{
// point with direction in plane perpendicular to s
public:
float   angle;          // orientation in plane perpendicular to s
};
```

```
class particle: public vect{
// physical characteristics of particle
public:
float           mass;           // particle’s mass
float           p;              // particle’s momentum
void    Propagate(vect cc) {
                phi = cc.phi; r = cc.r; s = cc.s; angle = cc.angle;}
virtual vect Intersect(cylpoint cc1, cylpoint cc2, float B) {
        vect vc;
        vc.phi = 4*M_PI; vc.angle =0;
        vc.r = 2*router; vc.s =0;
        return vc; }
#ifdef DEBUG
virtual void print() { }
#endif
};

class charged_particle: public particle{
// charged particle trajectory is circle
public:
int     charge;                 // particle’s charge (+1 or -1)
vect    Intersect(cylpoint cc1, cylpoint cc2, float B) {
        double x1 = cc1.r*cos(cc1.phi);
        double x2 = cc2.r*cos(cc2.phi);

        // calculate centre coordinates of circle
        double Radius = curv*p/B;
        float xm = Radius*cos(angle+charge*M_PI/2.) + r*cos(phi);
        float ym = Radius*sin(angle+charge*M_PI/2.) + r*sin(phi);
        // calculate intersection point of line with circle
        double tg = (cc1.r*sin(cc1.phi) - cc2.r*sin(cc2.phi)) /
                     (cc1.r*cos(cc1.phi) - cc2.r*cos(cc2.phi));
        double of = cc1.r*sin(cc1.phi)-cc1.r*cos(cc1.phi)*tg;
        double a = 1+tg*tg;
        double b = 2.*tg*(of -ym)-2*xm;
        double c = xm*xm+(of-ym)*(of-ym)-Radius*Radius;
        double det = b*b - 4*a*c;

        vect vc;
        vc.s = 0;
        if (det < 0) {          // no intersection point
                vc.phi = 4*M_PI;
                vc.r = 2*router;
                return vc;}
        double x = (-b + sqrt(det))/(2*a);
        if (x < min(x1,x2) | x > max(x1,x2)) { // no intersection
                x = (-b - sqrt(det))/(2*a);}
        double y = tg*x + of;
        vc.s = 0;
        vc.phi = atan2(y,x);
        vc.r = sqrt(x*x + y*y);
        vc.angle = atan2(ym-y,xm-x)-charge*M_PI/2.;
        return vc;
        }
#ifdef DEBUG
void    print(){
        cout << " Charged particle with phi: " << phi <<
                "  r:  " << r << "  angle " << angle << "\n" << flush;
        }
#endif
};

class neutral_particle: public particle{
```

```
// neutral particle trajectory is straight line
vect    Intersect(cylpoint cc1, cylpoint cc2, float B) {
        double tg1 = tan( angle);
        double of1 = r*sin(phi)-r*cos(phi)*tg1;

        double tg2 = (cc1.r*sin(cc1.phi) - cc2.r*sin(cc2.phi)) /
                     (cc1.r*cos(cc1.phi) - cc2.r*cos(cc2.phi));
        double of2 = cc1.r*sin(cc1.phi)-cc1.r*cos(cc1.phi)*tg2;

        double x = (of2-of1)/(tg1-tg2);
        double y = tg1*x + of1;
        vect vc;
        vc.s = 0;
        vc.phi = atan2(y,x);
        vc.r = sqrt(x*x + y*y);
        vc.angle = angle;
        return vc;
        }
#ifdef DEBUG
void    print(){
        cout << " Neutral particle with phi: " << phi <<
                "  r:  " << r << "  angle " << angle << "\n" << flush;
        }
#endif
};


class layerset{
float  items[50];
int  as,k;
public:
        layerset(){
                k = 0;
                as = 0;
                for( int i =0; i<50; ++i) {items[i] = 0.;}
        }

float*  detect(float phi ) {
                for (int i = k; i<as ; i = i+nr_det){
                if (items[i] > phi & items[i] < phi + 2*dphi){
                        k = i+nr_det;
                        return &items[i];} // coordinate in detector with
                                             // angle phi
                }
        k = 0;                          // start from beginning
        return null;}                   // no hit found

void    assign( float phi, float r, float s){
        items[as] = phi; as++;
        items[as] = r; as++;
        items[as] = s; as++;}
#ifdef DEBUG
void    print(){
        int i=0;
        while (i < as){
                cout << "          angle: " << items[i] <<"    r: " << items[i+1]
                << "   s: " << items[i+2] << "\n" << flush;
                i = i+nr_det;}
        }
#endif
};
```

