STAR

STAR Offline Library Long Writeup

StEvent

User Guide and Reference Manual

for Version 2

Revision: 2.80

Date: 2004/03/30 15:59:43

Contents

1 Introduction

This document contains the User Guide and Reference Manual for StEvent version 2. Like the new version of StEvent this documentation is a complete rewrite and supersedes all documentation with revision number 1.xx. All code and documentation for the new version has a cvs version number of greater or equal 2.00.

In this document more emphasis is put on the User Guide while the Reference Manual part is kept shorter in terms of description of usage. As StEvent changes this document will change accordingly and you should always check that the revision number of the document matches the one in the repository.

Version 2 of StEvent contains significant changes as compared to the previous versions. Part of the changes were made to cope with the modification of the DST format in Fall of 1999, others were made to overcome shortcomings in the previous implementation. This version is also more flexible in terms of extendibility to allow future track and vertex models to be incorporated easily. The current implementation is also meant to be used further upstream of the analysis, i.e. in the reconstruction phase. As a consequence the model itself became slightly more complex in terms of navigation and structuring.

In order to explain the model in practical terms many diagrams and plots were included in this document. Some of them show class diagrams using the Unified Modelling Language UML. A brief introduction to UML is given in Appendix A.

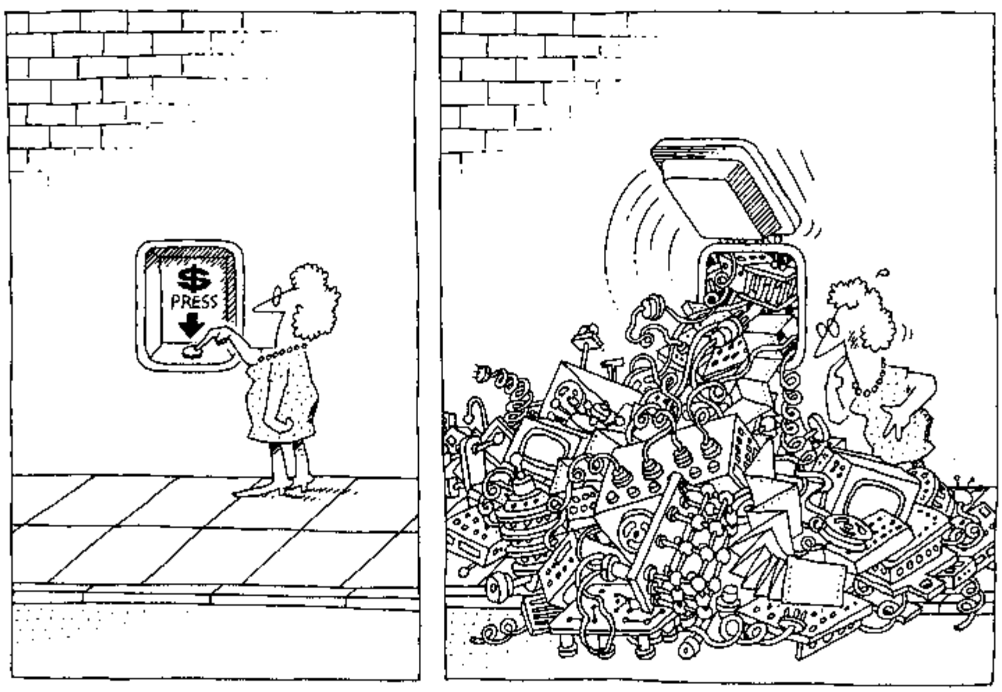

Figure 1.1: The task of the software development team is to engineer the illusion of simplicity.

Part I

User Guide

2 Basics

2.1 Header Files

The amount of header files included in the StEvent classes was minimized to decrease dependencies between the various classes and where ever possible forward declarations were used. This is especially true for the `StEvent` class itself and it is therefore *not* sufficient to include `StEvent.h` only. Many more header files would have to be included. This is very good for the developers since turnaround times are minimized but obviously bad for the users for it would be very cumbersome to each time figure out which header files one might need and which not. Therefore are *all* header files which are needed to use every little bit of StEvent contained in one single header file named `StEventTypes.h`. The disadvantage of this approach is that every time one StEvent class changes you have to recompile all your code, even if the changed class is not used. This, however, should not happen too often and it by far more convenient to deal with on header file only.

To summarize: All you need when using StEvent is to include `StEventTypes.h` and you are all set.

2.2 Enumerations and Constants

StEvent uses a lot of enumerations for all types of purposes. This is much more type-safe then using simple integer numbers and makes the code more readable. All enumerations used in StEvent are defined in `StEnumerations.h`. For users convenience some non-StEvent header files as `StDetectorId.h`, `StVertexId.h` and `StTrackMethod.h` are also included therein. To remind you of the names and

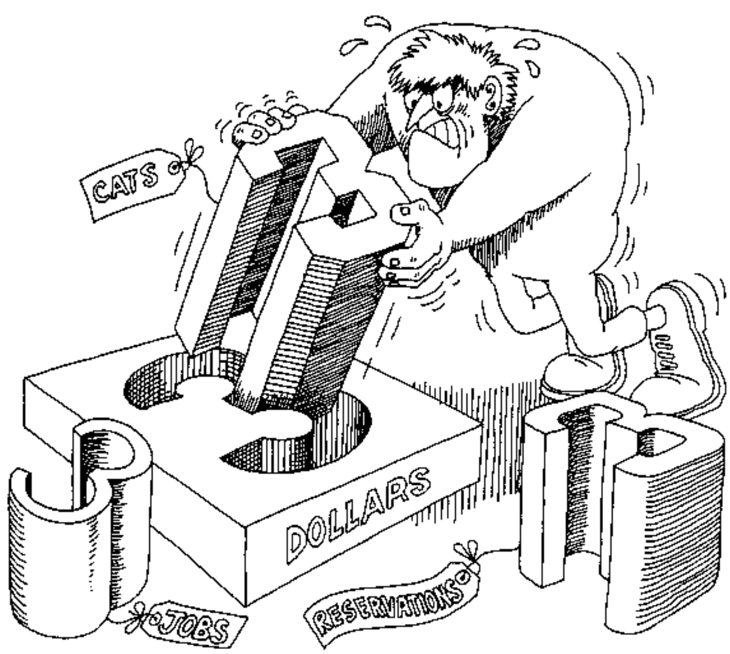

Figure 2.1: Strong typing avoids mixing abstractions.

save you the time to look them up up every you need one they are all listed below:

```
enum StBeamDirection        {east = 0
```

```
                             yellow = 0,
                             west = 1,
                             blue = 1};

enum StBeamPolarizationAxis {transverse, longitudinal};

enum StChargeSign           {negative, positive};

enum StTrackType            {global, primary, tpt, secondary};

enum StTrackModel           {helixModel, kalmanModel};

enum StDetectorId           {kUnknownId,
                             kTpcId,
                             kSvtId,
                             kRichId,
                             kFtpcWestId,
                             kFtpcEastId,
                             kTofId,
                             kCtbId,
                             kSsdId,
                             kBarrelEmcTowerId,
                             kBarrelEmcPreShowerId,
                             kBarrelSmdEtaStripId,
                             kBarrelSmdPhiStripId,
                             kEndcapEmcTowerId,
                             kEndcapEmcPreShowerId,
                             kEndcapSmdEtaStripId,
                             kEndcapSmdPhiStripId,
                             kZdcWestId,
                             kZdcEastId,
                             kMwpcWestId,
                             kMwpcEastId,
                             kTpcSsdId,
                             kTpcSvtId,
                             kTpcSsdSvtId,
                             kSsdSvtId,
                             kPhmdId,
                             kPhmdCpvId};

enum StVertexId             {kUndefinedVtxId,
                             kEventVtxId,
                             kV0VtxId,
                             kXiVtxId,
                             kKinkVtxId,
                             kOtherVtxId};

enum StDedxMethod           {kUndefinedMethodId,
                             kTruncatedMeanId,
                             kEnsembleTruncatedMeanId,
                             kLikelihoodFitId,
                             kWeightedTruncatedMeanId,
```

```
                             kOtherMethodId};

enum StTrackFittingMethod   {kUndefinedFitterId,
                             kHelix2StepId,
                             kHelix3DId,
                             kKalmanFitId,
                             kLine2StepId,
                             kLine3DId
                             kL3FitId};

enum StTrackFinderMethod    {svtGrouper,
                             svtStk,
                             svtOther,
                             tpcStandard,
                             tpcOther,
                             ftpcConformal,
                             ftpcCurrent,
                             svtTpcSvm,
                             svtTpcEst,
                             svtTpcPattern,
                             l3Standard};

enum StRichPidFlag          {eNoMip,
                             eFastEnough,
                             eLightOnPadPlane};

enum StRichHitFlag          {eDeconvoluted=1,
                             eMip=2,
                             eSaturatedPad=4 ,
                             ePhotoElectron=8,
                             eMultiplyAssignedToRing=16,
                             eAssociatedMip=32,
                             e1SigmaPi=64,
                             e2SigmaPi=128,
                             eInConstantAreaPi=256,
                             eInAreaPi=512,
                             eAssignedToRingPi=1024,
                             e1SigmaK=2048,
                             e2SigmaK=4096,
                             eInConstantAreaK=8192,
                             eInAreaK=16384,
                             eAssignedToRingK=32768,
                             e1Sigmap=65536,
                             e2Sigmap=131072,
                             eInConstantAreap=262144,
                             eInAreap=524288,
                             eAssignedToRingp=1048576};

enum StPwg                  {generic,
                             ebye,
                             hbt,
                             highpt,
```

```
                          pcoll,
                          spectra,
                          spin,
                          strangeness};
```

Note that often the enumeration type names (e.g. `StTrackType`) are used as argument types. The strong C++ type checking rules ensures the proper use of the enumeration constants already during compilation.

Another important set of constants should be mentioned here as well, namely the physical constants defined in `PhysicalConstants.h`. There are too many to be listed here but you should make yourself familiar with what constants are available. You will find the header file in the *StarClassLibrary* (see Sec. 2.8). In order to define the units of the various physical constants another set of constants defined in `SystemOfUnits.h` is used (also from *StarClassLibrary*). The latter is described in section 2.3.3.

2.3 Conventions

2.3.1 Numbering Scheme

All numbering follows *strictly* the C/C++ convention, i.e. the first element in an array has the index 0. This is valid for all container, collections and lists. Here it is important to remember that many (but not all) official STAR numbering schemes start counting at 1. Examples are TPC sectors and padrows, SVT barrels, layers, ladders and wafers. Do not forget to subtract 1 when using this scheme for addressing elements in a container.

TPC, SVT and FTPC hits return their hardware address in STAR units. In order to select the hit container in which a hit `h` is stored you must write:

```
evt->tpcHitCollection()->sector(h.sector()-1)->padrow(h.padrow()-1).hits();
```

Many of STARs numbering schemes were defined when FORTRAN was the main programming language and C/C++ played only a minor role. Again, the only place where you have to deal with these conventions is when you use one of the following methods:

- `StTpcHit::sector()`
- `StTpcHit::padrow()`
- `StFtpcHit::sector()`
- `StFtpcHit::plane()`
- `StSvtHit::layer()`
- `StSvtHit::ladder()`
- StSvtHit::wafer()
- `StSvtHit::barrel()`
- `StTrackTopologyMap::hasHitInRow(int)`
- `StTrackTopologyMap::hasHitInSvtLayer(int)`

See the corresponding reference sections for more.

2.3.2 References and Pointers

Many methods (or member functions) or StEvent classes return objects by *reference* or by *pointer*. This is sometimes confusing but there is a idea behind this. Whenever an object is returned by reference it is guaranteed to exist. No questions asked. If the object is a container it might be empty, i.e. it has zero size, but you ask for it you get it. Objects returned by pointer, however, are *not* guaranteed to exist. You might get a `NULL` pointer back. It is always a good idea to check if you really get what you asked for. Dereferencing a `NULL` pointer can be painful.

As you will see in the reference section many methods are provided in two versions: a constant and a non-constant version. Don't worry about the differences. The compiler will always choose the proper version.

2.3.3 Units

All physics quantities in StEvent are stored using the official STAR units: cm, GeV and Tesla. Angles are given in radians[1] In order to maintain a coherent system of units it is recommended to use the definitions in `SystemOfUnits.h` from the *StarClassLibrary*. They allow to 'assign' a unit to a given variable by multiplying it with a constant named accordingly (centimeter, millimeter, kilometer, Tesla, MeV, ...). The constants ensure that the result after the multiplication follows always the STAR system of units.

The following example illustrates their use:

```
double a = 10*centimeter;
double b = 4*millimeter;
double c = 1*inch;
double E1 = 130*MeV;
double E2 = .1234*GeV;

//
//   Print in STAR units
//
cout << "STAR units:" << endl;
cout << "a = " << a << " cm" << endl;
cout << "b = " << b << " cm" << endl;
cout << "c = " << c << " cm" << endl;
cout << "E1 = " << E1 << " GeV" << endl;
cout << "E2 = " << E2 << " GeV" << endl;

//
//   Print in personal units
//
cout << "\nMy units:" << endl;
cout << "a = " << a/millimeter << " mm" << endl;
cout << "b = " << b/micrometer << " um" << endl;
cout << "c = " << c/meter << " m" << endl;
cout << "E1 = " << E1/TeV << " TeV" << endl;
cout << "E2 = " << E2/keV << " keV" << endl;
```

The resulting printout is:

[1] Note, that here StEvent deviates from STAR guidelines where degrees are declared the official units.

```
STAR units:
a = 10 cm
b = 0.4 cm
c = 2.54 cm
E1 = 0.13 GeV
E2 = 0.1234 GeV

My units:
a = 100 mm
b = 4000 um
c = 0.0254 m
E1 = 0.00013 TeV
E2 = 123400 keV
```

Further documentation can be found in the *StarClassLibrary* manual (see Sec. 2.8).

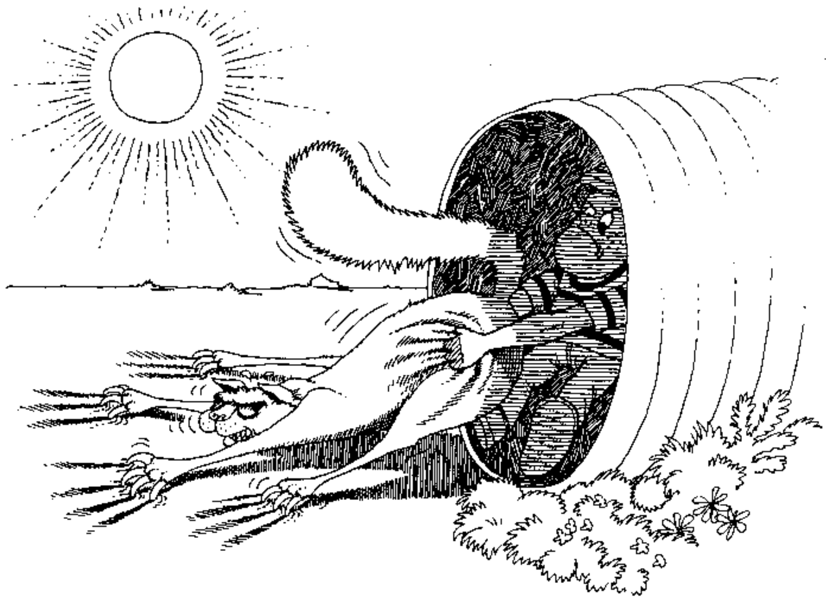

Figure 2.2: Persistence saves the state and class of an object across time or space.

2.4 Persistence and ROOT

All StEvent classes inherit from `StObject` which itself inherits from `TObject`. During the build of StEvent all classes run through `rootcint`. This adds the following features:

1. All StEvent classes can be used on the `root4star` command line.
2. Almost all StEvent classes are persistent capable, i.e. they can be stored in ROOT files.

As usual each coin has two sides. The disadvantage of this is that we cannot use some features of the ANSI/ISO C++ and from the Standard C++ Library as:

- type bool
- templates
- STL containers and algorithms
- namespaces

This however applies for the header files only. Source files are not processed via `rootcint` and therefore all the stuff mentioned above can be used. And indeed in the implementation of various StEvent classes we make heavily use of the STL.

ROOT uses typedefs for the built-in standard C++ types. This is pretty confusing but has a good reason when it comes to persistence. This way one can guarantee the same size (number of bytes) for the types independent of the platform. The ANSI/ISO standard only requires that: `char` ≤ `short` ≤ `int` ≤ `long` ≤ `long long` and `float` ≤ `double` ≤ `long double`.

