HERA B

Software Group

# UTA: Unified Tracking and Analysis Class Library

Jörg Rieling
MPI für Kernphysik Heidelberg

Oktober '98

# Contents

# 1 Introduction

## 1.1 What is UTA ?

UTA is a basic class library that is thought for particle tracking/reconstruction and hit/track data analysis. It:

- ...facilitates communication between different C++ modules by providing a set of interfaces.
- ...facilitates communication between developers by establishing a set of terms and useful abstractions/tools that enhances understanding.
- ...is extendable in a natural and consistent way if more complex tools/scenarios appear (vertically, specializations of existing classes can be inherited; horizontally, new class trunks can be attached).
- ...provides a uncomplicated arcitecture that allows to implement the interplay between tracks, track segments, hits and scatters of different kinds in a intuitive, natural way. The model is completely polymorphic, i.e. algorithms using UTA can handle heterogeneous lists of hits and tracks.
- ...implements functionality which can be re–used. It provides basic needs as e.g. a kalman filter for 2– and 3–d track representations, distance functions and intersection tests.
- ...last, but not least: UTA does neither depend on a special data representation, nor does it contain hit or geometry data by itself. It can be attached to any high–energy physics software environment.

The note is organized as follows:
in the first section the abstractions used in UTA are described together with the interplay of the four basic class trees. The second section contains an example that illustrates how to use UTA as a fundament for a more complex scenario.

This note concentrates mainly on the technical and architectural issues of UTA. There is no appendix that repeats function definitions of the header files. This documentation is written to the reader who wants to contribute to the `UTA` class library. But as far as necessary definitions and conventions are concerned, also the developer who normally does not read doc's and knows how to use things by oral tradition may find it useful.
It is required that you are familiar with the C++ programming language. This is not a C++ tutorial.

## 1.2 How to Get the Files

There is an official HERA–$B$ C++ repository under afs where you find all the files in the directory
`afs:/afs/desy.de/group/hera-b/cxx/repository`
. They are stored in the `cvs` format. You can check them out by doing the following:

1. to make cvs using the correct transfer protocol (i.e. `ssh`, which gets you the kerberos authentication), issue (or put into your `.profile`:
   `export CVS_RSH = ssh`
2. create an arte source code directory and check out the inc's and sources (the cvs command goes into one line):

   ```
   mkdir ~/developments; cd ~/developments
   cvs -d afs:/afs/desy.de/group/hera-b/cxx/repository
       co -r release_1_2 Uta include/Uta
   ```

3. you can now create your local makefile in `~/developments` and add new sources in (or below) that directory.

# 2 Architecture of the UTA Class Library

UTA consists of four orthogonal class trees. The interplay of these classes generates the functionality for different problem solutions in dealing with tracks and hits; such scenarios might e.g. be pattern recognition tasks, track fitting and track merging. UTA provides a set of virtual interfaces that facilitate the communication between the analysis software of different detector components. One cannot overestimate the value of properly chosen, flexible interfaces: *"interfaces are a company's most valuable resources"*[1].

## 2.1 Track Parameters

The first class hierarchy describes the *tracks parameters* or *parameter vectors*. They are the mathematical representation of a track. The vectors currently implemented in UTA are a set of numbers describing slopes, axis offsets (and a curvature variable, if necessary) together with a covariance matrix that relates the associated errors. A `UtaParameterSet` object is used as the central part of a track segment (see below).

### 2.1.1 Base Class

All track classes are derived from one base class `UtaParameterSet` (see fig. 2.1.2). The minimal (virtual) interface for all track representations looks as follows:

Every derived parameter set *must* provide a method
`void transport(UtaFloat zTo)`
that evaluates the track parameters at the given $z$–position.

Besides that, every derived parameter set *must* provide a method
`UtaFloat sensCoo ( UtaFloat const sine, UtaFloat const cosine ) const`
that returns the sensitive coordinate $x_s$ of the track representation if projected on the direction given by the angle $\alpha_{\rm st}$ =`atan2(si, co)`. More details on the stereo angle convention follow below.