```
class event{
layerset        set[nlayer];    // 1 event: nlayer layersets
public:
        event(){
        for (int pl = 0; pl <nlayer ; ++pl){
                set[pl].assign(M_PI/4+.5*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.25,0);
                set[pl].assign(M_PI/4+.5*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.3,0);
                set[pl].assign(M_PI/4+.5*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.5,0);
                set[pl].assign(M_PI/4+.5*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.66,0);
                set[pl].assign(M_PI/4+.5*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.75,0);
             }

        for ( pl = 0; pl <nlayer ; ++pl){
                set[pl].assign(M_PI/3+dphi*.01+pl*.023*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.25,0);
                set[pl].assign(M_PI/3+dphi*.015+pl*.023*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.3,0);
                set[pl].assign(M_PI/3+dphi*.017+pl*.023*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.5,0);
                set[pl].assign(M_PI/3+dphi*.02+pl*.023*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.66,0);
                set[pl].assign(M_PI/3+dphi*.022+pl*.023*dphi,
                        pl*(router-rinner)/nlayer+rinner+detcdr*.75,0);
             }

        }

layerset layer(int pl){ return set[pl];}
#ifdef DEBUG
void    print(){
        cout << " EVENT print \n";
        for (int i =0; i<nlayer; ++i){
                cout << "   Layer " << i << " \n" << flush;
                set[i].print();}
        cout<< " --------- End Event \n \n" << flush;
        }
#endif
};

class segment{
public:
vector<measur>  points;
cylpoint        entry;
cylpoint        exit;

void    enter(measur* m){ points.push_back(*m);}
bool    valid(float r,float phi) {
float   s_off, of;
float   ts_tg, tg;
float   chi;
float   s,t;            // coordinates after transformations
int     nms =0;                         // number of measurements
float   S_s = 0.;
float   S_ss = 0.;
float   S_t = 0.;
float   S_tt = 0.;
float   S_st = 0.;
        if (points.begin() != points.end()){
                vector<measur>::iterator m;
                for (m= points.begin(); m != points.end(); ++m){
                        nms = nms + 1;          //  number of entries
                        s = m->r*cos(m->phi);
                        t = m->r*sin(m->phi);
                        S_s = S_s + s;          //sum over s
                        S_t = S_t + t;          //sum over t
                        S_ss = S_ss+s*s;        //sum over s*s
                        S_st = S_st+t*s;        //sum over t*s
                        S_tt = S_tt+t*t;}       //sum over t*t
                }
        if (nms < 3){ return false;}    // at least three coordinates required
        float det = float(nms)*S_ss - S_s*S_s ;
        if (det <= 0) { return false;}
        s_off = (S_ss*S_t - S_s*S_st)/det;
        ts_tg = (nms*S_st - S_s*S_t)/det;
        chi = S_tt-s_off*S_t - ts_tg*S_st;

        // calculate entry and exit points of line
        entry.s = 0; exit.s = 0;                // default values
        tg = (sin(phi)-sin(phi+2.*dphi))/(cos(phi)-cos(phi+2.*dphi));
        // inner layer, entry point
        of = r*sin(phi)-tg*r*cos(phi);
        s = (s_off-of)/(tg-ts_tg);
        t = tg*s+of;
        entry.r = sqrt(s*s + t*t);
        entry.phi = atan2(t,s);

        // outer layer exit point
        r = r+detcdr+rspace;
        of = r*sin(phi)-tg*r*cos(phi);
        s = (s_off-of)/(tg-ts_tg);
        t = tg*s+of;
        exit.r = sqrt(s*s + t*t);
        exit.phi = atan2(t,s);

        return abs(chi) < linchi;
        }

};

// Define == on segment objects
bool operator==(const segment&  s1, const segment& s2)
{
        return s1.exit.phi == s2.exit.phi;
}
// Define < on segment objects
bool operator<(const segment&  s1, const segment& s2)
{
        return s1.exit.phi < s2.exit.phi;
}

class track{
vector<segment> trk;    // segments that constitute track
public:
cylpoint        entry;
cylpoint        exit;
void    enter(segment* s){
        entry = s->entry;
        if (trk.begin() == trk.end()){ exit = s->exit;}
        trk.push_back( *s);
        }
#ifdef DEBUG
```