The types used in StEvent are defined as follows:

```
typedef char           Char_t;      //Signed Character 1 byte
typedef unsigned char  UChar_t;     //Unsigned Character 1 byte
typedef short          Short_t;     //Signed Short integer 2 bytes
typedef unsigned short UShort_t;    //Unsigned Short integer 2 bytes
typedef int            Int_t;       //Signed integer 4 bytes
typedef unsigned int   UInt_t;      //Unsigned integer 4 bytes
typedef long           Long_t;      //Signed long integer 4 bytes
typedef unsigned long  ULong_t;     //Unsigned long integer 4 bytes
typedef float          Float_t;     //Float 4 bytes
typedef double         Double_t;    //Float 8 bytes
typedef unsigned char  Bool_t;      //Boolean
```

Note that since April 2001 the type `long` (`Long_t`) is not supported by ROOT any more and cannot be used for declaration of persistent data members.

This is fine and good but there is absolutely no reason to use them in code and function declarations. Even worse this can have disadvantages when it comes to calls to system functions and speed. It also makes code less portable and readable. Don't use them only because you see them used in StEvent. They are only used for the declaration of the persistent data members.

2.5 Container and Iterators

Version 2 of StEvent comes with a new naming scheme for containers. All containers used in StEvent store objects by pointer. Technically they are all vectors and therefore allow random-access as in

```
pointer_to_object = container[i];
```

that is they are ordered collections. There are two different types of containers, so called structural and non-structural containers. What that means is rather simple. Structural containers *own* the objects they contain the others not. If you delete a structural container all objects stored in it get deleted as well.

- All **s**tructural **vec**tors which store **p**oin**t**e**r**s carry the prefix **`StSPtrVec`**.
- All other **vec**tors which store **p**oin**t**e**r**s carry the prefix **`StPtrVec`**.

That's simple. To complete the name we append the type of objects they contain and we are done. Hence a structural container which holds objects (or better pointer to objects) of type `StTrackNode` is named `StSPtrVecTrackNode`. The `St` prefix of the class is always omitted.
In practice it makes little difference if you are using a structural or non-structural collection. Their interface is the same and they act they same. The secret lies in their implementation. If you create a container by your own you should always use the non-structural containers. Those you can create and delete without doing StEvent any harm. Never delete a structural container unless you stand with your back to a wall and a sharp knife on your throat.

All containers used in StEvent are defined in the `StContainers.h` header file and are based on `StArray` which was written by Victor Perevoztchikov. Currently the following containers are in use:

```
StPtrVecHit
StPtrVecTrack
StPtrVecTrackPidTraits
StSPtrVecFtpcHit
```

```
StSPtrVecKinkVertex
StSPtrVecPrimaryTrack
StSPtrVecPrimaryVertex
StSPtrVecSvtHit
StSPtrVecTpcHit
StSPtrVecTrack
StSPtrVecTrackDetectorInfo
StSPtrVecTrackNode
StSPtrVecTrackPidTraits
StSPtrVecV0Vertex
StSPtrVecXiVertex
```

All containers are based on modified ROOT collections. They allow to make StEvent persistent. They good thing with `StArray` is that all those containers offer an almost ANSI/ISO compatible interface. This means that *both* container classes provide the essential methods listed below. Replace `ClassName` with any StEvent class one might find in a container.

Public Constructors

`StPtrVecClassName();`
`StSPtrVecClassName();`
Constructs an instance with zero length.

`StPtrVecClassName(unsigned int nelem);`
`StSPtrVecClassName(unsigned int nelem);`
Constructs an instance with length `nelem`.

`StPtrVecClassName(const StPtrVecClassName& vec);`
`StSPtrVecClassName(const StSPtrVecClassName& vec);`
Copy constructor. Structural containers copy also the objects they contain.

Public Member Functions

`void push_back(const StClassName *pobj);`
Adds object pointed to by `pobj`. If the container is not large enough it will automatically resize.

`unsigned int size() const;`
Returns the current size of the container, i.e. the number of stored elements.

`void resize(unsigned int nelem);`
Resizes the collection to size `nelem`.

`void clear();`
Deletes all elements. If the container is a structural container all objects it holds get deleted.

`bool empty() const;`
Checks for zero size.

`const StPtrVecClassNameIterator begin() const;`
`const StSPtrVecClassNameIterator begin() const;`
Returns iterator to the the first element in the collection.

`const StPtrVecClassNameIterator end() const;`
`const StSPtrVecClassNameIterator end() const;`
Returns iterator to the the last+1 element in the collection.

`void erase(StPtrVecClassNameIterator iter) const;`
`void erase(StSPtrVecClassNameIterator iter) const;`
Deletes element referred to by iterator `iter`. If applied to structural containers the object gets also deleted.

Public Member Operators

`StClassName*& operator[](unsigned int i);`
Returns the pointer to the `i`'th element where `i` runs from 0 to `size()-1`.

There are many more than one can describe here. If you want to learn more you better have a look at the `StArray.h` source code.

Needless to say that every container comes with two iterators, a constant and a non-constant version. The name of each iterator is composed of the name of the container and the suffix `Iterator` or `ConstIterator`. Example: For the structural container `StSPtrVecTrackNode` the iterators `StSPtrVecTrackNodeIterator` and `StSPtrVecTrackNodeConstIterator` are defined. Iterators care if they iterate over structural or non-structural containers so there are different iterators for `StSPtrVecTrackNode` and `StPtrVecTrackNode` containers.

We already mentioned that all containers are ordered vectors, hence the two methods to iterator/loop over a collection work both as well. It's a matter of taste which one you choose, although the iterator version has some advantages and is somewhat safer.

```
StPtrVecTrack container;
float x;

\\ method 1
for (unsigned int i=0; i<container.size(); i++)
      x = container[i]->length();

\\ method 2
for (StPtrVecTrackIterator i = container.begin(); i != container.end(); i++)
      x = (*i)->length();
```

A warning at the end. Although `StArray` provides a interface compatible with the Standard C++ Library (former STL) it is not guaranteed that the standard algorithms will work (`sort`, `accumulate`, `copy`, `find`, ...). You better check this from case to case. Don't say you haven't been warned.

For your own analysis (or reconstruction) code you might use the standard STL containers together with StEvent provided that you classes are not processed via `rootcint`. Since STL containers are transient they are more efficient if speed and use less memory if this is your concern.

2.6 Getting StEvent: The StEventMaker

StEvent is set up and filled in a "maker" with the name `StEventMaker`. This maker reads DST tables stored in memory and does all the things to make StEvent nice and useful. How the DST gets into memory is another story and is explained in the next section (2.7). In principle all you have to do is to make sure that `StEventMaker` is in the chain and called at the right place and at the right time. The only public data member and the two methods you should be aware of are:

Public Data Member

`bool doLoadTpcHits;`
Controls if TPC hits should be loaded (default=kTRUE).

`bool doLoadFtpcHits;`
Controls if FTPC hits should be loaded (default=kTRUE).

`bool doLoadSvtHits;`
Controls if SVT hits should be loaded (default=kTRUE).

`bool doLoadTptTracks;`
Controls if TPT tracks should be loaded (default=kFALSE).

`bool doPrintEventInfo;`
Print or do not print info on the current StEvent event. (default=kFALSE). This produces a lot of output. Every major class is dumped, the sizes of all collections, and the first element in every container. Don't use it for production.

`bool doPrintMemoryInfo;`
Switch on/off checks on memory usage of StEvent (default=kFALSE). In order to get a memory snapshot we use `StMemoryInfo` from the *StarClassLibrary*. A snapshot is taken before and after the setup of StEvent. The numbers in brackets refer to the difference. Not available on SUN Solaris yet.

`bool doPrintCpuInfo;`
Switch on/off CPU usage (default=kFALSE). Tells you how long it took to setup StEvent. Timing is performed using `StTimer` from the *StarClassLibrary*.

Public Member Functions

`StEvent* event();`
Returns a pointer to the current `StEvent` object.

And don't forget to check if you got a `NULL` pointer. If something went wrong this might be the case. Something else should be mentioned here: Do *not* delete the `StEvent` object you get through these method. It wl be automatically deleted by the system once you read-in a new event.

2.7 A Standard Example: doEvents.C and StAnalysisMaker

In order to get started it is always a good idea to study a simple example which shows the essential steps on how to analyse data using StEvent. The procedure starting from scratch to run the provided StEvent usage example is

```
stardev
mkdir workdir
cd workdir
root4star
```

At the root4star prompt type:

```
.x doEvents.C(1,"-","<DST File>")
```

where `<DST File>` must be replaced by an actual DST file. Ask one of your colleges where to find the latest DST files in either XDF (extension .xdf) or ROOT (extension .root) format.

This will run the `$STAR/StRoot/macros/analysis/doEvents.C` macro which runs a chain consisting of two makers:

`StEventMaker`: Read events from DST input files (XDF files or ROOT files; the file is handled appropriately based on file type) and load StEvent.

`StAnalysisMaker`: Picks up the StEvent event and analyze it (incorporates a few simple examples).

It runs the chain on either a single file or all files under a specified root directory (see doEvents.C for details). Example invocations are:

Processes 10 events from the specified XDF file.
```
.x doEvents.C(10,"-","/some_directory/some_dst_file.xdf");
```

Processes 42 events from the specified ROOT file.
```
.x doEvents.C(42,"-","/some_directory/some_dst_file.root");
```

Processes all events from all files found recursively under the specified directory.
```
.x doEvents.C(9999,"/some_directory/"," ");
```

The multiple-files feature works for XDF and ROOT files. To play with it yourself you can pick up *StAnalysisMaker* and modify it piece by piece or use it as a template for a Maker of your own that works with StEvent:

```
mkdir StRoot/StMyAnalysisMaker
cp $STAR/StRoot/StAnalysisMaker/* StRoot/StMyAnalysisMaker/
[edit and modify]
cons +StMyAnalysisMaker
cp $STAR/StRoot/macros/analysis/doEvents.C ./
[edit to use your maker]
root4star
```

At the ROOT prompt type

```
.x doEvents.C(<your arguments>);
```

By the time you gain more experience your “maker” will become more and more sophisticated but the basic idea shown in the example stays the same.

2.8 Further Documentation

In STAR all documentation specific to a packages is under cvs control and stored in the same repository as the source code of the package. You will find it usually in a directory called `doc`. In addition to that every package should contain a `README` and a `index.html` file with further information. (Note the “should”.)

StEvent makes use of various classes from the *StarClassLibrary* (SCL). Examples are `StThreeVector`, `StHelix` and `StParticleDefinition`. You should have a version of the SCL manual at hand. It also contains a description of the helix track model used in STAR and contains many examples.

Very important is also the documentation from the `$STAR/pams/global/idl` area. Here you will find a detailed description of the DST tables content. Since StEvent pretty much reflects this content (although in a different way and approach) this is the place to check if you don’t understand the meaning of certain variables or methods. In this manual we cannot go too much into detail. It’s already thick enough.

And finally, you really should have the C++ bible from B. Stroustrup within 100 feet distance from your desk. The more you get into C++ and OO the more you will appreciate this book. We already mentioned that StEvent is a bit complex and especially when you look deeper into its internal structure you will find weird things like virtual constructors, overloaded new/delete operators and much more. Then it is nice to have Bjarnes book.

3 The StEvent Model

In the following we describe the basic concepts of StEvent. This is not to describe every class and every method in detail but to explain the idea behind it and illustrate a few things in simple examples. If you need more details have a look at the reference section and if you want to know *everything* about StEvent you have to visit the source code directly.

3.1 Event Header

The event header carries the same name as the whole package: `StEvent`. Confused? Don't worry, when we talk about the package we write StEvent, when we talk about the class we write `StEvent`.
The class `StEvent` plays a special role since it is the entry point and the upper most object of the whole StEvent tree. From here you can reach every single bit and byte there is on the DST.
Obviously, this makes the `StEvent` class somewhat "fat". Figure reffig:umlEvent shows only a very small fraction of the class design around `StEvent`. The class `StEventSummary` contains lots of information

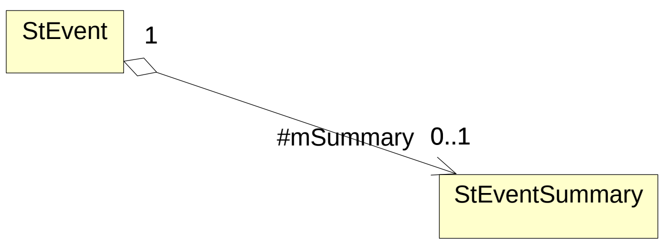

Figure 3.1: Class diagrams for `StEvent` and `StEventSummary`.

gathered during the reconstruction of the event like: the total number of tracks, the number of positive or negative tracks, the number of vertices of certain types, and several *quasi*-histograms which hold for example transverse momenta distributions and other important quantities. Check the pointer to the event summary before you use it. It could be `NULL`.

As already mentioned `StEvent` opens the door to all the info there is on the DST. In order to get there you have to navigate through the tree. Only few objects, mostly container and collections, can be reached directly from the `StEvent` objects. Here's a list of some important objects which are directly stored in `StEvent` and let you climb further down the tree:

1. Collection of software monitors
2. TPC hit collection
3. FTPC hit collection
4. SVT hit collection
5. List of all track nodes
6. List of the detector info for each track
7. Primary vertices (mostly only one)
8. List of all V0 vertex candidates
9. List of all Xi vertex candidates

10. List of all kink vertex candidates

11. Level-0 trigger

And remember, an object you get *by pointer* is not guaranteed to exist, an object you get *by reference* always exist.

What else does `StEvent` contain? Well, all the usual stuff one would expect to see in an event header: event identifier, time when the event was recorded, the trigger mask, the bunch crossing number and more. For a complete reference see section 5.18.

3.2 Software Monitors

The STAR DST contains a bunch of tables called software monitors. Before we go into details let's clarify what this is. During the reconstruction of the various detectors lots of statistics and summary information is generated which is not necessarily of importance for the physics of the event but tells you a lot on how the reconstruction programs performed. These are mostly quantities which cannot be derived from other objects in StEvent and would be lost otherwise. In a sense they *monitor* the reconstruction details. That's where the name 'software monitor' comes from.

There are many of these monitors and even the "global" reconstruction has one. This is not really a detector but a large fraction of our software deals with combining all the detectors in order to create global tracks and find the primary vertices.

Since there are many they have to be organised in a transparent way. This is depicted in Fig. 3.2 where all monitor classes and their relations are shown. You get the actual instance of `StSoftwareMonitor` from `StEvent` and then you can select which component, i.e. which monitor object you want by invoking the proper method. These methods are named after the component they return: `tpc()` returns a pointer to the `StTpcSoftwareMonitor`, `svt()` to the `StSvtSoftwareMonitor` – well, you get the idea. As usual you should check for `NULL` pointers. If a detector was not reconstructed in the reconstruction chain it's likely that you will not find the corresponding monitor.

The specific software monitor classes are pretty simple flat classes. They have no relation with any other class. All they do is to hold data. Because of this, they have no member access functions and all data members are public. In order to make things easier for people moving from table-based analysis to StEvent-based analysis we kept even the table names. With other words the software monitor classes match their table counterparts 1:1. The names are not always descriptive but the author got tired of inventing new names. You'll find more details on what is what in the reference section of this manual.

Here a simple example on how to use the software monitors:

```
void printTpcClusterInfo(ostream& os = cout, StEvent* event)
{
    StTpcSofwareMonitor *tpcMon = 0;

    if (event && event->softwareMonitor())
        tpcMon = event->softwareMonitor()->tpc();


    if (!tpcMon) return;       // no monitor

    os << "Total # of TPC cluster:" << tpcMon->n_clus_tpc_tot;
    for (int i=0; i<24; i++) {
```

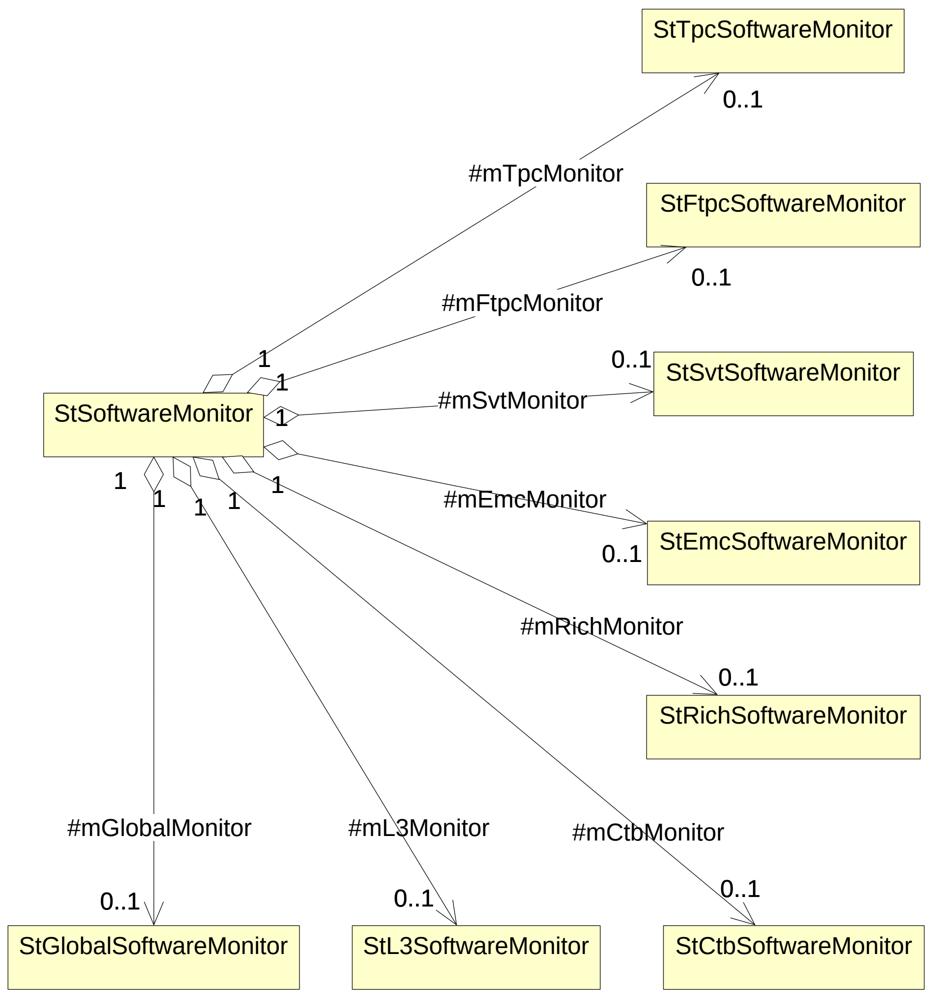

Figure 3.2: Class diagrams for the software monitors.

```
        os << "Inner sector " << i << " has "
           << tpcMon->n_clus_tpc_in[i] << " cluster" << endl;
        os << "Outer sector " << i << " has "
           << tpcMon->n_clus_tpc_out[i] << " cluster" << endl;
    }
}
```

Note that, as everywhere in StEvent indices run from `0` to `size-1`. If you are new to C/C++ and wonder why this is so, you really should read section 2.3.1.