Every derived parameter set can provide a (virtual) method
`void filter (UtaNode const & aNode)`
that updates the vector by adding the information of a measurement or scattering process by `aNode` (see below for more information about nodes).

Finally, every derived parameter set can provide a protected (virtual) method

```
void protectedPrint (ostream & stream) const
```
that generates specialized output if `*this` is directed to an `ostream`. This method is one part of a mechanism that emulates a polymorphic `operator<<`, i.e. special output for derived track classes is generated if a dereferenced base class pointer to a derived object is sent to an `ostream`. The other part is an overloaded `ostream operator<<` which is declared `friend` in `UtaParameterBase`:
```
friend ostream & operator << (ostream & stream, UtaParameterSet const &
set)
{
    set.protectedPrint(stream); return stream;
}
```
The same mechanism is used in the other class hierarchies and will not be mentioned again.

Note that no data members are specified in this abstract base class.

### 2.1.2 Derived Classes

Starting from `UtaParameterSet`, the inheritance tree falls into two branches: one for 2–dimensional vectors which are e.g. used for pattern recognition and one for 3–d tracks (see fig. 2.1.2). Specialization leads to tracks with momentum (for vertex fitting), with inverse momentum (for tracking/fitting), with interaction time as an additional parameter (for drift devices). The classes derived from `UtaParameterSet` are the only ones in UTA which contain data members: all values are stored as float–type variables inside the class. This is also valid for the covariance matrices.
[—*Note.* Some people may find it more handy to use dedicated matrix classes to describe the interplay of track vectors and nodes. Though such special classes are not used by UTA, you can easily place such algorithms inside the described architecture by inheriting from `UtaParameterSet` and `UtaNode`. —*end Note.*]

The classes `UtaPar2D` and `UtaPar3D` include ctors which construct the respective track vector from 2 rsp. 3 one–dimensional measurements `UtaHit1Strip` (see below). In case of a 3–d vector, the ctor inverts the measurement equation

$$m_i = (\cos\alpha_{st,i} - \sin\alpha_{st,i}, z_i\cos\alpha_{st,i}, -z_i\sin\alpha_{st,i})^T \cdot \vec{p}, \tag{2.1}$$
$$\vec{m} = \mathcal{M}\vec{p}, \tag{2.2}$$

where the track vector $\vec{p} = (x, y, t_x, t_y)$ is evaluated at $z = 0$. $m_i$ is the sensitive coordinate of the measurement at $z_i$. The ctor is fast because it uses analytical inversion.

Though expected and obvious to the experienced reader, it might be worthwhile to comment upon a few details:

- Access to the data members is given by access functions. To get e.g. the point of evaluation of a vector $x$, type `UtaFloat z = x.getZ()`. Please consult the header files for detailed information.

- In UTA, the convention that relates the stereo angle and the (signed) sensitive coordinate of a 3–dimensional track is

$$x_s(z) = x(z)\cos\alpha_{\mathrm{st}} + y(z)\sin\alpha_{\mathrm{st}}\,, \tag{2.3}$$

  where $x, y$ are the axis offsets if the track is evaluated at $z$. The projection of a 2–dimensional track vector to direction given by $\alpha_{\mathrm{st}}$ is not implemented by default in UTA. If the view of the vector does not agree with $\alpha_{\mathrm{st}}$ within a tolerance, the projection method terminates with an error. However, a user–provided specialized projection that accounts for detector shape and angles may override the default behaviour.

- The view of the *2–d parameter sets* can be changed, that is an instance of a `UtaPar2D` can be specified to be rotated w.r. to the $z$–axis by a certain angle $\alpha$ which is counted mathematically negative (just like the stereo angle, see above). To create a 2–d track vector $x$ which lies in the 2.5°–view, a ctor exists which takes the angle as an argument:
  `UtaPar2D x(2.5);`.
  Please consult the header files for a complete list of ctors.