```
void    print(){
        cout << "TRACK with entry.phi: " <<  entry.phi << "  exit.phi: " << exit.phi << "\n";
        cout << "           entry.r:   " << entry.r << " exit.r:  " << exit.r << "\n";
        vector<segment>::iterator sg;
        for (sg= trk.begin(); sg != trk.end(); ++sg){
                cout << "          segment exit phi: " << sg->exit.phi;
                cout << "  exit r: " << sg->exit.r << "\n" << flush;
                cout << "          segment entry phi: " << sg->entry.phi;
                cout << "  entry r: " << sg->entry.r << "\n";}
        }
#endif
};

// Define == on track objects
bool operator==(const track&  t1, const track& t2)
{
        return t1.entry.phi == t2.entry.phi;
}
// Define < on track objects
bool operator<(const track&  t1, const track& t2)
{
        return t1.entry.phi < t2.entry.phi;
}



class detector{
float           B;      // magnetic field in detector
public:
cylpoint        pt[8];  // detector defined by 8 points in space
vector<segment> seglst;  // list of segments
vector<measur>  measlst; // list of measurements
detector*       r_bk;    // links to surrounding detectors
detector*       r_fw;
detector*       ph_bk;
detector*       ph_fw;

        detector() { };
        detector(float r, float phi){
        r_bk = null; r_fw = null; ph_bk = null; ph_fw = null;
        for (int i = 0 ; i< 4; i++) {pt[i].s = 0; pt[i+4].s=detcds;}
        B = 1.;
        pt[0].r = r; pt[1].r = r;
        pt[4].r = r; pt[5].r = r;
        pt[2].r = r+detcdr; pt[3].r = r+detcdr;
        pt[6].r = r+detcdr; pt[7].r = r+detcdr;
        pt[0].phi = phi - dphi; pt[4].phi = phi- dphi;
        pt[3].phi = phi - dphi; pt[7].phi = phi- dphi;
        pt[1].phi = phi + dphi; pt[5].phi = phi+ dphi;
        pt[2].phi = phi + dphi; pt[6].phi = phi+ dphi;
        }  // end of detector constructor
void    add(measur m){measlst.push_back( m);}
void    clean(){
        // remove all segments and measurements
        seglst.erase(seglst.begin(),seglst.end());
        measlst.erase(measlst.begin(),measlst.end());
        } // end of clean
void    FindSegment(){
        if (measlst.begin() != measlst.end()){ // measurements present
                segment sg;
                vector<measur>::iterator m;
                for (m= measlst.begin(); m != measlst.end(); ++m){
                        sg.enter(m);} // all measurements in one segment
                if (sg.valid(pt[0].r,pt[0].phi)){seglst.push_back(sg);}
                                                // add sg to segment list
                } // end of if (measlst...
        } // end of FindSegment

virtual void trace(particle& ptk) {
        detector* dtp;
        vect cc = ptk.Intersect(pt[2],pt[3], B);
        dtp = r_fw;
        if (cc.phi > pt[2].phi | cc.phi < pt[3].phi){
                cc = ptk.Intersect(pt[0],pt[3], B);
                dtp = ph_bk;
                if (cc.r < pt[0].r | cc.r > pt[3].r){
                        cc = ptk.Intersect(pt[1],pt[2], B);
                        dtp = ph_fw;}
                } // end if(cc.phi < pt[2].phi.....
        ptk.Propagate(cc);              // advance particle
#ifdef DEBUG
        ptk.print();
#endif
        dtp->trace(ptk);                // next detector
        } // end of trace

#ifdef DEBUG
void    print(){
        cout << " Detector ";
        cout << " pt[0].phi: " <<  pt[0].phi  << " pt[0].r: " << pt[0].r;
        cout << " pt[2].phi: " <<  pt[2].phi  << " pt[2].r: " << pt[2].r;
        cout << " \n" << flush;

        if (seglst.size() !=0){
           vector<segment>::iterator sg;
           for (sg= seglst.begin(); sg != seglst.end(); ++sg){
                cout << "          segment entry phi: " << sg->entry.phi;
                cout << "  entry r: " << sg->entry.r << "\n";
                cout << "          segment exit phi: " << sg->exit.phi;
                cout << "  exit r: " << sg->exit.r << "\n" << flush;}
           } // end of if (seglst.size ...