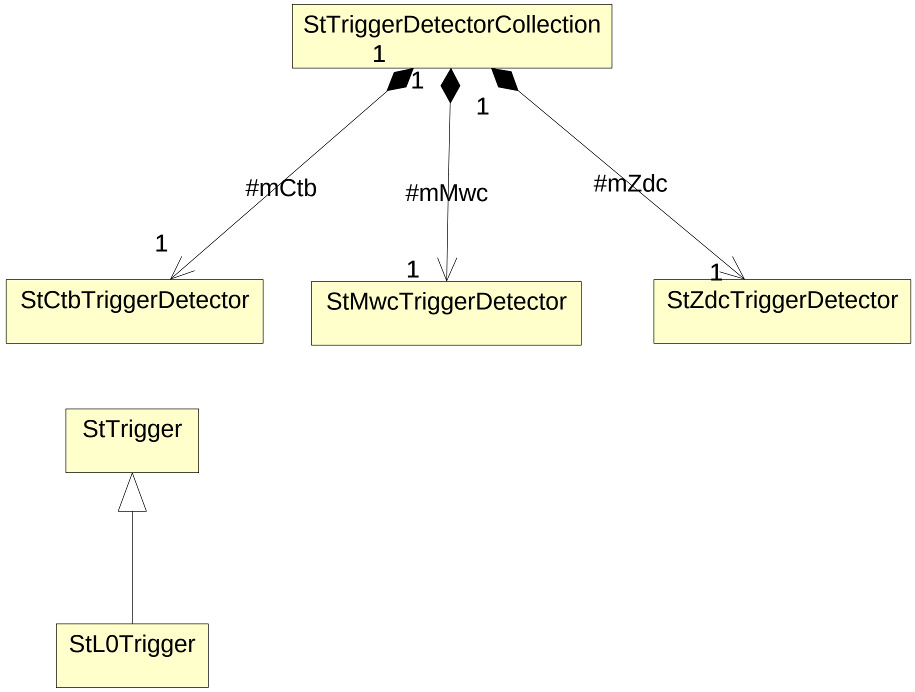

Figure 3.3: Class diagrams for the trigger detector collection and the `StTrigger` hierarchy.

3.3 Trigger and Trigger Detectors

The **trigger** is put together from data recorded by a bunch of trigger detectors combined in some logic. So far STAR deals with 4 trigger levels numbered 0 – 3. Currently only level-0 (L0) is implemented. Others will follow. All trigger classes inherit from a common base class `StTrigger`. As mentioned above, at the moment there is only one derived class `StL0Trigger` as depicted in Fig. 3.3. The class contains everything there is available about the actual trigger: trigger word, trigger action word, multiplicities, and more. The trigger is directly contained in the `StEvent` class. In order to get a pointer to the L0 trigger use: `StEvent::l0Trigger()`. Even if we repeat us here: it is a pointer and therefore can be `NULL`. At the moment all simulations have no trigger data. You were warned.

The **trigger detectors** are those detectors which data is used in the trigger (which doesn't mean that the data isn't useful for other things as well). There's a couple of them: the Central Trigger Barrel (CTB), the Zero Degree Calorimeter (ADC), the Vertex Position Detector (VPD), and the Multiwire Proportional Chamber (MWC). This means we need a collection to hold them together and indeed this is what `StTriggerDetectorCollection` is all about. The trigger detector design is shown in Fig. 3.3. The collection holds all classes which describe the different trigger detectors: `StMwcTriggerDetector`, `StCtbTriggerDetector`, `StZdcTriggerDetector`, and `StVpdTriggerDetector` (not shown). These trigger detectors store the actual ADC and TDC values including some calculated quantities. Check in the reference section for more details. The collection is a member of `StEvent`. To get a pointer to the collection use: `StEvent::triggerDetectorCollection()`. From there you get the specific trigger detectors through a set of methods. The methods are named after the component they return by reference: `ctb()` returns a reference to the `StCtbTriggerDetector`, `mwc()` to the `StMwcTriggerDetector`, and so on. Since they are returned by reference you can be sure the objects exist. No checks necessary. Note that "exist" is not a synonym for "makes sense". The reason for this is that the DST contains the data for all trigger detectors in one big table. If it available the collection (StTriggerDetectorCollection) is created else `StEvent::triggerDetectorCollection()` will return `NULL`. Once created the data in the table is used to setup the instances of the various trigger detectors. If a specific detector wasn't used its data is set 0 (so the author hopes) but the data is still there.

Here's an example which dumps the CTB data in form of a table:

```
void dumpCtb(StEvent* event)
{
    if (!(event && event->triggerDetectorCollection())) return;

    StCtbTriggerDetector &ctb = event->triggerDetectorCollection()->ctb();

    cout << " tray | slot |    mips    |      time   \n";
    cout << "----------------------------------------\n";

    for (int i=0; i<ctb.numberOfTrays(); i++)
        for (int j=0; j<ctb.numberOfSlats(); j++)
            cout << setw(5) << i <<   " | "
                 << setw(4) << j <<   " | "
                 << setw(10) << ctb.mips(i, j, 0) << " | "
                 << ctb.time(i, j, 0) << endl;


    cout << "\nL0 trigger:\n";
    if (event->l0Trigger()) {
        PR(event->l0Trigger()->mwcCtbMultiplicity());
        PR(event->l0Trigger()->mwcCtbDipole());
        PR(event->l0Trigger()->mwcCtbTopology());
        PR(event->l0Trigger()->mwcCtbMoment());
    }
    else
        cout << "not available" << endl;
}
```

Again, we are using the `PR()` macro from `StGlobals.hh` to save some typing. The names of the methods speak for themself.

3.4 Tracks

This is probably the most complex part of the design. Before we get into too much detail we give a brief introduction on what a track is and explain the differences between *global* and *primary* tracks. We then introduce the track *node* which plays a very central role in the StEvent track model. The different pieces of information which make a track such as the track geometry and the various traits are explained later together with a short introduction to filters, which, as you will learn, allow to apply predefined algorithm to select and filter information out of the data.

3.4.1 Introduction to Tracks

The STAR tracker, known as `tpt`, performs the tracking in the main STAR tracking detector the TPC. It finds a set of hits, which `tpt` assume to belong to one track and applies fits in order to determine the track parameters. Once this is done the track is passed along the chain. Points from other detectors might be added. At the end this track is then fitted with a more sophisticated fitting method and from there on is called a **global** track (class `StGlobalTrack`). The name ”global” stems from the fact that this is a fit which is possibly composed of his from several tracking detectors.

But wait, this is not the end of the story. STAR can do better than this. By using all global tracks we can reconstruct the primary vertex (or vertices) with pretty good accuracy. A track which originates from the primary vertex (and most do) can be refitted using the primary vertex as additional point. This increase dramatically the accuracy in which STAR can measure particles, both in terms of direction and momentum. If a global track points back close enough to the primary vertex and the refitting works out well (whatever that means) then this track, or better the refitted track, becomes a **primary** track (class `StPrimaryTrack`). A primary track only makes sense if it refers to a primary vertex. If a primary track is found the global track which was used to create it makes almost no sense any more and could be dropped, `if` you trust the procedure. However, things aren't as perfect and the primary track might have been misidentified. For that reason STAR keeps currently all global tracks. That means that for every primary track there is one corresponding global track but every global track does not necessarily have a corresponding primary track. The fit might have failed badly. In future this might change and we might be able to drop a fraction of the global tracks if the primary track is superior.
If a primary track fit succeeds the new track parameters and its errors are stored. To really confuse you, we should mention that even the number of hits might change, since the newly refitted track might exclude some hits and/or add new hits. StEvent is able to cope with all these scenarios and that is one of the reasons why version 2 is somewhat more complex than good old version 1.

So far so good. But what's with the tracks which fail the fit. Obviously these aren't primary tracks and – you guessed it – come from a secondary vertex. Here, things become a bit difficult. While a primary vertex can be found easily secondary vertices are more tricky to detect (at least in a Heavy-Ion collision) and can hardly be identified unambiguously. If one could do so, one could repeat the same trick as with the primary tracks and refit the global track using the secondary vertex such making it a secondary track. But we can't – at least for now. As as consequence STAR doesn't use the concept of secondary tracks yet.

All global and primary tracks are fitted according to a certain tracking model. Some models include the effect of energy loss and multiple scattering in the fit and the fit parameters therefore depends on the mass of the particle which created the track. This is not know a priori or at least cannot be determined unambiguously. In this case the same track might be fitted with different mass hypothesis. This not only alters the fit parameters and errors but possibly also the hits assigned to the track. In a sense these are tracks created from the same `seed`. How we keep track of all these different flavours is explained in the next section.

To summarize: STAR has two kinds of tracks global tracks which can come from wherever they want and primary tracks which always point back to the primary vertex. The position of the primary vertex was used to refit the primary tracks.

3.4.2 The Concept of the Track Node

As we have seen in the previous section there are two kinds of tracks (global and primary) of which each might get possibly fitted with different models or algorithms such creating a whole bunch of tracks. But we have to keep in mind that all come originally from the same `seed` formed early in the reconstruction chain. Only one of them can be the true track, or better only one comes closest to the truth. If we count tracks we can only count all of them as one. Many students spent by far too much time hunting the problem of double-counting.

We have to have a way to tell that all these “flavours” belong together, even if they have different fit parameters or even a slightly different set of hits. This is were the track **node** comes into the game (class `StTrackNode`).

A track node holds all tracks which originate from the same seed. Every track knows about the node it belongs to and thus allows to navigate from one track in the node to the other. Each node contains 1–n

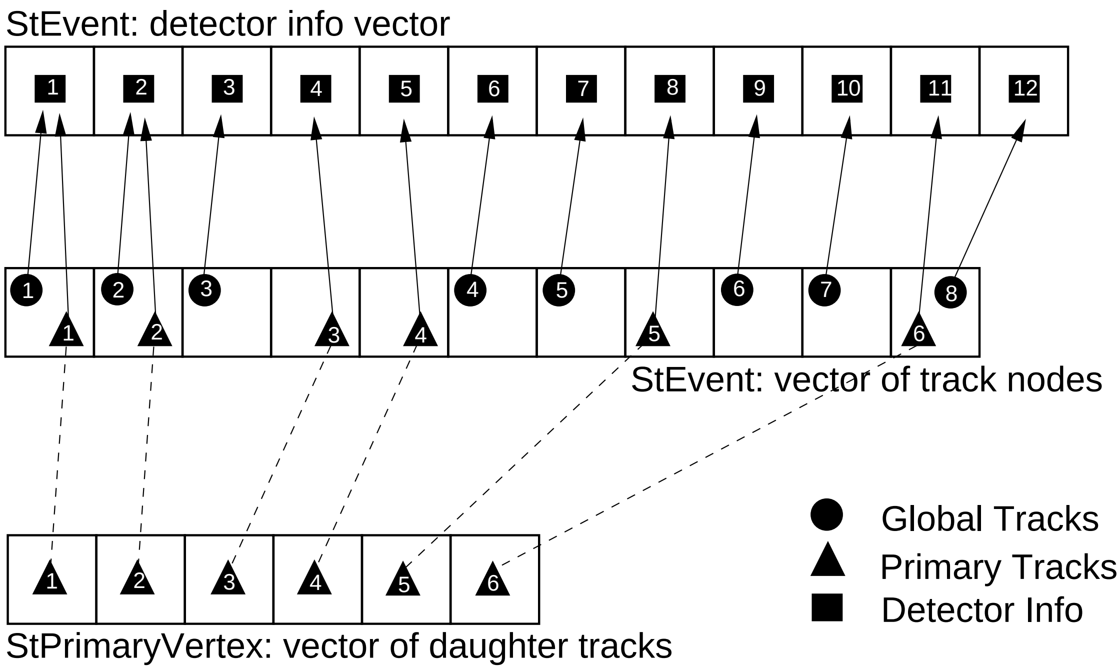

Figure 3.4: Schematic view of the track node collection and its relation to the detector info collection and the list of daughter tracks of the primary vertex.

tracks. This is depicted in Fig. 3.4. The array shown in the middle of the picture shows the collection of nodes as held by the StEvent class itself. Every element (depicted as a box) represents one node which contains a primary (solid triangle) and/or a global track (solid circle). The length of the track node list lies between: max($N_{primary}$, N_{global}) and $N_{primary}$ + N_{global}.

3.4.3 Detector Information

From the previous section you probably got the impression that a given primary track and its referring global track share lots of information. Actually, there is much less to share then one might think. Almost everything changes or can change when a track is refitted. One of the few things which often do not change are the hits used in the tracks. If the global track fit points back to the vertex the additional constraint, i.e. the position of the primary vertex, changes the parameters in fact only slightly.
If the set of hits, or the detector information, is the same then it belongs in a separate class so one can use it for all tracks in the same node. This is why there is a class `StTrackDetectorInfo`.

All detector specific information (essentially the list of hits) is contained in this class. A track can well live without them since all the reconstruction is already done. And indeed on the long term STAR cannot afford to write *all* hits to DST. In this case each track might or might not have a pointer to an existing instance of `StTrackDetectorInfo`. Since several tracks can share this instance it is obvious that no track can own them. This is why all objects of type `StTrackDetectorInfo` are stored in a separate, flat and simple list which is directly accessible from `StEvent`. Each track only points to its detector info. This is depicted in Fig. 3.4. The upper array represents a possible list of detector info objects. As you can see tracks in a node mostly share the same detector info but this doesn't need to be the case. If a primary vertex fitter decides to reject one or more hits and/or adds new hits than the detector infos might be different although the tracks are in the same node (see right most node in the figure as an example). It makes obviously no sense to keep both tracks in the same node if the hits are *very* different but if only one or two hits are different they still are related - somewhat.

Note, that the size of the detector-info list is larger or equal the number of nodes.

3.4.4 The Track Classes

So far we only discussed the basic concepts. It is time now to have a closer look at the design of specific classes. It is really helpful to look at the class diagrams in Fig. 3.5. It looks complicated but once you get the idea things become easy.

The base class `StTrack` is an abstract class, i.e. you won't be able to create an instance of it. The two concrete classes are `StGlobalTrack` and `StPrimaryTrack`. Both have the **same** interface as `StTrack`. Whatever you can do with an instance of `StGlobalTrack` you can do with `StPrimaryTrack` as well. The difference is in the implementation but not in the interface. For this very reason whenever a track is returned by a method or is used as an argument, a pointer or a references to `StTrack*` is used. This is were polymorphism comes in handy.