- The *transport methods* which are implemented by default in `release_1_2` are based on the Kalman filter formalism [2].
  [—*Example.* For a 3–d vector the transport equation reads

$$\vec{p}_i,\ \mathcal{C}_i \ \longmapsto\ \vec{p}_f = \mathcal{F}_{if}\,\vec{p}_i,\quad \mathcal{C}_f = \mathcal{F}_{if}\,\mathcal{C}_i\,\mathcal{F}_{if}^T, \quad \text{where} \tag{2.4}$$

$$\mathcal{F}_{if} = \begin{pmatrix} 1 & 0 & z_f - z_i & 0 \\ 0 & 1 & 0 & z_f - z_i \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

  with

$$p(z_0) = \begin{pmatrix} x(z_0) \\ y(z_0) \\ m_x \\ m_y \end{pmatrix}, \tag{2.5}$$

$$\mathcal{C} \stackrel{\mathrm{def}}{=} \mathrm{cov}(p(z))\,. \tag{2.6}$$

  $i, f$ stand for *initial* rsp. *final*, and $m_x, m_y$ denote the slopes in $x$ and $y$. —*end Example.*]

- The filtering of a vector with a node (see below) depends on the type of the node. Therefore, the filter–method in the derived classes simply applies a correction `deltaP` to the current state vector and is therefore implemented as follows:
  ```
  void UtaPar3D::filter (UtaNode const & aNode)
  {
  *this += aNode.deltaP (*this);
  } .
  ```
  The filter correction `deltaP` is implemented in the node–hierarchy. It certainly uses the UTA–wide angle convention eq.2.3.

- Every class provides a set of useful ctor's and some basic arithmetic and logical operators such as `operator+`, `operator=`, `operator+=`, `operator==`, `operator!=`.


Figure 2.1: The parameter vector class hierarchy. U stands for public, O for protected visibility.

## 2.2 Track Segments

A track segment represents a part of a particle trajectory. It could e.g. be reconstructed in a subdetector, it could be the result of a merging operation between two smaller segments; or, a segment might be slighly by performing a refit. The `UtaSegment` hierarchy encapsulates the higher–level operations that are typically performed with parameter sets. As in the case of the `UtaParameterSet` hierarchy, a base class defines a purely operational interface with higher–level operations. The base class contains no data members but just interfaces which will be implemented by inherited classes. The derived classes contain a `UtaParameterSet` as a data member which stores the state of the segment (see fig. 2.2).
The `UtaSegment` hierarchy in its present state does not implement a functionality. As will be shown, it rather forms a part of an architecture into which functionality like matching, merging, re–fitting etc. can be built in a natural way.

### 2.2.1 Base Class

The base class `UtaSegment` foresees a method

```
void refit (UtaFloat zTo)
```

that re–calculates the state vector of a segment at a given z based on all the information associated with that segment. To provide this functionality, an instance of a derived class will typically maintain a list of (pointers to) measurements and crossed matter.

There is a method

```
void propagate (UtaParameterSet const & start)
```

that takes a start vector as an argument and finds all nodes that can be associated with the segment. The meaning of "can be associated" is given bu derived classes. This virtual table entry can e.g. be used for pattern recognition purposes, but also for hit–removal or recollection for a complete track.

The method

```
UtaFloat const_match (UtaFloat z, <UtaSegment> const & other) const;
```

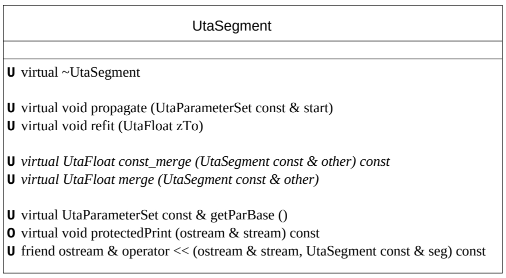



Figure 2.2: The track segment base class. `U` stands for public, `O` for protected visibility.