        if (measlst.size() !=0){
                vector<measur>::iterator m;
                for (m= measlst.begin(); m != measlst.end(); ++m){
                        cout <<"      measurement r: " << m->r <<
                        "   phi: " << m->phi << " \n" << flush;}
                } // end of if
        } // end of print
#endif
};

class dummy_detector: public detector{
public:
void trace(particle& ptk) {
#ifdef DEBUG
cout << "End of particle trace \n \n" << flush;
#endif
        }
};

// Define == on detector objects
bool operator==(const detector&  d1, const detector& d2)
{
        return d1.pt[0].phi == d2.pt[0].phi;
}
```

```
// Define < on detector objects
bool operator<(const detector&  d1, const detector& d2)
{
        return d1.pt[0].phi < d2.pt[0].phi;
};


class layer{
public:
vector<detector>        dts;
        layer(float ri) {
// create a layer with inner circle defined by ri
        for (int i=0; i < ndetct; i++){
                dts.push_back( *new detector(ri, dphi*i*2));}
        sort(dts.begin(), dts.end(),less<detector>() );
        }  //end of layer constructor
void    connect(layer* ly){
        vector<detector>::iterator dup = ly->dts.begin();
        vector<detector>::iterator d,dbk = dts.begin();
        for (d= dts.begin(); d != dts.end(); ++d){
                dbk->ph_fw = d; d->ph_bk = dbk;
                d->r_fw = dup; dup->r_bk = d; ++dup;
                dbk = d;}
        // dbk points to last detector
        d = dts.begin(); dbk->ph_fw = d; d->ph_bk =dbk;
        } // end of connect(layer)
void    connect(detector* dup){
        vector<detector>::iterator d,dbk = dts.begin();
        for (d= dts.begin(); d != dts.end(); ++d){
                dbk->ph_fw = d; d->ph_bk = dbk;
                d->r_fw = dup;
                dbk = d;}
        // dbk points to last detector
        d = dts.begin(); dbk->ph_fw = d; d->ph_bk =dbk;
        } // end of connect(detector)
void    assign(layerset ls){
        float*  pm;
        measur  m;
        m.dr =0.; m.dphi = 0.; m.ds =0.;
        vector<detector>::iterator d;
        for (d= dts.begin(); d != dts.end(); ++d){
                d->clean();             // remove segments and measurements
                pm = ls.detect(d->pt[0].phi);
                while (pm != null){
                        m.phi = pm[0];
                        m.r   = pm[1];
                        m.s   = pm[2];
                        d->add(m);
                        pm = ls.detect(d->pt[0].phi);} // end of while
                d->FindSegment(); // construct segments
                } // end of for
        } // end of assign
#ifdef DEBUG
void    print(){
        int i =1;
        vector<detector>::iterator d;
        for (d= dts.begin(); d != dts.end(); ++d){
                cout << i; i++; d->print();}
        }
#endif
};

class tracklist{
```

```
public:
vector<track>   tklst;  // list of tracks
void    extend  (layer* ly){
        float   cr_ent;                 // entry point of current track
        vector<track>::iterator t = tklst.begin();
        if (t == tklst.end()){ cr_ent = 2*M_PI+dphi;}  // empty track list
        else { cr_ent = t->entry.phi;}

        // store all segments of all detectors in layer ly into sglst
        vector<detector>::iterator d;
        vector<segment>  sglst;         // list of segments in layer ly
        for (d= ly->dts.begin(); d != ly->dts.end(); ++d){
             copy(d->seglst.begin(), d->seglst.end(), back_inserter(sglst));}
        sort(sglst.begin(),sglst.end());        // sorted segments of layer ly

        // start extension of creation of tracks
        vector<segment>::iterator s= sglst.begin();
        while (s != sglst.end()){
                while ((s->exit.phi < cr_ent - phi_dev) & (s != sglst.end())){
                        track tk;               // new track
                        tk.enter( s); s++;      // track with one segment
                        tklst.push_back( tk);}  // add new track to list
                if (s != sglst.end()){
                        while ( cr_ent < s->exit.phi-phi_dev){
                            t ++;                       // next track
                            if (t == tklst.end()){ cr_ent = 2*M_PI+dphi;}
                            else { cr_ent = t->entry.phi;}
                            } // end while ...
                        if (abs(s->exit.phi - cr_ent) < phi_dev) {
                            t->enter( s); s++;}         // extend track
                        } // end of if (s != ....
                } // end while (s!=sglst ...
        sort(tklst.begin(),tklst.end());
        } //end extend