With other words it is sufficient to explain `StTrack` and the other two come for free. As you can see in Fig. 3.5 `StTrack` is composed of several classes. It either contains them by value or by pointer. There are:

StTrackGeometry This is an abstract class which only serves as an interface the the actual, concrete implementation. You get a pointer to the instance via the `StTrack::geometry()` method. The track geometry contains exactly what the name implies. It describes the parameters of the track which let us describe the path of the particle in the detector. Which set of parameters are actually obtained from a fit depends strongly on the track model. However, we don't want any new track model to make you change your code and this exactly is the *reason d'etre* for `StTrackGeometry`. It defines the interface and with it the parameters it has to provide. If the track model does not directly use or produce them they have to be derived. This insures that every tracking model which gets plugged in doesn't break anything. The class guarantees that you always get:

- curvature (in cm^{-1})
- charge (in units of +e)
- dip angle (in radians)
- psi (in radians) , i.e. ψ not ϕ_0 – watch out[2]
- origin (as a `StThreeVectorF`)
- momentum at the origin (as a `StThreeVectorF`)
- a helix (as a `StPhysicalHelixD`)

The helix now is somewhat special since it obviously implies that the track can be described as such. Although this is not always true (FTPC, low momentum tracks in TPC) it is a very good approximation for almost all TPC tracks – and a helix can be handled analytically. This makes it very useful to find the distance-of-closest approach to a given point, to extrapolate the path of the track and to easily get the 3-momentum at every point along the trajectory.

At the moment there is actually only one concrete class implemented and that is – you guessed it – the helix model (`StHelixModel`). This is where all the calculations (if any) are done to make sure you get what you ask for.

If you want to know which model is actually used you may call the `StTrackGeometry::model()` method which returns an element of the enumeration type `StTrackModel`. See in section 2.2 what types are available or check directly in `StEnumerations.h`.

[2] if you don't know the difference have a look in the appendix of the *StarClassLibrary* manual. There the parameters are explained in detail.

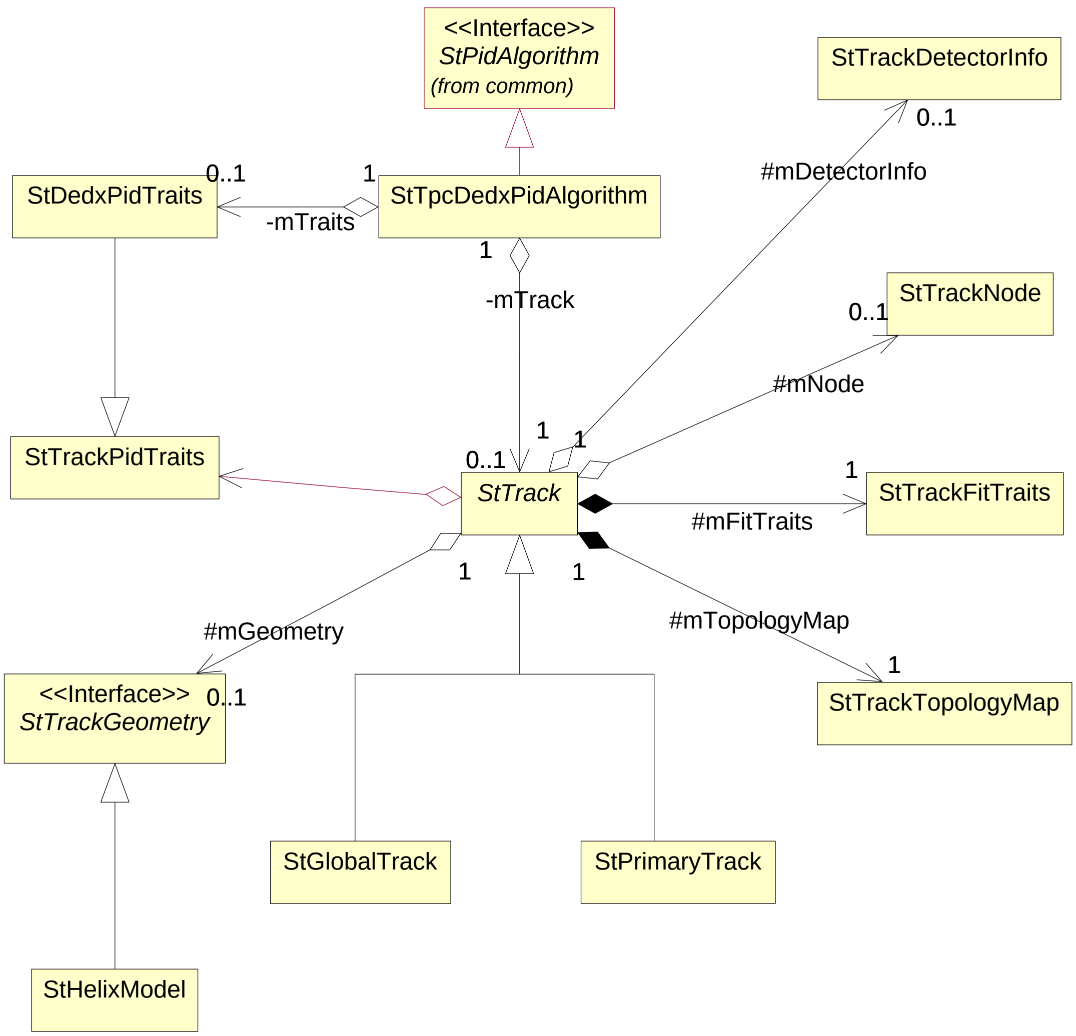

Figure 3.5: Class diagrams for `StGlobalTrack` and `StPrimaryTrack` including related classes and dependencies.

StTrackFitTraits Every track gets fitted and every fit algorithm provides errors, a covariant matrix and a χ^2 value – if the algorithm is worth a penny. This and a bit more is stored in the `StTrackFitTraits` which you get through `StTrack::fitTraits()` by reference! By reference since `StTrack` contains the instance by value. It is always present. No need to check for `NULL` pointer and such crap. There's no need for an abstract layer hence we don't need a pointer.
There might be different ways to fit and different ways to calculate the errors but they better be available, always. After all, this is what determines the quality of the track and thus decides if tracks get included in the analysis or get rejected.

StTrackNode See section 3.4.2. `StTrack::node()` will return a pointer to the node the track belongs to.

StDetectorInfo See section 3.4.3. `StTrack::detectorInfo()` will return a pointer to its respective detector info. Note, that there is no way to navigate back from the detector info to the tracks which are using it.

StPidTraits Each track has a list (container) of so called PID traits. Each of them contains information on the ID of the particle. What they actual provide is not specified. All we know is that we get an object which tells us something about the identity of the track. `StPidTraits` is an abstract class. The concrete classes are `StDedxPidTraits`, `StRichPidTraits`, and `StTofPidTraits`. The latter two are not implemented yet. This part is a bit complicated and that's why it got its own section (see 3.4.6 below).

StTrackTopologyMap The STAR detectors produces all together almost a million hits. In order to keep the DST size at a moderate level all cannot get stored, probably none on the long term. There are however many reasons to keep a minimum level of information about the hits used to fit the tracks. This minimum level is contained in `StTrackTopologyMap`. For more check out the reference manual.

3.4.5 TPT Tracks

You probably never will need them but they are mentioned here for completeness. The main reason they are in StEvent is for debugging purposes and studies by reconstruction experts. For physics always use global or primary tracks.

The current STAR tracker for the TPC is called TPT. It not only finds the tracks but also performs some simple fits. In a sense 'TPT tracks' are real tracks, but, and this is important, TPC only tracks. In addition the track parameters are determined in a very simplistic fitting method.
TPT tracks are described by `StTptTrack` which is identical in look and feel to `StGlobalTrack`. Tracks of this type are owned by the referring track node. And this is almost everything there is to say about them.

3.4.6 PID Traits

PID traits contain information about the identity of the track. Every detector will supply some sort of information useful for PID and there will be several methods for each detector to derive the same kind of information. The most basic ways to find out about the PID of a track are:

dE/dx in TPC, FTPC and SVT.

Ring area densities in the RICH detector.

TOF information from the TOF patch.

Topology info where the ID of a track can be derived, or at least be constraint, from its measured decay products (e.g. kinks).

It seems natural that, as the experiment progresses, STARs PID methods will be refined and new algorithms will get developed. If every PID method for every detector would require an concrete interface (via concrete classes) the class `StTrack` would be subject to permanent modifications. Schema evolution would become daily business. Very bad. The only way out of this dilemma is to shield `StTrack` from this kind of PID inflation by adding an abstract layer. And this is all what `StTrackPidTraits` is for.
`StTrack` now holds only a list of pointers to `StTrackPidTraits` and doesn't need to know about any specific details. Since the various ways of doing PID differ quite significantly there is hardly any data member or method they have in common. That's why the abstract class `StTrackPidTraits` has only one member which returns the ID of the detector the PID info originates from. The PID traits collection in `StTrack` obviously contains concrete objects which will provide the data you are looking for but `StTrack` is screened from any further details.
There is currently only one concrete class implemented which is meant to contain the dE/dx derived from various methods in the TPC, FTPC and SVT: `StDedxPidTraits`. If a specific PID method or detector needs more than this class provides a new one has to be created. For sure, a new class is needed for the RICH, for the TOF and for topology-PID. But that's something for the future.
The class `StDedxPidTraits` gives you the mean dE/dx, the error on the mean, the number of points used, and the method which was applied to calculate it. This method is returned as an enumeration (`StDedxMethod`) and can take the following values: `kTruncatedMeanId`, `kEnsembleTruncatedMeanId`, `kLikelihoodFitId`, `kWeightedTruncatedMeanId`, and `kOtherMethodId`. The latter is a place-holder which can be used for tests and code development (see also sec. 2.2).

So now I have a list of `StTrackPidTraits` with which I hardly can do anything – how do I get the object I need? Good questions with an easy answer. You have to scan the list and pick out the object you are looking for and **cast** it up to the concrete class for only the concrete class will reveal its content. This is where you obviously need RTTI (Real Time Type Information) as provided by ANSI/C++. Alternatively you can use ROOT-RTTI which we will not discuss here. And here an example to show how it works:

```
//
// Given a pointer 'track' to a valid track object
// we first get the list.
//
StSPtrVecTrackPidTraits& traits = track->pidTraits()

//
// What we want here is the dE/dx from the TPC from
// a simple truncated mean. This means:
// 1. detector = kTpcId
// 2. class    = StDedxPidTraits
// 3. method   = kTruncatedMeanId
//
StDedxPidTraits* pid;        // this is what we want

for (int i=0; i<traits.size(); i++) {
     if (traits[i]->detector() == kTpcId) {
         // Here we know it is some PID object derived from TPC data

         // Now the dynamic cast
         pid = dynamic_cast<StDedxPidTraits*>(traits[i]);

         // If traits[i] is NOT of type StDedxPidTraits the dynamic cast
```

```
          // returns a NULL pointer. No other cast can do this !!!
          // If we succeed we found the right object.
          if (pid && pid->method() == kTruncatedMeanId) break;
      }
}

if (pid) {
   // We found what we wanted  ....
   cout << pid->mean() << endl;
}
```

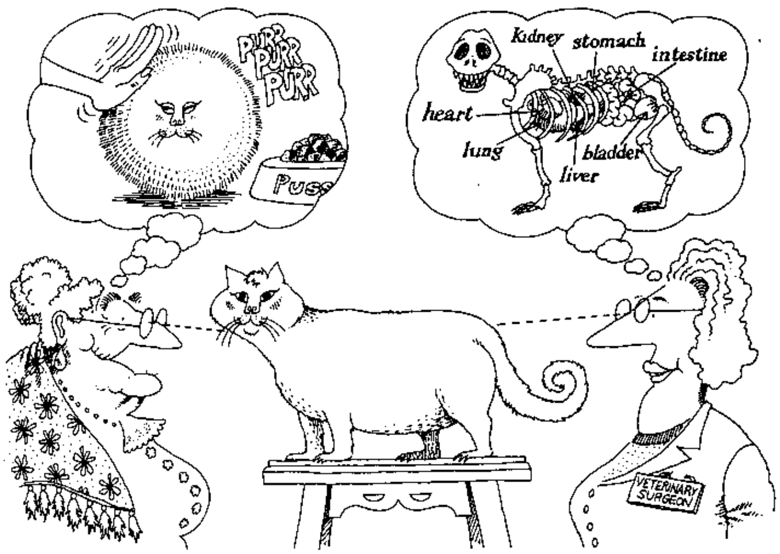

Figure 3.6: Abstraction focuses upon the essential characteristic of some object, relative to the perspective of the viewer.

Instead of a `dynamic_cast` one also could use `typeid()` as in

```
        if (typeid(*pid) == typeid(StDedxPidTraits))
           pid = static_cast<StDedxPidTraits*>(traits[i]);
```

which is probably even faster.

In the example we used `StTrack::pidTraits()` to get the whole list. In fact we can do better. Since we already know we want PID from the TPC we can use the overloaded version `StTrack::pidTraits(StDetectorId)` to get all PID traits for one specific detector. What happens internally is that the method scans the whole list, creates a new container and puts in all the objects (or better the pointer to the objects) with PID data from a the requested detector. This is what the method returns.

Now we save a line and the example above looks like:

```
StPtrVecTrackPidTraits traits = track->pidTraits(kTpcId);
StDedxPidTraits* pid;

for (int i=0; i<traits.size(); i++) {
```

```
    pid = dynamic_cast<StDedxPidTraits*>(traits[i]);
    if (pid && pid->method() == kTruncatedMeanId) break;
}

if (pid) {
   // We found what we wanted  ....
   cout << pid->mean() << endl;
}
```

But that's not the end of it. We can do even better, but this is described in the next section (3.4.7) since it needs a bit more explanation. With what was shown here you already get very far. Remember that every cast but a `dynamic_cast` will cause you nothing but trouble. Have a look at your favourite introductory C++ textbook on `dynamic_cast` and RTTI. (If your favourite introductory C++ textbook doesn't discuss `dynamic_cast`, carefully tear out all pages and recycle them. Dispose of the book's cover in an environmentally sound manner, then borrow or buy a better textbook.)

3.4.7 PID Algorithm, Filters and Functors

In OO one often talks about *functors* which are essentially nothing but functions wrapped in a class. The reason why one wants to do this are manifold. One is that one can build up a hierarchy of functions by inheritance and, this is even more important, lifetime control. A function is gone when it finishes while an object still lives happily in memory. Thus a functor can do some work and then rest until someone comes and picks up the information it has stored. Also it is *much* easier to pass objects than pointer to functions. (Ever tried to pass an array of function pointers in C ?).

If a functor is used to scan a list and returns only a subset of the elements it is called a filter. In the context of PID traits we use a PID algorithm which serves as a filter but is supposed to do a bit more than this.

The essential method all tracks provide is:

```
const StParticleDefinition*
StTrack::pidTraits(StPidAlgorithm& algo) const;
```

As usual `StPidAlgorithm` is an abstract class (functor) which does nothing but defining the interface to the "real" function, i.e. it defines the arguments it takes and what it has to return. `pidTraits()` then calls this function, passing to it the proper arguments. The important thing is that we require `pidTraits()` to return 'something', namely the definition of the most probable particle (for `StParticleDefinition` see the *StarClassLibrary*). How it does that is up to the guy who implements the concrete functor, that is you.
The decleration of `StPidAlgorithm` from `StFunctional.h` looks as follows:

```
struct StPidAlgorithm
{
    virtual StParticleDefinition*
    operator() (const StTrack&, const StSPtrVecTrackPidTraits&) = 0;
}
```

The function which does the work is invoked when the operator() is invoked. All the data needed to do the job are passed as arguments. This is the track itself and the list of all PID traits. The PID algorithm now can pick up the detector (or detectors) and methods of its choice and derive the final answer. With other words the algorithm is doing the PID. Over time you will collect a set of PID algorithms which you can plug in whenever needed. They may use different detectors and methods or possibly combine them.

To make it completely clear, here's an example of a PID algorithm which uses the dE/dx of the TPC and the SVT and returns the most probable particle:

```
// MyPID.h
#include "StEventTypes.h"
struct MyPID : public StPidAlgorithm
{
    StParticleDefinition*
    operator() (const StTrack&, const StSPtrVecTrackPidTraits&);
};

// MyPID.cxx
#include "MyPID.h"
StParticleDefinition*
MyPID::operator() (const StTrack& track,
                   const StSPtrVecTrackPidTraits& traits)
{
   StDedxPidTraits* tpcPid = 0;
   StDedxPidTraits* svtPid = 0;

   for (int i=0; i<traits.size(); i++) {
        StDedxPidTraits *pid = dynamic_cast<StDedxPidTraits*>(traits[i]);
        if (pid && pid->method() == kTruncatedMeanId) {
           if (pid->detector == kTpcId)
               tpcPid = pid;
           else if (pid->detector == kSvtId)
               svtPid = pid;
        }
   }

   if (svtPid && tpcPid) {
       // do something with the numbers and figure
       // out what particle is most likely
       // ....

       // Assume it's a pion
       if (track.geometry()->charge() > 0)
           return StPionPlus.instance();
       else
           return StPionMinus.instance();
   }
   else
      return 0;
}
```

The piece of code where you make use of the class might look as this:

```
#include "MyPID.h"
// ....

MyPID mypid;
const StParticleDefinition *part = track->pidTraits(mypid);
```

```
cout << "The name of the particle is " << part->name() << endl;
cout << "its mass is m = " << part->mass() << " GeV/c2" << endl;
```

So far so good, but what if I don't want to return something, what if I simply want to have a look without making a decision? Easy, return a `NULL` pointer – who cares. As long as you know what the algorithm is doing this should work fine.