returns the $\chi^2$ value if one merges `*this` with another segment at the specified $z$–position. The method
`UtaFloat match (UtaFloat z, <UtaSegment> const & other) const;`
returns the $\chi^2$ value if one merges `*this` with another segment of type `<UtaSegment>` at the specified $z$–position and changes `*this` according to the new information.
[—*Note.* If one overloads `const_match` and `match` in the base class with other segment types, a 'function space' is span up by the combination of overloading and virtuality where every merging calculation between arbitrary parameter set has it's natural place. Note that these calculations can further be overridden if wished. This technique is usually called *double dispatch*. In UTA it is used to implement the relation between parameter sets of different type and nodes of different type (see below). —*end Note.*]

Access to the track parameter data members *must* be provided by a reference to the track vector:
`virtual UtaParameterSet const & getParBase () const = 0;`.
The derived classes will additionally provide a method
`<track_vector_type> & getPar () const;`,
which returns a reference to the complete track vector object and not only to the base interface of it.

Just like the state vector base class, `UtaSegment` provides a virtual `operator<<` (see above).

[—*Note.* Note that also a collection of segments is in turn a segment which can be e.g. refitted. A class `MatchedSegments` could contain a vector of segments which would be used in the method `MatchedSegments::refit(UtaFloat zTo)`. Together with the double dispatch pattern for merging described in the foregoing annotation, an elegant way is thus provided to organize the interplay of the reconstruction programs in different detector components. —*end Note.*]

## 2.3 Shapes

A shape encapsulates the geometrical expertise that describes the relation between a 2–dimensional or 3–dimensional body and a track vector. As track vectors are a part of a track segment, a `UtaShape` will be a part of a node (see below).

### 2.3.1 Base Class

To be instantiable, an inherited class has to provide a method
`UtaFloat zPos() const;`
that describes where in the parametrizing $z$–direction the body is located (see fig. 2.3). Note that no assumption is made how the $z$–position is precisely defined. The convention is due to inherited classes, and the following methods will be implemented by them based

on this (and other) conventions.

An inherited class has to provide the two overloaded methods
`bool shapeIsHit (UtaParXD & vec) const;` and
`bool const_shapeIsHit (UtaParXD const & vec) const;  X=2, 3.`
They test if a given parameter `vec` intersects the shape. The difference between the two methods is that the former transports `vec` to the $z$–position of the body, whereas the latter lets `vec` unchanged and uses a local copy for its calculation.
[—*Note.* As can be seen from the overloading of the above methods, the present form of the shape base class foresees only straight–line tracks. If UTA shall be used in a magnetic field, it is wise to extend the base class interface by overloading these methods further with e.g. an `UtaPar3DInvMom`. To ensure the same scope, this must be done in `UtaShape`. —*end Note.*]

An inherited class has to provide the overloaded method
`UtaParXD inPar (UtaParXD const & vec) const;   X=2, 3.`
It returns the entry point of a given vector into the shape.