#ifdef DEBUG
void    print(){
        cout << "TRACK LIST \n";
        if (tklst.end() == tklst.begin()) { cout << "       EMPTY \n";}
        int     cnt = 1;
        vector<track>::iterator  t;
        for (t= tklst.begin(); t != tklst.end(); ++t){
                cout << cnt << "  " ; cnt++;
                t->print();}
        }
#endif
};


class installation{
layer*          ly[nlayer];     // nlayer layers in installation
tracklist*      tklist;         // list of tracks
detector*       exitd;          // end of tracing detector
public:
        installation(){
        exitd = new dummy_detector();
        tklist =  new tracklist();      // for printing empty tracklist
        ly[nlayer-1]= new layer(router);
        ly[nlayer-1]->connect( exitd);
        for (int i = nlayer-2; i > -1; i--) {
                ly[i]= new layer(i*(router-rinner)/nlayer + rinner);
                ly[i]->connect( ly[i+1]); }
        }
```

```
tracklist*      GetTracks(event ev){
        delete tklist;                  // remove old tracks
        tklist = new tracklist();       // new empty track list

// find tracks by going from outer layer to inner layer
        for (int i = nlayer-1; i>-1; i--) {
                ly[i]->assign(ev.layer(i));  // measurements into layer i
                tklist->extend(ly[i]);       // tracks include layer i segments
                }
        // measurements are assigned to layers
        return tklist;
        }
void    Trace(particle& ptk){
#ifdef DEBUG
        cout << " Start particle trace \n" << flush;
        ptk.print();
#endif
        float phip = ptk.phi+dphi;
        phip = int(phip/(2*dphi))*2*dphi;
        detector key = detector(rinner,phip);
        vector<detector>::iterator where = find(ly[0]->dts.begin(),
                                ly[0]->dts.end(), key);
        if (where != ly[0]->dts.end()) {where->trace(ptk);}
        else { cout << " COULD NOT FIND DETECTOR for Tracing \n" << flush;}
        }

#ifdef DEBUG
void    print(){
        for (int i=0; i < nlayer; i++){
                cout << " Layer " << i << " \n";
                ly[i]->print();
                cout << " \n" << flush;}
        tklist->print();                // print found tracks
        }
#endif
};


int main(){
installation    instal;
event           ev;
#ifdef DEBUG
/*      ev.print();
        cout << "debug \n";
        instal.print();
*/
#endif
        tracklist*  trlst = instal.GetTracks(ev);
        vector<track>*   vt = &(trlst->tklst);
        vector<track>::iterator t;

        neutral_particle        ptk;
        ptk.mass = 1.;
        ptk.p = 1.;
        double dx, dy;
        for (t = vt->begin(); t != vt->end(); t++){
                ptk.phi = t->entry.phi;
                ptk.r = t->entry.r;
                ptk.s = t->entry.s;
                dx = t->exit.r*cos(t->exit.phi)-ptk.r*cos(ptk.phi);
                dy = t->exit.r*sin(t->exit.phi)-ptk.r*sin(ptk.phi);
                ptk.angle = atan2(dy,dx);
                instal.Trace(ptk);}
        charged_particle        cptk;
        cptk.charge = -1;
        cptk.mass = 1;
        cptk.p = 1;
        for (t = vt->begin(); t != vt->end(); t++){
                cptk.phi = t->entry.phi;
                cptk.r = t->entry.r;
                cptk.s = t->entry.s;
                dx = t->exit.r*cos(t->exit.phi)-cptk.r*cos(cptk.phi);
                dy = t->exit.r*sin(t->exit.phi)-cptk.r*sin(cptk.phi);
                cptk.angle = atan2(dy,dx);
                instal.Trace(cptk);}

        cptk.charge = 1;
        cptk.mass = 1;
        cptk.p = 1;
        for (t = vt->begin(); t != vt->end(); t++){
                cptk.phi = t->entry.phi;
                cptk.r = t->entry.r;
                cptk.s = t->entry.s;
                dx = t->exit.r*cos(t->exit.phi)-cptk.r*cos(cptk.phi);
                dy = t->exit.r*sin(t->exit.phi)-cptk.r*sin(cptk.phi);
                cptk.angle = atan2(dy,dx);
                instal.Trace(cptk);}

#ifdef DEBUG
        instal.print();
#endif
cout << " main stopped\n";
}
```