Here's a simple version of this approach. Let's say we are interested in TPC dE/dx (truncated mean) and nothing else:

```
// MyTpcAlgo.h
#include "StEventTypes.h"
class MyTpcAlgo : public StPidAlgorithm
{
public:
    MyTpcAlgo() {mTraits = 0;}

    StParticleDefinition*
    operator() (const StTrack&, const StSPtrVecTrackPidTraits&);

    StDedxPidTraits* traits() { return mTraits; }

private:
    StDedxPidTraits *mTraits;
};

// MyTpcAlgo.cxx
#include "MyTpcAlgo.h"
StParticleDefinition*
MyTpcAlgo::operator() (const StTrack& t, const StSPtrVecTrackPidTraits& traits)
{
   mTraits = 0;
   for (int i=0; i<traits.size(); i++) {
        if (traits[i]->detector() != kTpcId) continue;
        StDedxPidTraits *pid = dynamic_cast<StDedxPidTraits*>(traits[i]);
        if (pid && pid->method() == kTruncatedMeanId) {
           mTraits = pid;
           break;
        }
   }
   return 0;
}
```

This now works really as a filter. We added three things which `StPidAlgorithm` does not require: A private data member `mTraits` which is meant to hold the "right" type of PID traits we want to filter out, a method to return it `traits()`, and a constructor to initialize the private data member to `NULL`. Note, that the base class `StPidAlgorithm` only wants us to define the `operator()`, the rest is up to us. We are free to add whatever we want.

This is how it can be used:

```
#include "MyTpcAlgo.h"
// ....
```

```
MyTpcAlgo tpcDedx;
track->pidTraits(tpcDedx);

cout << tpcDedx.traits()->mean() << endl;
cout << tpcDedx.traits()->errorOnMean() << endl;
```

This code uses very few lines. The code in `MyTpcAlgo` is highly re-usable and whoever uses the PID algorithm saves a lot of typing.

In StEvent there is actually one concrete PID algorithm implemented: `StTpcDedxPidAlgorithm`. The algorithm used stems from Craig Ogilvie. It filters out the TPC dE/dx object (StDedxPidTraits) and returns the most probable particle, but also keeps all the information selected. The additional methods now make use of the stored information and let you work with the object after the select/filter operation is done. It is much more complicated then the examples shown here but the basic idea is the same. See 5.86 for details.

3.5 Vertices

A vertex is, after all, a measured point in space and that's why the basic vertex class `StVertex` inherits from an abstract base class called `StMeasuredPoint`. The same is actually true for all hits. A measured point has a position, position errors, and even a covariant error matrix but it doesn't implement them. All it does is to guarantee that everything which inherits from it provides these methods. The advantage is that all measured points (i.e. hits and vertices) have the same basic methods which is a great advantage when it comes to fitting or drawing.

The base class for all vertices is `StVertex`. In addition to the methods inherited from `StMeasuredPoint` each vertex provides a `type()` method which returns an enumeration type StVertexId (see section 2.2), a χ^2 value from the fit, a pointer to the parent track and a list of daughter tracks. However, `StVertex` is still abstract. The five concrete vertex classes are:

`StPrimaryVertex` to hold the events vertex (or vertices)

`StCalibrationVertex` to hold the various vertices used for calibration and test purposes.

`StV0Vertex` which is primarily used for K_0 and Λ decay vertices

`StXiVertex` for Ξ decay vertices

`StKinkVertex` for kink vertices

The UML diagram for the vertices classes is shown in Fig. 3.7.

The primary vertex `StPrimaryVertex` class plays an important role since it holds all primary tracks. This is depicted in Fig. 3.4. If the class gets deleted all primary tracks get deleted.
The primary vertex, or vertices, are directly stored in the `StEvent` class. Usually, in Au-Au collisions there's only one "primary" vertex but there are cases (pile-up events) where there can be more than one. That's why `StEvent` has the `numberOfPrimaryVertices()` method and the access member function has an optional argument `primaryVertex(unsigned int i=0)`. Hence `event->primaryVertex()` implies `event->primaryVertex(0)`. If there is more than one you have to give the index.
The primary vertices are ordered according to the number of daughters (i.e. primary tracks) they hold. The first in the list is always the the vertex with the most daughter tracks.

All secondary vertices `StV0Vertex`, `StXiVertex`, and `StKinkVertex`, store only references to their daughter tracks but they do not own them. In the reconstruction phase the daughter tracks are actually

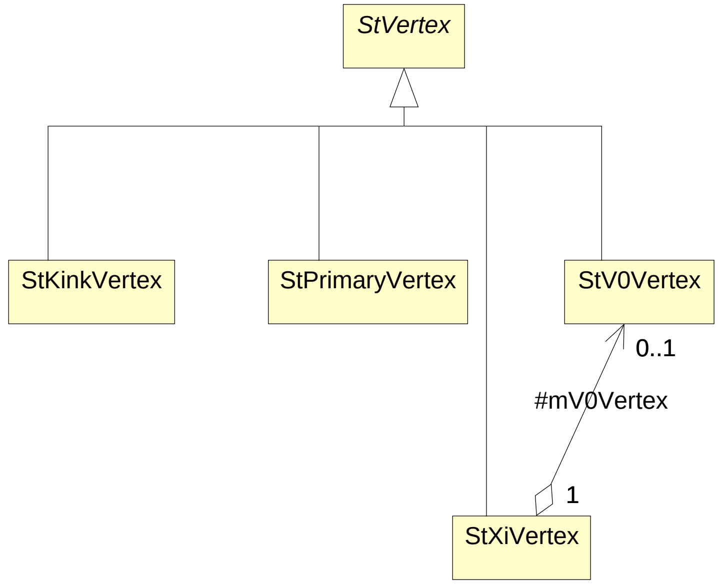

Figure 3.7: Vertex class diagrams and their dependencies. `StCalibrationVertex` is not shown.

refitted with the vertex constraint but do not become StTrack objects. Only the momentum is extracted and stored as data member in the referring vertex class. This might change later but this is how it is done now. The referenced daughter tracks are always global tracks (`StGlobalTrack`).

By the way, although all vertex classes hold a parent pointer it is (currently) always zero. This might change in future.
All secondary vertices are stored in flat containers and accessible from `StEvent`. The methods to obtain a reference (!) to the collections are `StEvent::v0Vertices()`, `StEvent::kinkVertices()`, and `StEvent::xiVertices()`.

3.6 Hits

The base class for all concrete hit classes is `StHit` which itself inherits from `StMeasuredPoint`. So all the 'position' related stuff comes with the measured point: position, position errors, and a covariant error matrix. `StHit` adds what all hits have in common such as a charge, a detector ID, a track reference count, i.e. the number of tracks which use the hit, and a method to return the list of tracks which reference the hit. The list of tracks is created on the fly since a hit doesn't know anything about tracks. Only tracks hold references to their hits.

Here an example on how to obtain a list of all global and primary tracks which use a given hit:

```
void f(StHit& hit, StEvent& event)
{
   StPtrVecTrack gvec, pvec;
   gvec = hit.relatedTracks(event->trackNodes(), global);
   pvec = hit.relatedTracks(event->trackNodes(), primary);

   if (gvec.size() + pvec.size() != hit.trackReferenceCount())
      cerr << "This cannot happen unless something is very wrong." << endl;
```

```
   cout << "The hit is used to fit "
        << gvec.size() << " global tracks and "
        << pvec.size() << "primary tracks."

   cout << "The momenta of the tracks are:" << endl;
   int i;
   for (i=0; i<gvec.size(); i++)
       cout << gvec[i]->geometry()->momentum() << endl;
   for (i=0; i<pvec.size(); i++)
       cout << pvec[i]->geometry()->momentum() << endl;
}
```

Needless to say that `StHit` is an abstract class. The concrete classes are: `StTpcHit`, `StSvtHit`, and `StFtpcHit`. The class diagram in Fig. 3.9 shows their relation. In the following we describe the different classes in detail.

3.6.1 TPC hits

Each hit returns its position in global coordinates. In addition hit classes also provide information on their *local* coordinates through methods which decode a detector specific data word. For the TPC hits there are methods to return the sector number (1-24) and row number (1-45). Please read section 2.3.1 about the difficulties with numbering schemes starting at 1. There are two additional member functions `padsInHit()` and `pixelsInHit()` which return information on the number of pads and pixels used to compose the hit. See 5.87 for details.

The hits are stored in a tree-like structure organized according to their "natural" location, i.e. sector- and row-wise. The collection you obtain (by pointer) via `StEvent::tpcHitCollection()` holds a list of sectors (`StTpcSectorHitCollection`) which itself holds a list of padrows (`StTpcPadrowHitCollection`). Each padrow finally contains the list of hits in this padrow. This is illustrated in the class diagrams in Fig. 3.10. You get the idea when you look at the following example:

```
const int isec = 3;    // 4th sector
const int irow = 24;   // 25th row

cout << "sector " << isec << " contains "
     << event->tpcHitCollection()->sector(isec)->numberOfHits()
     << " hits" << endl;

StSPtrVecTpcHit& theHits =
event->tpcHitCollection()->sector(isec)->padrow(irow)->hits();

cout << "sector " << isec << " padrow " << irow << " contains "
     << theHits.size() << " hits" << endl;

for (int i=0; i<theHits.size(); i++) {
    cout << theHits[i]->position() << endl;
    cout << theHits[i]->charge() << endl;
    cout << theHits[i]->padsInHit() << endl << endl;
    cout << theHits[i]->pixelsInHit() << endl << endl;
    assert(theHits[i]->padrow() == irow);
    assert(theHits[i]->sector() == isec);
```

```
}
```

The top collection (`StTpcHitCollection`) and each sector (`StTpcSectorHitCollection`) provide a method `numberOfHits()` which returns just that, the number of all TPC hits and the number of hits in the corresponding sector.

This organization scheme makes it much easier to perform gain and residual studies and can be better integrated into the reconstruction phase than a long flat list of hits. The disadvantage is that looping over *all* hits does require somewhat more code but selecting hits in certain rows or sectors is easy and very efficient.

3.6.2 FTPC hits

The FTPC hit class `StFtpcHit` is very similar to the TPC version. However, because of the different detector geometry the hits are stored according to planes (20) and then sectors (6). This is depicted in Fig. 3.11. Other than the TPC hit the FTPC hit has a method to return the size of the hit in time direction (`timebinsInHit()`) but no method to return the number of pixels per hit. See 5.24 for details. You probably should also read the warnings in section 2.3.1 on the numbering schemes.

The following is the equivalent example to the one above but for the FTPC:

```
const int iplane = 11;
const int isec   = 3;

cout << "plane " << iplane << " contains "
     << event->ftpcHitCollection()->plane(iplane)->numberOfHits()
     << " hits" << endl;

StSPtrVecFtpcHit& theHits =
event->ftpcHitCollection()->plane(iplane)->sector(isec)->hits();

cout << "plane " << iplane << " sector " << isec << " contains "
     << theHits.size() << " hits" << endl;

for (int i=0; i<theHits.size(); i++) {
    cout << theHits[i]->position() << endl;
    cout << theHits[i]->charge() << endl;
    cout << theHits[i]->padsInHit() << endl << endl;
    cout << theHits[i]->timebinsInHit() << endl << endl;
    assert(theHits[i]->plane() == iplane);
    assert(theHits[i]->sector() == isec);
}
```

3.6.3 SVT hits

The SVT consist of 3 barrels (6 layers) with up to 16 (8) ladders each. Each ladder has up to 7 wafers. Consequently the `StSvtHit` class provides methods to return the local coordinates exactly in these "units": `barrel()` returns 1–3, `layer()` returns 1–6, `ladder()` returns 1–16, and `wafer()` returns 1–7. If you are puzzled why these numbers start at 1 have a look at section 2.3.1. Because of the more detailed local coordinates there is not enough space to store any further details on the hits as it is the case for the TPC and FTPC. The SVT hits are stored in a tree organized as shown in Fig. 3.12. The top collection (`StSvtHitCollection`) contains 3 barrel collections. Each barrel collection (`StSvtBarrelHitCollection`) contains up to 16 ladder collections and each (`StSvtLadderHitCollection`) contains up to 7 wafer collection (`StSvtWaferHitCollection`). The latter finally contains the hits. With other words, the hits are stored per wafer. Each of the shown classes provide a method (`numberOfHits()`) to return the number of hits stored in the referring subcomponent.

The following is the equivalent example to the two previous ones but for the SVT hits. The scheme is always the same:

```
const int ibarrel  = 1;  // 2nd barrel
const int iladder  = 10; // 11th ladder
const int iwafer   = 3;  // 4th layer

cout << "barrel " << ibarrel << " contains "
     << event->svtHitCollection()->barrel(ibarrel)->numberOfHits()
     << " hits" << endl;

StSPtrVecSvtHit& theHits =
event->svtHitCollection()->barrel(ibarrel)->ladder(iladder)->wafer(iwafer).hits();

cout << "barrel " << ibarrel << ", ladder " << iladder
     << ", wafer " << iwafer << " contains "
     << theHits.size() << " hits" << endl;

for (int i=0; i<theHits.size(); i++) {
    cout << theHits[i]->position() << endl;
    cout << theHits[i]->charge() << endl;
    assert(theHits[i]->barrel() == ibarrel);
    assert(theHits[i]->wafer() == iwafer);
}
```

3.6.4 SSD hits

The SSD consist of one layer of 20 ladders with 16 wafers each. The `StSsdHit` class therefore provides the two methods `ladder()` and `wafer()` which return the appropriate 'hardware' coordinate. Note that the numbering starts at 1 (see Sec. 2.3.1). In addition the class has member functions to return the strip number and the cluster size of the p and n side, respectively. The SSD hits are stored in a tree organized similar to the SVT (Fig. 3.12) but without the barrel collection (there's only one). The hits are stored per wafer.

The following is the equivalent example to the previous ones but for the SSD hits. Again the same scheme:

```
const int iladder  = 2;
const int iwafer   = 3;

cout << "The SSD has  "
     << event->ssdHitCollection()->numberOfHits()
     << " hits" << endl;

StSPtrVecSsdHit& theHits =
event->ssdHitCollection()->ladder(iladder)->wafer(iwafer).hits();

cout << "ladder " << iladder
```

```
     << ", wafer " << iwafer << " contains "
     << theHits.size() << " hits" << endl;

for (int i=0; i<theHits.size(); i++) {
    cout << theHits[i]->position() << endl;
    cout << theHits[i]->charge() << endl;
    assert(theHits[i]->ladder() == iladder);
    assert(theHits[i]->wafer() == iwafer);
}
```

3.7 Remarks on Hits and Vertices

The fact that hits and vertices inherit from the same base class `StMeasuredPoint` can be used wherever positions and errors are what count. Take for an example a track fit. What you have to pass to the fitting algorithm are the points and the errors. A fitter usually gives a damn if the position was taken in the SVT or TPC. What counts are the coordinates and their weight which depends on the errors. To illustrate this lets assume we have a helix fitting algorithm implemented in a fitter class `MyOwnSimpleFitter`. It provides a method to add points and one to actually fit a helix to the points.

```
MyOwnSimpleFitter::addPoints(vector<StMeasuredPoint*>);
MyOwnSimpleFitter::apply();
```

Now we can put together a simple function which takes a track as argument, extract the points and the vertex, and fills them in a vector. It doesn't matter if the track is a global or a primary track or where ever the hits may come from. This could look as follows:

```
void fillPoints(vector<StMeasuredPoint*> vec, StTrack *track)
{
    if (!track) return;

    //
    //   First add hits
    //
    StTrackDetectorInfo* info = track->detectorInfo();
    if (info)
       for (int i=0; i<info->hits().size(); i++)
            vec.push_back(info->hits()[i])

    //
    //   Add vertex
    //
    if (track->vertex()) vec.push_back(track->vertex());
}
```

And things become really easy:

```
vector<StMeasuredPoint*> points;
StTrack                  *track;
MyOwnSimpleFitter        fitter;

// ... get track from somewhere ...
```

```
fillPoints(points, track);

// some print-out
for (int i=0; i<points.size(); i++)
    cout << points[i]->position() << '\t'
         << points[i]->positionError() << endl;

fitter.addPoints(points);
fitter.apply();

// ... now print the results ...
```

The same scheme can be applied when you want to draw points along a helix or sketch a helix by drawing lines between its points.
You can mix vertices and hits freely as long as you stick to the functionality `StMeasuredPoint` provides.

One can now sort the hits according to their radius ρ in cylindrical coordinates. This is easy using the STL sort algorithm. All what is needed is the definition of the sort 'rule':

```
struct compareRadiusOfPoints {
    bool operator()(const StMeasuredPoint *x, const StMeasuredPoint *y) {
        return x->position().perp() < y->position().perp();
    }
};
```

The following simple line does the job:

```
    sort(points.begin(), points.end(), compareRadiusOfPoints());
```

3.8 The EMC

... missing ...