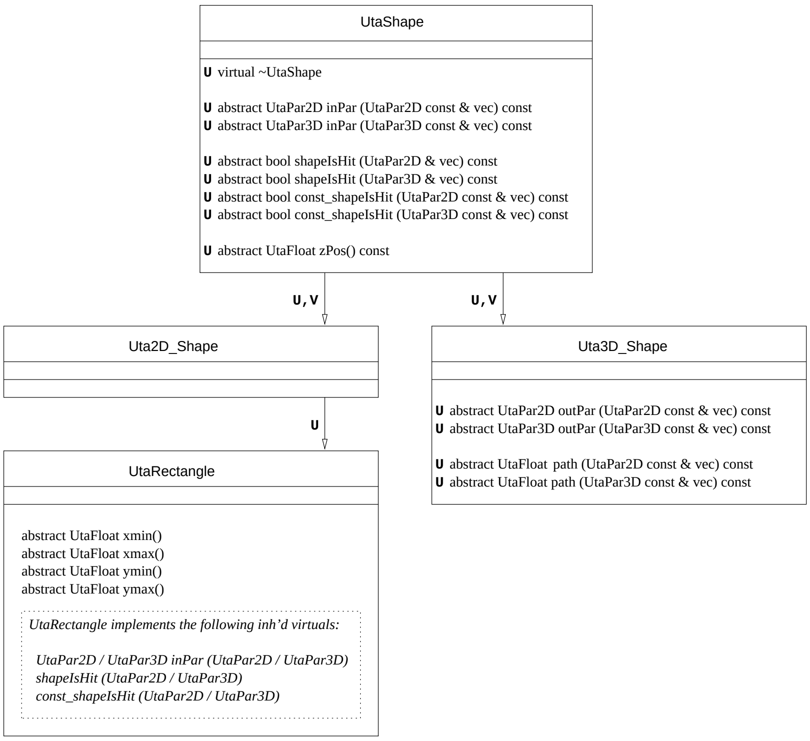



Figure 2.3: The shape hierarchy. `U` stands for public, `O` for protected visibility.

### 2.3.2 Derived Classes

Starting from `UtaShape`, the inheritance tree falls into two branches: one for 3–dimensional shapes and one for 2–dimensional shapes, the latter being a simple (and fast) approximation of 3–d shapes. A track intersecting a 3–d shape certainly has also an exit point, and a certain path length inside the body. Thus, two overloaded abstract methods
`UtaParXD outPar (UtaParXD const & vec) const;` and
`UtaFloat path (UtaParXD const & vec) const;` X=2, 3.
are introduced here.

The class `UtaPar2D` does not add new interfaces, it *is* the base class and has just been introduced for symmetry.

To provide an example of how to fill such an interface with life (and because I needed the functionality), a class `UtaRectangle` was written which describes an unrotated rectangle. The abstract interfaces
`UtaFloat xmin() const; UtaFloat xmax() const; UtaFloat ymin() const;`
`UtaFloat ymax() const;` specify the boundary of the rectangle and complete the definition of this shape type. The methods `inPar`, `const_shapeIsHit` and `shapeIsHit` are implemented in this `UtaRectangle`; they use the still abstract interface.
[—*Example.* The method `UtaRectangle::shapeIsHit (UtaPar3D & p)` is implemented as follows:

```
bool UtaRectangle::shapeIsHit (UtaPar3D & p) const
{
     p.transport (zPos());
     UtaFloat const xtrack = p.getX();
     UtaFloat const ytrack = p.getY();
     return xtrack>xmin() && xtrack<xmax() &&
            ytrack>ymin() && ytrack<ymax();
}
```

—*end Example.*]
To adapt a `UtaRectangle` to the specific data representation of an experiment, also the geometry interface has to be implemented.
[—*Example.* A rectangle class based on UTA which is specific for the HERA–*B* software environment looks as follows:

```
  # include <arte/ArteAll.hh>
# include <arte/ArteFlag.h>
# include <Uta/UtaRectangle.hh>

class SusiGedeRectangle :  public UtaRectangle
{
public:
```

```
      // -- ctors, dtor
      [...]
      // -- public virtual geo interface
public:
      UtaFloat xmin() const { return gede->sxmn; }
      UtaFloat xmax() const { return gede->sxmx; }
      UtaFloat ymin() const { return gede->symn; }
      UtaFloat ymax() const { return gede->symx; }
      UtaFloat zPos() const { return gede->zmin; }

protected:
      ArtePointer<GEDE> gede;
};
```

The data member `gede` is a pointer to a row of a data table `GEDE` which contains the geometrical information. Note that changes in the way how data are stored in the `GEDE` table are of no concern for a user of `SusiGedeRectangle` because the virtual interface hides the actual implemenation. —*end Example.*]

## 2.4 Nodes

A *node* is an entity which can alter the state of a parameter vector, i.e. any wall that scatters, but also sensitive planes.

### 2.4.1 Base Class

The base class `UtaNode` defines overloaded virtual functions that describe this alteration numerically for every parameter set derived from `UtaParameterSet`:
for every parameter vector type <`vec_type`> there exists a virtual method

```
<vec_type> deltaP ( <vec_type> const & aTrack ) const;
```

which describes the change that the vector `aTrack` undergoes if the information of `*this` is added to it. In case of a scattering wall, this alteration will just add an uncertainty to the covariance matrix of `vec`, in case of a measurement also the vector itself will be changed. Both is implemented in UTA and will be described below.

A node is associated with a certain geometric shape. In case of a measurement, this may e.g. be the shape of the detector which measured the track, in case of a scatterer it is the block of matter. `UtaNode` therefore defines a method

```
UtaShape const & getShapeBase();
```

which must be implemented by every instantiable inherited class. It returns the `UtaShape` part of the particular shape associated with `*this`.

The class `UtaNode` foresees an abstract method

```
UtaFloat getMsAngle2() const;
```

that describes the multiple scattering properties of a node. The precise meaning of `getMsAngle2()` depends on the type of the shape in `*this`. I define it as follows: In case of a shape derived from `Uta2D_Shape`, it returns the mean squared multiple scattering angle for a particle of 1 GeV if it intersects the shape in a rectangular way. In case of a shape derived from `Uta3D_Shape`, it returns the mean squared multiple scattering angle per unit length for a particle of 1 GeV. To compute the influence of the scattering process for a 3–d node, the method `Uta3D_Shape::path(...)` thus has to be used.

`UtaNode` defines and implements an overloaded, **non–virtual** method
`bool nodeIsHit (UtaParXD const & vec) const;`
which is just a shortcut to `getShapeBase().const_shapeIsHit(vec)`.

Finally, as in the parameter set and segment class trees, the node hierarchy provides a virtual `operator<<` mechanism (see above).

### 2.4.2 Derived Classes

To describe the multiple scattering of a thin wall, a class `UtaThinWall` is derived from `UtaNode`. It does not introduce new interfaces but implements the inherited `deltaP` methods from `UtaNode` by simply calculating the mean angular uncertainty imposed on `vec` which depends on the intersection angle of `vec` and its momentum.
[—*Note.* A class that describes scattering in thick walls, i.e. that calulates the effect on axis offsets and the off-diagonal elements of the covariance matrix is not included in `release_1_2`. It would certainly be inserted analogously to `UtaThinWall`. —*end Note.*]

The class `UtaHit` is the base class for every type of hit. A hit has the property to have (signed) a residual distance to a given parameter vector. This is defined by the overloaded method
`UtaFloat distance (UtaParXD const & vec) const;`.
More elaborated distance concepts like e.g. a vector difference are not foreseen but can be introduced by specializing `UtaHit`.
[—*Example.* To calculate the distances between a track `<track_par_type> track` and a hit `<hit_type> hit`, a user will invoke
`UtaFloat residual = hit.distance(track);`.
Because of polymorphism, this works for heterogeneous lists of hits like e.g. tube–like drift chamber hits and pixel hits. —*end Example.*]

One of the most heavily employed types of hits is the one–dimensional measurement of a track as e.g. provided by a silicon strip detector. A class `UtaHit1Strip` represents it. It introduces an abstract interface that describes its fundamental properties:
`UtaFloat sensCoo() const;`
is the sensitive coordinate which is related with the stereo angle $\alpha_{\rm st}$ by equation 2.3. The methods
`UtaFloat sinStAngle() const; UtaFloat cosStAngle() const;`


Figure 2.4: The node hierarchy. `U` stands for public, `O` for protected visibility.

are the sine rsp. the cosine of $\alpha_{\mathrm{st}}$. The method
`UtaFloat acc() const;`
returns the measurement accuracy.