3.9 The PHMD

The Photon Multiplicity Detector (PMD) measures photon multiplcity event by event. It covers an η region of -2.2 to -3.5. It is equipped with Charge Particle Veto (CPV) detector having same granularity and placed before the particle encounters the converter. The information stored from this detector is StEvent are hits (StPhmdHit) and clusters (StPhmdCluster) kept in proper containers (StPhmdCollection, StPhmdDetector, StPhmdModule).

Most useful objects for analysis purpose is StPhmdCluster which contains detail information about the cluster and puts a tag on the cluster about its identity as obtained from particular discrimination method.

To obtain Phmd info from StEvent, one needs to navigate through following scheme.

(i) Obtain PhmdCollection from StEvent pointer.

(ii) PhmdCollection contains two detector pointers (PMD and CPV)

(ii) Each detector has 12 modules and it has the clustercollection attached to each detector.

(iii) Each module has StPhmdHits attached to them, so that the module acts as hit container.

(iv) ClusterCollection is the container for StPhmdCluster.

3.10 The RICH

The RICH classes, like those of the EMC do not depend on tables for their construction. They are directly filled into the `StEvent` structure during reconstruction and analysis. The `StRichCollection` is the main container class which holds all of the objects associated with the RICH—that is the pixels, clusters, and hits, and these are held in the ROOT based “STL-like” containers within `StEvent`. This allows internal changes to be made within the `StRichCollection` without imposing overhead on the `StEvent` infrastructure, or changes to other users code, so long as the persistent objects fields do not change.

The RICH detector is in the curious situation that not all the Monte Carlo information is available at the GEANT stage. Specifically a major source of background and spurious signals come from feed back photons which are generated in the avalanche process of the signal generation, i.e. in the detector simulation stage. For this reason it is necessary to keep track of the origin of the signals at the pixel level, and to include the fractional contribution to all pixels from this process at the `StEvent` level. Because of the relatively small data volume that the RICH produces—the order of 1500 10 bit ADC values per central event, this is not an overly large burden.

In the reconstruction stage of the analysis, the pixels are grouped into `StRichCluster`s based on topology characteristics, and from these objects, hit positions are reconstructed. The `StRichHit` follows the generic hit structure defined in `StEvent` for the various detectors. The `StRichHit` is an object which inherits from both `StMeasuredPoint` and `StHit`. The position of the hit is available in STAR global coordinates (as is required by the conventions of `StEvent`) as well as the local coordinate system of the RICH, so that analysis using various survey geometries can occur after the `StEvent` structure exists. In the case of Monte Carlo generated data, which includes embedded events, the process which spawned or contributed to the hit is also recoverable. In the cases of merged hits that produce large clusters, the origin of each individual pixel can be recovered by tracing back from the hit, to the cluster, to the pixel level. All of these objects are kept in ROOT generated collections within the `StRichCollection`:

- `StRichHitCollection`
- `StRichClusterCollection`
- `StRichPixelCollection`

For generic studies of efficiecy it is expected that `StMCEvent` will be utilized. However as mentioned previously, the addition of pixel labelling in the internal `StRichCollection` infrastruture will allow us to quantify, very accurately, the efficiency of utilizing a proximity match for the basis of these calcuations, as well as study the effect of various modes and generations of feedback photons, and detector noise. This labelling can also be extended to add a parameterized neutron background downstream of the GEANT calculation should this prove to be a large source of additional background.

The RICH particle identification is initially expected to run at the `StEvent` level. The identification algorithm is currently based on calculating the areal photon density on the RICH pad plane for the different particle hypthesis and selecting the most probable based on a selected set of criteria which will not be detailed here. This algorithm is contained in a Maker module called the `StRICHPIDMaker` which requires `StEvent` global tracks in addition to the `StRichCollection` as input.

The mode of accessing this data is shown below:

```
//
// retrieve the StEvent data structure
//
StEvent* mEvent;
mEvent = (StEvent *) GetInputDS("StRichEvent");

if (!mEvent) {
  cout << "No StEvent*\n";
  cout << "Can not continue. Aborting..." << endl;
  return kStWarn;
}

//
// Get the StRichCollection
//
StRichCollection* theRichCollection = mEvent->richCollection();
if (!theRichCollection) {
  cout << "StEvent::RichCollection does not exist\n";
  cout << "Aborting...\n" << endl;
  return kStWarn;
}

//
// Get the Hits from the collection
//
if (!theRichCollection->hitsPresent()) {
  cout << "StRichCollection::hitCollection does not exist\n";
  cout << "Aborting...\n" << endl;
  return kStWarn;
}

StSPtrVecRichHit& theRichHits = theRichCollection->getRichHits();

//
// Get the Global Tracks
//
StSPtrVecTrackNode& theTrackNodes = mEvent->trackNodes();
```

3.11 The L3 Trigger

The L3Trigger class looks in essence like the “little brother” of the StEvent class. It provides a subset of the data member and methods from StEvent with respect to TPC hits, track nodes and vertices. This allows, to some extend, to use exactly the same analysis code for the analysis of tracks and hits from the L3 reconstruction as for the standard offline reconstruction chain. The only difference is that the user has to replace

```
StEvent* event;
// ...
event->someMethod();
```

with

```
StEvent* event;
// ...
event->l3Trigger()->someMethod();
```

Only the entry point `event` has to be replaced by `event->l3trigger()`. However, the level of detail in the offline reconstructed data will always be somewhat larger and some information might not be available in the L3 tree (e.g. certain PID traits, fit traits etc.).

3.11.1 Event Summary Information

In addition to the track/hit information the L3 tree also contians the complete trigger information of the algorithms switched on for the given run and all neccessary counters for the cross section calculataion. Global event information is stored in the `StL3EventSummary` class, whereas detailed outcome of the algorithms is put into `StL3AlgorithmInfo`.

Since the L3 triggered events are in general not unbiased, e.g. the high-pt-RICH triggered events for a flow analysis, the average user will want to sort out these event classes. Therefore a simple member function is provided which checks how the event decision was made: unbiased, i.e. not triggered by a L3 algorithm, or biased, i.e. triggered by L3. One exception is the L3 vertex algorithm, where a bias is not seen, or exspected, for central events. This can be checked with a similar function.

The way to access this is shown below:

```
StEvent* myEvent;
myEvent = (StEvent* )chain->GetInputDS("StEvent");

if (!myEvent) {
  cout <<"No StEvent found.\n";
  return kStWarn;
}

//
// Get L3 entry point
//
myL3Trigger = (StL3Trigger* )myEvent->l3Trigger();
if (!myL3Trigger) {
  cout << "No l3 found inside StEvent.\n";
  cout << "That means l3 was switched off. \n";
```

```
    // return or continue, what ever you want
    return kStWarn;
  }

  //
  // Get L3 event summary
  //
  const StL3EventSummary* myL3EventSummary = myL3Trigger->l3EventSummary();
  if (!myL3EventSummary) {
    cout << "No l3 event summary found." << endl;
    return kStWarn;
  }

  //
  // now check the event
  //
  if (!myL3EventSummary->unbiasedTrigger()) {
    cout << "This event was triggered by L3. \n";
    cout << "Accept vertex trigger only \n";
    cout << "and skip the rest. \n";
    if (!myL3EventSummary->zVertexTrigger())
      continue;
  }
```

3.11.2 Algorithm Information

The L3 algorithm information is stored in a pointer vector of `StL3AlgorithmInfo` type. Access point is again the `StL3EventSummary` class. Ideally every information for each running algorithm is provided which is neccessary to define the algorithm and the mode it was running in, e.g. pre/postscaling factors and run parameters. Please note that the unique algorithm id might not be enough to define it, since it's possible to run an algorithm twice at the same time with different parameter sets.
An additional pointer vector is kept to get the information of all algorithms which triggered the given event to save the effort of looping over all algorithms. The following gives an example how to access the algorithm information:

```
  //
  // Get number of algorithms switched on
  //
  unsigned int nAlgorithms = myL3EventSummary->numberOfAlgorithms();

  //
  // Print info for all algorithms
  //
  StSPtrVecL3AlgorithmInfo& myL3AlgInfo = myL3EventSummary->algorithms();
  for (int i=0; i<nAlgorithms; i++) {
    cout << " alg id " << myL3AlgInfo[i]->id()
         << ":\t #proc " << myL3AlgInfo[i]->numberOfProcessedEvents()
         << "\t #accept " << myL3AlgInfo[i]->numberOfAcceptedEvents()
         << "\t #build " << myL3AlgInfo[i]->numberOfBuildEvents()
         << endl;
  }
```

```
//
// Print id of triggered algorithms only
//
StPtrVecL3AlgorithmInfo& myL3TriggerAlgInfo;
myL3TriggerAlgInfo = myL3EventSummary->algorithmsAcceptingEvent();
for (int i=0; i<myL3TriggerAlgInfo->size(); i++) {
  cout << myL3TriggerAlgInfo[i]->id() << "  " << endl;
}
```

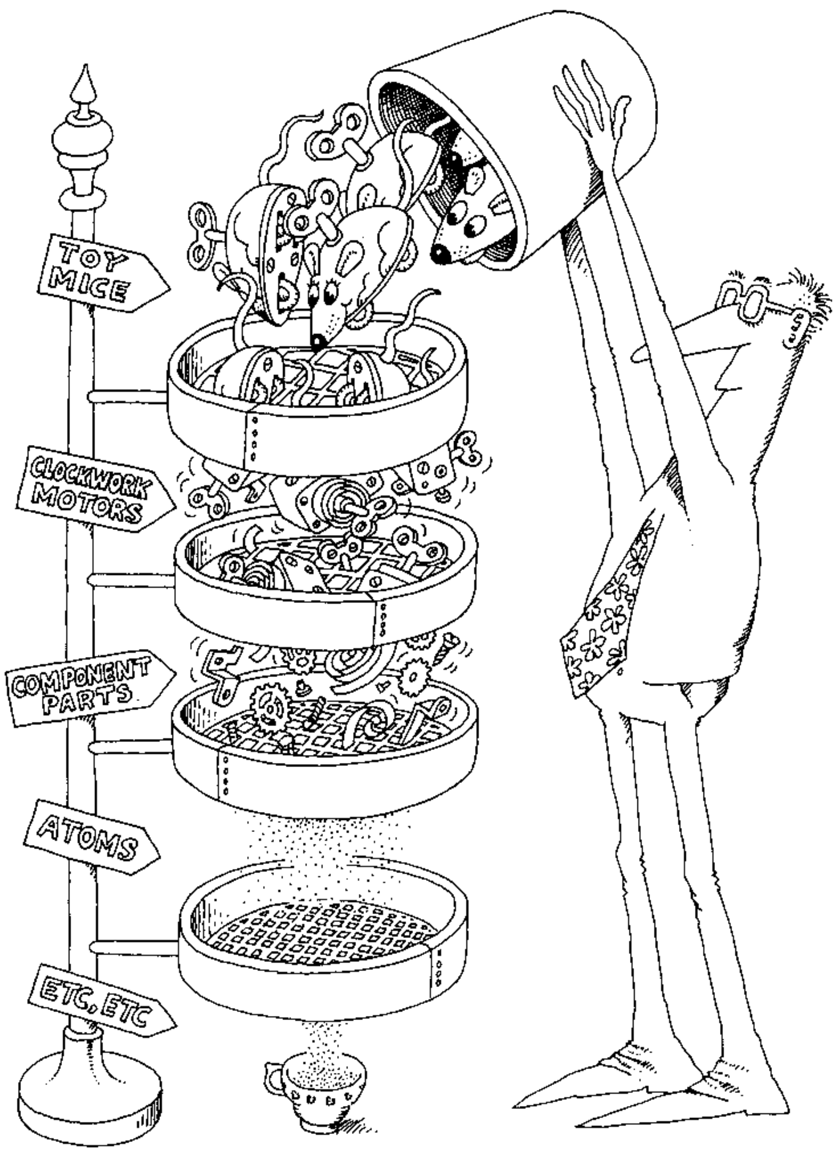

Figure 3.8: Abstraction from a hierarchy

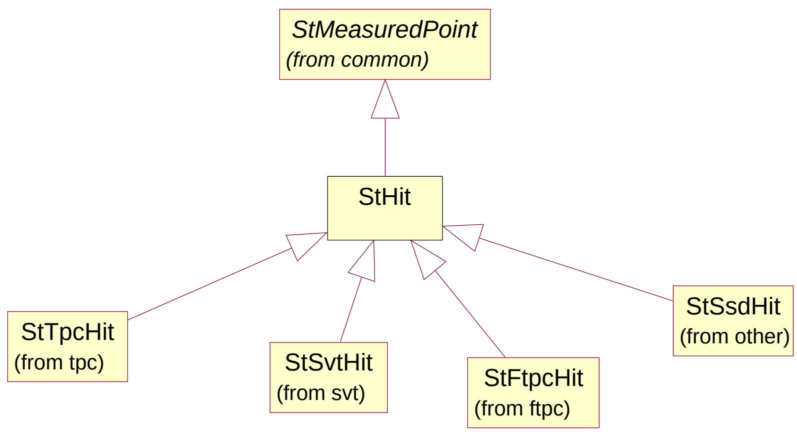

Figure 3.9: The `StHit` class and its subclasses and superclasses.

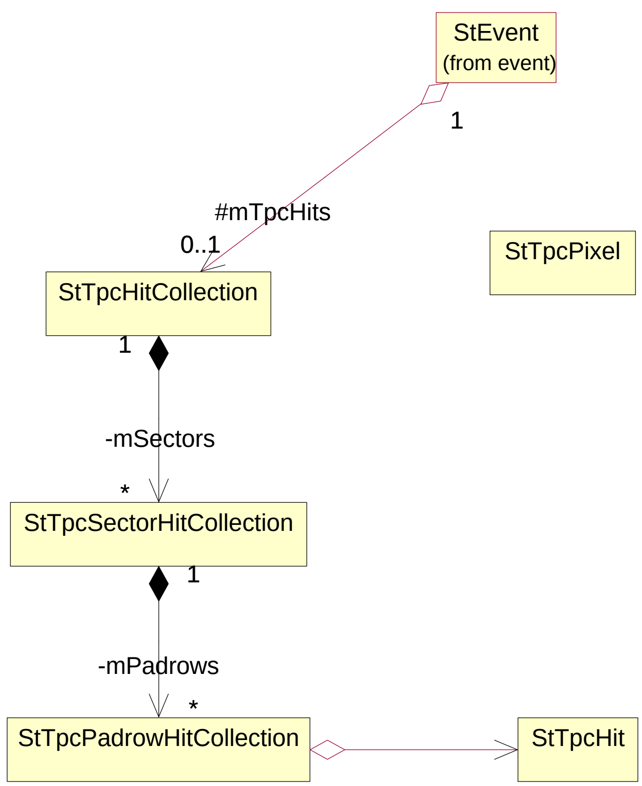

Figure 3.10: Class diagrams of the TPC hit storage scheme: sector/padrow.

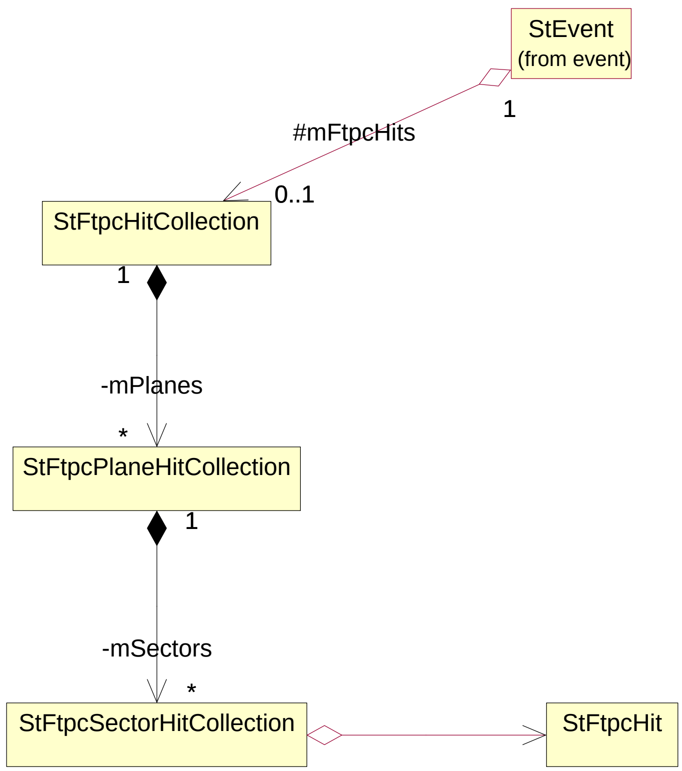

Figure 3.11: Class diagrams of the FTPC hit storage scheme: plane/sector.