In `UtaHit1Strip`, the influence of the measurement on the track vector is calculated using the Kalman filter method (see e.g. [2]). The filter equation reads:

$$\vec{p}_{f,\mathrm{filt}} = \vec{p}_f + \mathcal{K}\,(m - \mathcal{H}\,\vec{p}_f) \tag{2.7}$$

$$\mathcal{C}_{f,\mathrm{filt}} = (1 - \mathcal{K}\,\mathcal{H})\,\mathcal{C}_f, \text{ where} \tag{2.8}$$

$$\mathcal{K} \stackrel{\mathrm{def}}{=} (\sigma_f^2 + \mathcal{H}\,\mathcal{C}_f\,\mathcal{H}^T)^{-1} \cdot \mathcal{C}_f\,\mathcal{H}^T \qquad \text{(Kalman gain matrix)}$$

$$\mathcal{H} \stackrel{\mathrm{def}}{=} (\cos\alpha_{st,f},\ -\sin\alpha_{st,f},\ 0,\ 0) \qquad \text{(projector on sens. coordinate)}\,.$$

For any track par type and any node type, the differences $\vec{p}_{f,\mathrm{filt}} - \vec{p}_f$ and $\mathcal{C}_{f,\mathrm{filt}} - \mathcal{C}_f$ are returned by the call to deltaP(...).

[—*Note.* The methods `<vec_type> deltaP ( <vec_type> const & aTrack ) const` are not used by the normal developer. They are used by the filter methods in the track parameter hierarchy. These methods are implementations of the entry `void filter (UtaNode const & aNode)` in the virtual table of `UtaParameterSet` and look as follows:

```
void <vec_type>::filter (<node_type> const & aNode )
{
  *this += aNode.deltaP (*this);
}.
```

—*end Note.*]

[—*Note.* As mentioned, the combination of virtuality (in `UtaParameterSet`) and overloading (in `UtaNode`) is called *double dispatch* and a completely transparent use of nodes and track vector types. For example the track fitting code

```
for (vector<UtaNode*>::iterator hit=hitList.begin();
     hit!=hitList.end(); ++hit)
{
  track.propagate ((*hit)->z);
  track.filter (*hit);
}
```

works no matter if `track` is a `UtaPar2D` or a `UtaPar3DInvMom`, but also for heterogeneous lists of nodes, e.g. walls and hits from different detector types.

Please note that

- ...it would have also been possible to implement double dispatch by overloading in `UtaParameterSet` and using virtuality in the `UtaNode` hierarchy. However, it was felt that the number of track representations was less likely ro increase in the future than the number of node types. Consequently, the previously described implementation was chosen because this reduces the number of overloaded functions in the base class.
- ...the track parameter hierarchy in fig. 2.1.2 is chosen in such a way that the number of `deltaP` methods that have to be explicitely coded is minimized by inheritance.

—*end Note.*]

# 3 Extending UTA: a Case Study

As an example how to attach UTA to already existing data strucutres and how to re–use the already implemented algorithms, a node class for pattern recognition purposes is described in this section. It is taken from a class library `SusiCL` that has been successfully used in the online track reconstruction of the silicon vertex detector of the HERA–*B*–experiment.

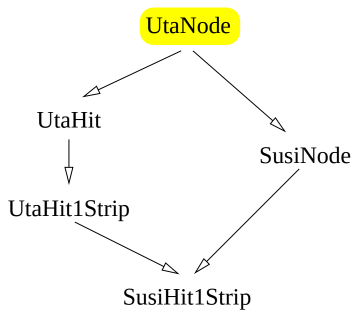
## 3.1 A Node Class for Pattern Recognition

Local pattern recognition algorithms often deal with more than one type of measurement: in addition to normal one–dimensional hits one might also use crossing points, i.e. measurements that consist of more than one strip. At HERA–*B*, the representation for measurements in the silicon detector are 1–d hits which are stored in a data table `HITB`. That means: more complicated hit objects should maintain a list of `HITB` row indices to allow direct access to the underlying data structures if required. Obviously, this requirement is also valid for any object type that is composed of more than one `UtaNode`.

Besides that, it is often necessary to emulate a virtual ctor. In C++, this is normally done by introducing a virtual `clone` membership function in the base class that returns a reference to an object of the respective type.