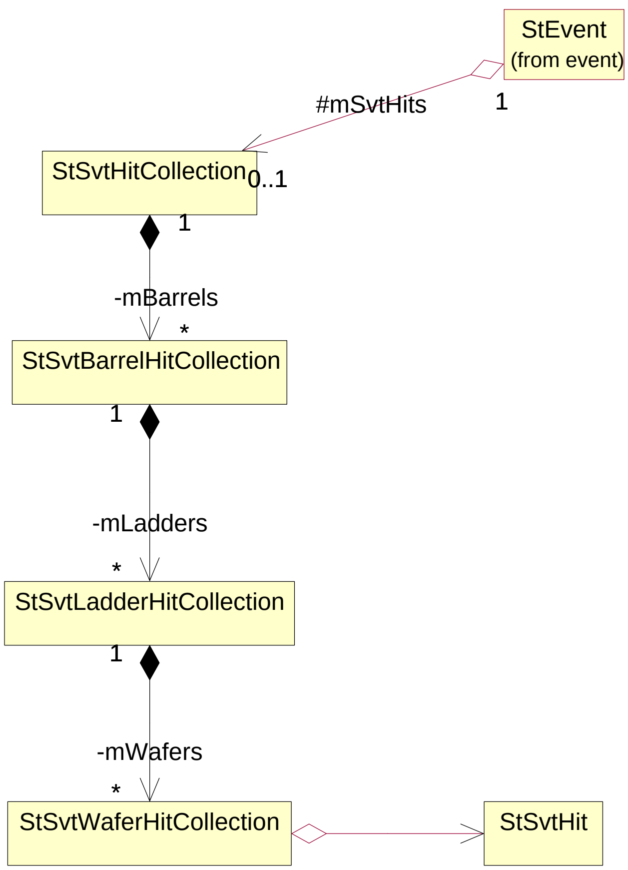

Figure 3.12: Class diagrams of the SVT hit storage scheme: barrel/ladder/wafer.

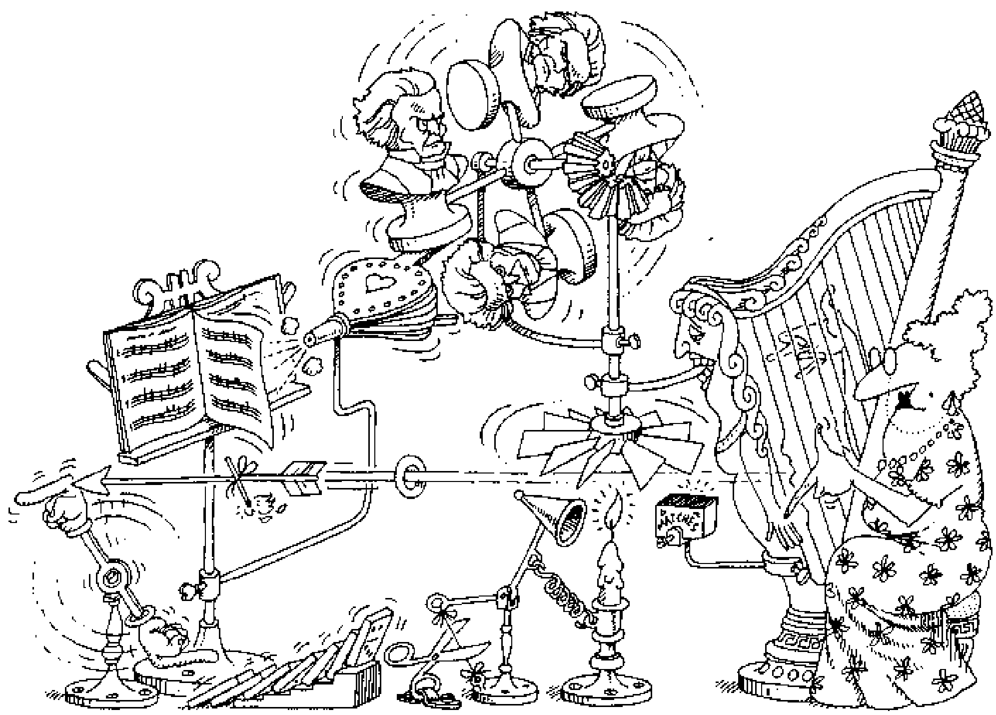

Figure 3.13: Mechanisms are the means whereby objects collaborate to provide some higher level behaviour.

4 Writing MiniDSTs using StEvent

Following STARs terminology a miniDST is the next higher level in the DST hierarchy after the DST itself. What follows are the microDST and nanoDST. The miniDST should be readable STAR-wide which implies that it has to be based on StEvent. This has several advantages:

- programs and code that works based on StEvent can be used directly; in fact the analysis code doesn't have to change at all between DST and miniDST
- schema evolution comes for free
- the standard IO maker handles the writing and reading of the miniDST

The miniDST contains a fraction of the StEvent data tree plus (optionally) user defined components. Which fraction of StEvent gets written is up to the user. The obvious candidates to go on miniDSTs are primary tracks, vertices, or certain global tracks. There is also no rule which says that there can only be one miniDST. It makes perfectly sense to keep different variants of miniDSTs. Some Physics Working Groups might want highly compressed miniDSTs for rare probe searches since they deal with many events. Other groups might want more complete miniDSTs since they need less events in total. Several PWG can go for a common miniDST, STAR can go for a common miniDST. This is a matter a choice. Important is that the scheme does not limit us in our choice that there's a way of achieving this.

The basic principle used in the implementation is not to copy and store information in a new scheme in a different place (e.g. in a TTree or user defined format) but to **remove** those pieces of the current StEvent one doesn't need. The result is the same but removing or marking instread of copying is faster and more efficient.

The extension for miniDSTs is `.mdst.root`.

4.1 The StEventScavanger class

To get remove unwanted objects from StEvent one can do two things: deleting them (operator `delete`) or marking them (using the zombie mechanism). Both methods require some level of expertise and there are many things that can go wrong. What is needed is a way to savely remove arbitrary parts of StEvent in a simple way. This is exactly what the `StEventScavanger` class is for.

The class has only static member functions which means you can call them from everywhere in your code without creating an instance of `StEventScavanger`.
Example:

```
StEvent* event = (StEvent *) GetInputDS("StEvent");
StEventScavenger::removeEventSummary(event);
StEventScavenger::removeSoftwareMonitor(event);
StEventScavenger::removeTpcHitCollection(event);
StEventScavenger::removeFtpcHitCollection(event);
```

The class has methods to delete individual tracks, all hits, event summary, software monitors, and much more (see section 5.20). Note, that most of these methods actually mark the objects and do not remove via operator `delete`. Marked objects will be ignored by the IO maker and not written to the miniDST. So don't worry if you still see them after you called StEventScavenger. They won't show up on the miniDST.

4.2 An Example: StMiniDstMaker

The STAR library contains a complete example on how to write a miniDST. The example consist of the `StMiniDstMaker` maker and two macros: `mDstWrite.C` and `mDstRead.C`.

The basic idea behind the example is to show how to write a miniDST of primary kaons which contains nothing but the kaons and the trigger information.

The most important thing is that you use the macro `mDstWrite.C` instead of the usual `doEvents.C`-like macros. `mDstWrite.C` is especially setup to write StEvent read in from a DST into a single output file. The version of the macro in the library calls the `StMiniDstMaker` package to select what should be written and what not.

So a typical run looks like this:

```
root4star -b
.x mDstWrite.C(1000,"/star/data03/reco/P00hi/2000/09","*.dst.root","mykaons");
```

In this case 1000 events will be read from various `*.dst.root` files in the `/star/data03/reco/P00hi/2000/09` directory and written to a miniDST with name `mykaons.mdst.root`. For every event the `StMiniDstMaker::Make()` method selects the kaon tracks and removes the rest. To remove primary track which are not identified as kaons the `remove(StTrack*)` from `StEventScavanger` class is used. The code looks as follows:

```
    StSPtrVecPrimaryTrack& theTracks = event->primaryVertex()->daughters();
    for (unsigned int i=0; i<theTracks.size(); i++)
        if (!accept(theTracks[i]))
            StEventScavenger::remove(theTracks[i]);
```

where `accept(StTrack*)` is a filter method defined in `StMiniDstMaker`.

To read the file back use for example:

```
root4star -b
.x mDstRead.C(1000,"/star/data03/reco/P00hi/2000/09","*.mdst.root");
```

The `mDstRead.C` macro in the library actually calls the standard StAnalysisMaker. Of course one can also create a macro which reads in a miniDSTs and writes out another, more compressed version of it. Please note that `StMiniDstMaker`, `mDstWrite.C`, and `mDstRead.C` are only examples and should be used as templates for you own purposes.

4.3 Advanced features

4.3.1 Using Zombies

Most methods of the `StEventScavanger` class are based on the 'zombie' mechanism introduced by Victor in Sep 2000. The scheme relies on the fact that all StEvent objects inherit from `StObject` and `StObject` itself has a few free user bits which can be used to mark the object. Once the object is marked it becomes a zombie and will be ignored when the rest of StEvent is written to the miniDST. The object can be marked by invoking the `void StObject::makeZombie()` method and the state of each object can be checked with the `bool StObject::isZombie()` method.

If you feel that your understanding of StEvent is sufficient enough you always can use the zombie mechanism directly. Once an object becomes a zombie also all references to the object will disappear on the miniDST, i.e. pointer are nulled, at least. In this sense zombies are pretty save to use.

Please note, that - as in the bad movies - zombies are visible. After an object is made a zombie you still can use the same way as before. With other words it does not get deleted in memory.

4.3.2 Adding user defined classes

The miniDST mechanism is in principle not restricted to StEvent classes only. Objects of every class which inherits from `StObject` and use the `ClassDef()` and `ClassImp()` macros can be written to a miniDST. You will need of course the class definition when you read the data back in. If the class definition is not availabe the object cannot be retrieved from the miniDST; you still we be able to read the rest though. Needles to say that it is the responsibility of the user to make sure the class is available to everyone who tries to read the miniDSTs.

Here's a very simple example:

```
// StMyDstClass.h
#include "StObject.h"
class StMyDstClass : public StObject {
public:
    StMyDstClass();
    int myCounter;
    int eventIsGood;
    ClassDef(StMyDstClass,1);
};
```

The `StMyDstClass.cxx` file of course has to exist and should contain

```
// StMyDstClass.h
#include "StMyDstClass.h"
ClassImp();
StMyDstClass::StMyDstClass()
{
    myCounter = 0;
    eventIsGood = 0;
}
```

All you have to do is to create an instance of this class in the `Make()` method of your maker (`StMiniDstMaker`) and attach it to StEvent

```
StMyDstClass *my = new StMyDstClass;
my->myCounter = 17;
my->eventIsGood = 1;
event->content().push_back(my);
```

From here on the `StMyDstClass` object is integral part of StEvent and will be treated as such during I/O. Please not that you are of course not limited to build in data types but can more complex ones as pointers to `StTrack` objects for example. One could imagine to have a class `StMyPair` which links two tracks which form a pair to name just one of many applications.

Part II

Reference Manual

5 Class References

The classes which are currently implemented and available from the STAR CVS repository are listed in alphabetic order.

Inherited member functions and operators are not described in the reference section of a derived class. Always check the section(s) of the base class(es) to get a complete overview on the available methods.

In general each class has a public:

- Default constructor
- Copy constructor
- Assignment operator
- Virtual destructor

There are a few exceptions from this rule which are explained in the referring class reference.

Not every member function listed is explained in detail since many are trivial and their names are chosen such that one can easily figure out what they are all about. Macros and `Inline` declarations are omitted throughout the documentation and so is the `virtual` keyword. The state-of-the-art reference is always the class definition in the header file.

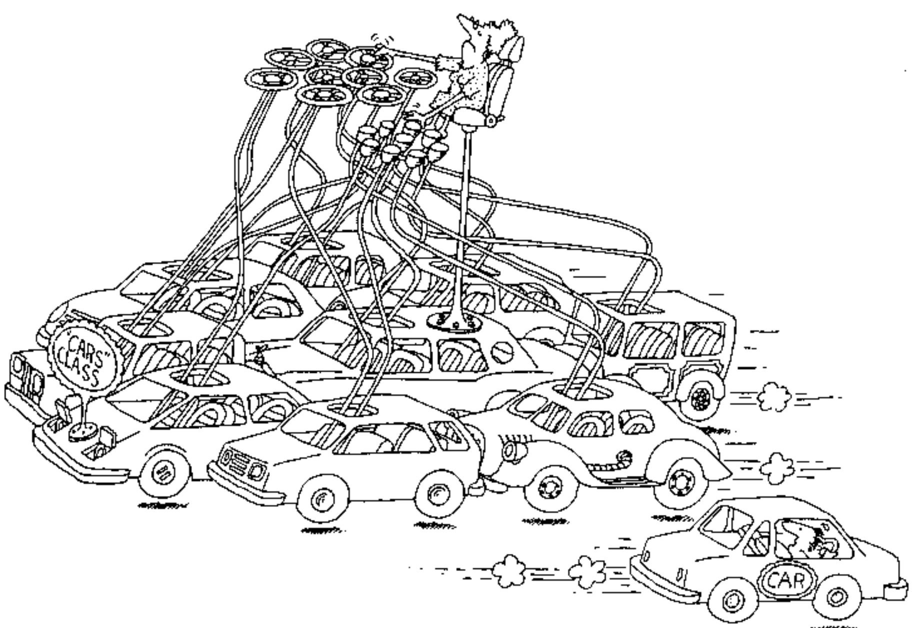

Figure 5.1: A class represents a set of objects that share a common structure and a common behavior.

5.1 StBbcTriggerDetector

Summary

Synopsis

```
#include "StBbcTriggerDetector.h"
class StBbcTriggerDetector;
```

Description

Related Classes

Public Constructors

```
StBbcTriggerDetector();
```

Public Member Functions

```
unsigned int numberOfPMTs() const;
unsigned int numberOfRegisters() const;
unsigned int numberOfPedestalData() const;
unsigned int numberOfScalars() const;
unsigned short adc(unsigned int) const;
unsigned short tdc(unsigned int) const;
unsigned short bbcRegister(unsigned int) const;
unsigned short pedestalData(unsigned int) const;
unsigned int scalar(unsigned int) const;
unsigned short pedestal(unsigned int id) const;
unsigned short pedestalWidth(unsigned int id) const;
unsigned short mip(unsigned int id) const;
unsigned short mipWidth(unsigned int id) const;
int nHitEast();
int nHitWest();
int nHitAll();
int adcSumEast();
int adcSumWest();
int adcSumAll();
float zVertex(); //z vertex in cm
void setAdc(unsigned int, unsigned short);
void setTdc(unsigned int, unsigned short);
void setRegister(unsigned int, unsigned short);
void setPedestal(unsigned int, unsigned short);
void setScalar(unsigned int, unsigned int);
void dump();
```

5.2 StCalibrationVertex

Summary Represents vertices used for test and calibration purposes.

Synopsis
```
#include "StCalibrationVertex.h"
class StCalibrationVertex;
```

Description Concrete implementatin of the `StVertex` class. It represents various types of vertices useful for calibration and diagnostics. These vertices have no daughters and no parent. All vertices of this category are of type `kOtherVtxId` (see `StVertex::type()`). Don't worry if you don't what they are good for. There are nor relevant for physics analysis.

Related Classes Inherits from `StVertex`. `StCalibrationVertex` doesn't add any new data member of methods.

Public Constructors
```
StCalibrationVertex();
StCalibrationVertex(const dst_vertex_st&);
```

Public Member Functions See `StVertex` (5.108) for available methods.

5.3 StContainers

Summary Definitions of all container types used in StEvent.

Synopsis `#include "StContainers.h"`

Description `StContainers.h` includes `StArray.h` which contains the guts of the container implementation. In StContainer.h (and .cxx) the appropriate macros are called to declare and define the container types. If a new container type has to be defined it *must* be defined here and only here.

5.4 StCtbSoftwareMonitor

Summary Monitors details of the Central Trigger Barrel (CTB) reconstruction.

Synopsis

```
#include "StCtbSoftwareMonitor.h"
class StCtbSoftwareMonitor;
```

Description

Related Classes

Public Constructors

```
StCtbSoftwareMonitor();
```

```
StCtbSoftwareMonitor(const dst_mon_soft_ctb_st&);
```

Public Data Member

```
int mult_ctb_tot;
```

Total multiplicity (or ADC sum) in CTB.

5.5 StCtbTriggerDetector

Summary Interface to CTB event data.

Synopsis

```
#include "StCtbTriggerDetector.h"
class StCtbTriggerDetector;
```

Description

Related Classes Inherits from StObject.

Public Constructors

```
StCtbTriggerDetector();
StCtbTriggerDetector(const dst_TrgDet_st&);
```

Public Member Functions

`unsigned int numberOfTrays() const;`
Returns number of CTB trays (usually 120).

`unsigned int numberOfSlats() const;`
Returns number of slats (usually 2).

`unsigned int numberOfPreSamples() const;`
Number of pre-samples taken.

`unsigned int numberOfPostSamples() const;`
Number of post-samples taken.