Given the above, it is obviously necessary to extend the virtual `UtaNode` interface by inheriting a `SusiNode` class in which the above mentioned functionalities are foreseen.

On the other hand, one wants to re–use the code that has already been implemented in e.g. a `UtaHit1Strip`. This leads to the following inheritance graph:




[—*Note.* The use of MI is clean from the architectural point of view in this case, because only interfaces are combined, no data members are involved. Compare `www.peerdirect.com/resources/gotw037.html`. With its `interface` concept JAVA

provides a more elegant mechanism for this type of problem. —*end Note.*]
[—*Note.* The part of the virtual table introduced in the `UtaHit` branch of the above graph can naively be used if one has a `SusiNode*` pointing to an instance of a `SusiHit1Strip`. This is non–trivial but required by the standard (see section 10 in [3]). —*end Note.*]

Implementing the above graph is a single line of C++ code. What remains to be done is to connect a `SusiHit1Strip` to the data structures used at HERA–*B*. In code, that is:

```
 Inher-Lvl-1 --25%---------| Inher-Lvl-2 --All---------| Blank-3 --All------------
 // -- other access methods:
 //  - implementation of inherited virtuals from UtaHit1Strip
public:
  UtaFloat sensCoo     () const { return  hit->x; }
  UtaFloat sinStAngle () const { return  gedeSine   (ArtePointer<GEDE>(hit->gede)); }
  UtaFloat cosStAngle () const { return  gedeCosine(ArtePointer<GEDE>(hit->gede)); }
  UtaFloat acc         () const { return  ArtePointer<GEDE>(hit->gede)->acc;   }
```

, where the fct implementations use a data member `ArtePointer<HITB>` that point into the appropriate row of the `HITB` table. The global functions `gedeSine` and `gedeCosine` are used to calculate the required absolute stereo angles from the values that are stored in the detector description table `GEDE`. They are necessary because the HERA–*B* software uses different geometry conventions from one version to another.

Analogously to the node classes, a similar inheritance diagram was used for the track segments. The track vectors hierarchy was re–used unextendedly. To attach the shapes to the geometry data representation used at HERA–*B*, shape classes were inherited which look like the one on page 2.3.2.

Let me conclude this section by copy–pasting a code example taken from a simple pattern recognition program. The job is to propagate a track vector `track` through a spectrometer towards the target. On its way, the vector comes into the vicinity of one–dimensional hits. If the track vector approaches these hits close enough, the rsp. measurement information shall be added to the vector, giving rise to a change of the track vector. This is a central part in a classical local track reconstruction algorithm.

Using the SUSI class library, the algorithm is quickly written:

```
for (vector<SusiHit1Strip>::const_iterator
     it=hits.begin(); it!=hits.end(); ++it)
{   track.transport((*it).getZ());
  if (fabs((*it).distance(track)) < .02)
    track.filter(it);
}
```

The vector `hits` is a sorted in $z$.

# Bibliography


[1] G. Booch: OOA/D with Applications, 1994

[2] R. Frühwirth: Application of Kalman Filtering to Track and Vertex Fitting, NIM A262 444, 1987

[3] Accredited Standards Comittee, X3, INFORMATION PROCESSING SYSTEMS: Working Paper for Draft Proposed International Standard for Infromation Systems — Programming Language C++, 2. Dec 1996

[4] The HERA–B collaboration: An Experiment to Study CP Violation in the $B$ System Using an Internal Target at the HERA Proton Ring (Technical Design Report), DESY–PRC 95/01, 1995

## Ack's

Dank. An alle. Besonders aber an Christoph. Aus unseren wirklich fruchtbaren Debatten ist UTA entstanden.

## Version and Change History

ID: `(dt=30.10.1998-14:30 bdir=/home/joerg/utadoku.newer/utadoku/tex src=uta.tex intro.tex architecture.tex odir=../pic idir=../inc u=joerg@elmo.desy.de)`:
creation (Jörg Rieling).