`unsigned int numberOfAuxWords() const;`
Number of auxiliary words (usually 16). No physical significance at present, these data are simple reserved for possible future use.

`float mips(unsigned int tr, unsigned int sl, unsigned int evt = 0) co`
Return MIPS for tray `tr` and slot `sl`. The trays run from 0 – `numberOfTrays()`-1, the slots from 0 – `numberOfSlots()`-1. The first half of the tray numbers are for the west side, the second half for the east side (usually 0-59 on west end, 60-119 on east end). Each tray has 2 slats, 0 is the inner slat, closest to the central membrane, and 1 is the outer slat, further from the central membrane. The last argument `evt` has the following meaning:
0 is the triggered event, and from 1 to `(npresamples+npostsamples)` are the other "events", in chronological order. For example, assuming 5 pre events and 5 post events:

evt = 0 triggered event at t = 0

evt = 1 1st pre event at t = -5

evt = 2 2nd pre event at t = -4

evt = 3 3rd pre event at t = -3

evt = 4 1st post event at t = -2

evt = 5 2nd post event at t = -1

evt = 6 2nd post event at t = +1

evt = 7 2nd post event at t = +2

evt = 8 2nd post event at t = +3

evt = 9 2nd post event at t = +4

evt = 10 2nd post event at t = +5

`char time(unsigned int tray, unsigned int slot, unsigned int evt = 0) con`
Same arguments as for `mips()` (see above).

`float aux(unsigned int, unsigned int evt = 0) const;`
Argument `i` has no physical significance at present, these data are simple reserved (on request of Hank Crawford) for possible future use. For the second argument (`evt`) see `mips()` above.

```
void setMips(unsigned int, unsigned int, unsigned int, float);
void setTime(unsigned int, unsigned int, unsigned int, char);
void setAux(unsigned int, unsigned int, float);
void setNumberOfPreSamples(unsigned int);
void setNumberOfPostSamples(unsigned int);
```

5.6 StDedxPidTraits

Summary

Synopsis

```
#include "StDedxPidTraits.h"
class StDedxPidTraits;
```

Description

Related Classes

Public Constructors

```
StDedxPidTraits();
```

Default constructor.

```
StDedxPidTraits(StDetectorId det, short emethod,
                unsigned short np, float dedx, float sig);
```

Create an instance of `StDedxPidTraits` for detector `det`, encoded method `emethod`, number of points `np`, dE/dx mean `dedx`, and error on mean `sig`.

Public Member Functions

```
unsigned short numberOfPoints() const;
```

Number of points used to calculate the dE/dx value.

```
float mean() const;
```

The derived dE/dx value.

```
float length() const;
```

Track length used in dE/dx calculations.

```
float errorOnMean() const;
```

Returns the error on the dE/dx value.

```
StDedxMethod method() const;
```

```
short encodedMethod() const;
```

5.7 StDetectorState

Summary Base class for detector state information.

Synopsis

```
#include "StDetectorState.h"
class StDetectorState;
```

Description `StDetectorState` is a non-abstract base class to hold information on the "state" of a detector. All concrete classes reflecting the state of actual detectors need to inherit from this class in order to allow storage in `StEvent`. The only data member it contains are the detector ID and a Boolean to reflect the overall state (good/bad). More detailed information may be added by the derived classes.

Related Classes

Public Constructors

```
StDetectorState();
StDetectorState(StDetectorId, bool);
```

Public Member Functions

`StDetectorId detector() const;`
Returns detector ID (see 2.2).

`bool good() const;`
Return `true` if the overall state of the detector is good, otherwise `false`.

`bool bad() const;`
Returns the negative of `good()`.

```
void setDetector(StDetectorId);
void setGood(bool);
```

5.8 StEmcCluster

Summary

Synopsis

```
#include "StEmcCluster.h"
class StEmcCluster;
```

Description

Related Classes

Public Constructors

```
StEmcCluster();
```

Public Member Functions

```
float eta() const;
float phi() const;
float sigmaEta() const;
float sigmaPhi() const;
float energy() const;
int nHits() const;
int nNeighbors() const;
int nTracks() const;
StPtrVecEmcRawHit& hit();
const StPtrVecEmcRawHit& hit() const;
StPtrVecEmcCluster& neighbor();
const StPtrVecEmcCluster& neighbor() const;
StPtrVecTrack& track();
const StPtrVecTrack& track() const;
void setEta(float);
void setPhi(float);
void setSigmaEta(float);
void setSigmaPhi(float);
void setEnergy(float);
void addHit(StEmcRawHit*);
void addNeighbor(StEmcCluster*);
void addTrack(StTrack*);
void print(ostream& os = cout) const;
```

5.9 StEmcClusterCollection

Summary

Synopsis

```
#include "StEmcClusterCollection.h"
class StEmcClusterCollection;
```

Description

Related Classes

Public Constructors

```
StEmcClusterCollection();
```

Public Member Functions

```
StDetectorId detector() const;
void setDetector(StDetectorId);
int numberOfClusters() const;
StSPtrVecEmcCluster& clusters();
const StSPtrVecEmcCluster& clusters() const;
void addCluster(StEmcCluster*);
int clusterFinderId() const;
int clusterFinderParamVersion() const;
void setClusterFinderId(int);
void setClusterFinderParamVersion(int);
```

5.10 StEmcCollection

Summary

Synopsis

```
#include "StEmcCollection.h"
class StEmcCollection;
```

Description

Related Classes

Public Constructors

```
StEmcCollection();
```

Public Member Functions

```
StEmcDetector* detector(StDetectorId);
const StEmcDetector* detector(StDetectorId) const;
StSPtrVecEmcPoint& barrelPoints();
const StSPtrVecEmcPoint& barrelPoints() const;
StSPtrVecEmcPoint& endcapPoints();
const StSPtrVecEmcPoint& endcapPoints() const;
void addBarrelPoint(const StEmcPoint*);
void addEndcapPoint(const StEmcPoint*);
void setDetector(StEmcDetector*);
```

5.11 StEmcDetector

Summary

Synopsis

```
#include "StEmcDetector.h"
class StEmcDetector;
```

Description

Related Classes

Public Constructors

```
StEmcDetector();
StEmcDetector(StDetectorId, unsigned int);
```

Public Member Functions

```
StDetectorId detectorId() const;
unsigned int numberOfModules() const;
bool addHit(StEmcRawHit*);
unsigned int numberOfHits() const;
StEmcModule* module(unsigned int);
const StEmcModule* module(unsigned int) const;
StEmcClusterCollection* cluster();
const StEmcClusterCollection* cluster() const;
void setCluster(StEmcClusterCollection*);
```

5.12 StEmcModule

Summary

Synopsis

```
#include "StEmcModule.h"
class StEmcModule;
```

Description

Related Classes

Public Constructors

```
StEmcModule();
```

Public Member Functions

```
unsigned int numberOfHits() const;
StSPtrVecEmcRawHit& hits();
const StSPtrVecEmcRawHit& hits() const;
```

5.13 StEmcPoint

Summary

Synopsis

```
#include "StEmcPoint.h"
class StEmcPoint;
```

Description

Related Classes

Public Constructors

```
StEmcPoint();
StEmcPoint(const StThreeVectorF&,
           const StThreeVectorF&,
           const StThreeVectorF&,
           unsigned int, float,
           float, float,
           unsigned char = 0);
```

Public Member Functions

```
float energy() const;
float chiSquare() const;
void setEnergy(const float);
void setChiSquare(const float);
StThreeVectorF size() const;
void setSize(const StThreeVectorF&);
float energyInDetector(const StDetectorId) const;
float sizeAtDetector(const StDetectorId) const;
void setEnergyInDetector(const StDetectorId, const float);
void setSizeAtDetector(const StDetectorId, const float);
StPtrVecEmcCluster& cluster(const StDetectorId);
const StPtrVecEmcCluster& cluster(const StDetectorId) const;
void addCluster(const StDetectorId, const StEmcCluster*);
StPtrVecEmcPoint& neighbor();
const StPtrVecEmcPoint& neighbor() const;
void addNeighbor(const StEmcPoint*);
int nTracks() const;
StPtrVecTrack& track();
const StPtrVecTrack& track() const;
void addTrack(StTrack*);
```

5.14 StEmcRawHit

Summary

Synopsis

```
#include "StEmcRawHit.h"
class StEmcRawHit;
```

Description

Related Classes

Public Constructors

```
StEmcRawHit();
StEmcRawHit(StDetectorId, unsigned int, unsigned int, unsigned int, u
StEmcRawHit(StDetectorId, unsigned int, unsigned int, unsigned int, u
```

Public Member Functions

```
StDetectorId detector() const;
unsigned int module() const;
unsigned int eta() const;
unsigned int sub() const;
unsigned int adc() const;
float energy() const;
void setId(StDetectorId, unsigned int, unsigned int, unsigned int);
void setAdc(const unsigned int);
void setEnergy(const float);
```

5.15 StEmcSoftwareMonitor

Summary

Synopsis

```
#include "StEmcSoftwareMonitor.h"
class StEmcSoftwareMonitor;
```

Description

Related Classes

Public Constructors

```
StEmcSoftwareMonitor();
StEmcSoftwareMonitor(const dst_mon_soft_emc_st&);
```

Public Data Member

```
float energy_emc;
```

Total energy (or ADC sum) in EMC.

5.16 StEnumerations

Summary Header file which contains all enumeration types used in StEvent.

Synopsis `#include "StEnumerations.h"`

Description All enumeration types used in StEvent are defined in this header file. It also includes other header files which are common to all STAR code. For a complete list of enum types see section 2.2.

5.17 StEmcTriggerDetector

Summary

Synopsis

```
#include "StEmcTriggerDetector.h"
class StEmcTriggerDetector;
```

Description

Related Classes

Public Constructors

```
StEmcTriggerDetector();
StEmcTriggerDetector(const dst_TrgDet_st&);
```

Public Member Functions

```
int numberOfTowers() const;
int highTower(unsigned int) const;
int patch(unsigned int) const;
void setHighTower(unsigned int, int);
void setPatch(unsigned int, int);
```

5.18 StEvent

Summary

Event header and entry point to the StEvent tree.

Synopsis

```
#include "StEvent.h"
class StEvent;
```

Description

The class `StEvent` is the key class to work with the whole StEvent tree. It itself contains data which describes and characterizes the event and gives references and pointers to all information there is in the event. Don't forget to check for `NULL` pointers if a method returns an object by pointer. Only if a method returns an object by reference it is guaranteed to exist.
The package StEventMaker (see Sec. 2.6) provides a pointer to the current instance of `StEvent`.

Related Classes

Class `StEvent` inherits from `St_DataSet`.

Public Constructors

```
StEvent();

StEvent(const event_header_st&,
        const dst_event_summary_st&,
        const dst_summary_param_st&);

StEvent(const event_header_st&);
```

Public Member Functions

`void Browse(TBrowser*);`
Overwrite inherited `Browse()` method from `St_DataSet`. Method is invoked from ROOT to allow interactive browsing of StEvent.

`static const TString& cvsTag();`
CVS tag of the version you are using.

`void statistics() const;`
Prints information to `cout`, as number of tracks, hits, vertices, event and run ID, time, and more. This method can be also invoked from the ROOT GUI.

`const TString& type() const;`
Character string which contains a short description of the type of the event you got.

`int id() const;`
Unique event identifier.

`int runId() const;`
Unique run identifier.

`int time() const;`
Time when the event was taken. The format depends pretty much on the DST version. Older DST version return HHMMSS while newer runs will return the standard UNIX time. For the latter one then is able to use the many UNIX standard tools available for this time format (e.g. `ctime()`).

`unsigned int triggerMask() const;`

`unsigned int bunchCrossingNumber(Uint_t i) const;`
Returns the bunch crossing numbers, i.e. two 32 bit words. `i = 0` returns the lower and `i = 1` the upper number.

`StEventSummary* summary();`
`const StEventSummary* summary() const;`
Returns pointer to the event summary with many useful information for QA/QC and event characterization.

`StEventInfo* info();`
`const StEventInfo* info() const;`
Returns a pointer to the one and only instance of `StEventInfo`. `StEventInfo` provides no additional information but is solely used to hold the header data. Some info can also be directly obtained from StEvent. See also 5.19.

`StRunInfo* runInfo();`
`const StRunInfo* runInfo() const;`
Returns a pointer to the one and only instance of `StRunInfo`. `StRunInfo` contains parameters related to the current run. See also 5.63.

`StSoftwareMonitor* softwareMonitor();`
`const StSoftwareMonitor* softwareMonitor() const;`
Returns pointer to the software-monitor collection. This class holds “monitors” for every detector which contain information gathered during the event reconstruction. Mostly statistic on number of hits, tracks etc.

`StTpcHitCollection* tpcHitCollection();`
`const StTpcHitCollection* tpcHitCollection() const;`
Pointer to the TPC hit collection. If no hits are stored on the DST this pointer is `NULL`. You better check for this.

`StFtpcHitCollection* ftpcHitCollection();`
`const StFtpcHitCollection* ftpcHitCollection() const;`
Pointer to the FTPC hit collection. If no hits are stored on the DST this pointer is `NULL`. You better check for this.

`StSvtHitCollection* svtHitCollection();`
`const StSvtHitCollection* svtHitCollection() const;`
Pointer to the SVT hit collection. If no hits are stored on the DST this pointer is `NULL`. You better check for this.

`StSsdHitCollection* ssdHitCollection();`
`const StSsdHitCollection* ssdHitCollection() const;`
Pointer to the SSD hit collection. If no hits are stored on the DST this pointer is `NULL`. You better check for this.

`StRichCollection* richCollection();`
`const StRichCollection* richCollection() const;`

`StTofCollection* tofCollection();`
`const StTofCollection* tofCollection() const;`

`StFpdCollection* fpdCollection();`
`const StFpdCollection* fpdCollection() const;`

`StPhmdCollection* phmdCollection();`
`const StPhmdCollection* phmdCollection() const;`

`StL0Trigger* l0Trigger();`
`const StL0Trigger* l0Trigger() const;`

```
StL3Trigger* l3Trigger();
const StL3Trigger* l3Trigger() const;
```

```
StTriggerDetectorCollection* triggerDetectorCollection();
const StTriggerDetectorCollection* triggerDetectorCollection() const;
```

Returns pointer to the current trigger detector collection. Trigger detectors are CTB, ZDC, VPD, and MWC.

```
StTriggerIdCollection* triggerIdCollection();
const StTriggerIdCollection* triggerIdCollection() const;
```

Contains trigger IDs (summaries) starting at RUN III (2003).

```
StTriggerData* triggerData();
const StTriggerData* triggerData() const;
```

Contains all trigger information. Implemented for Run III (2003).

```
StTriggerDetectorCollection* triggerDetectorCollection();
const StTriggerDetectorCollection* triggerDetectorCollection() const;
```

```
StSPtrVecTrackDetectorInfo& trackDetectorInfo();
const StSPtrVecTrackDetectorInfo& trackDetectorInfo() const;
```

```
StSPtrVecTrackNode& trackNodes();
const StSPtrVecTrackNode& trackNodes() const;
```

```
unsigned int numberOfPrimaryVertices() const;
```

Number of primary vertices (aka event vertices). Usually there is only one but future implementations of the vertex finder will be able to also detect pile-up vertices in which case you better check the number before dealing with the event.

```
StPrimaryVertex* primaryVertex(unsigned int i = 0);
const StPrimaryVertex* primaryVertex(unsigned int i = 0) const;
```

Returns pointer to the `i`'th primary vertex. Since in most of the cases there is only one primary vertex `i` defaults to the first (`i=0`).
The primary vertices are ordered according to the number of daughters (i.e. primary tracks) they hold. The first in the list is always the the vertex with the most daughter tracks.

```
unsigned int numberOfCalibrationVertices() const;
```

Number of vertices stored for calibration and test purposes.

```
StCalibrationVertex* calibrationVertex(unsigned int i);
const StCalibrationVertex* calibrationVertex(unsigned int i) const;
```

Returns pointer to the `i`'th 'calibration' vertex. Calibration vertices are of type `StCalibrationVertex` (see 5.2) and are mainly used for test, calibration, and diagnosis. Not relevant for physics analysis.

```
StSPtrVecV0Vertex& v0Vertices();
const StSPtrVecV0Vertex& v0Vertices() const;
```

Returns container with V0 vertices.

```
StSPtrVecXiVertex& xiVertices();
const StSPtrVecXiVertex& xiVertices() const;
```

Returns container with Xi vertices.

```
StSPtrVecKinkVertex& kinkVertices();
const StSPtrVecKinkVertex& kinkVertices() const;
```

Returns container with kink vertices.

`StSPtrVecObject& content();`
Returns the content of `StEvent`. Use with great care!
The class `StEvent` itself is, technically spoken, not much more than a container of objects of type `StObject`. It is the level of sophistication in the member functions which gives the class its look-and-feel. Any class inheriting from `StObject` can be added to StEvent and such be made persistent. To add use:
`StEvent::content().push_back(StObject*);`
Of course it is now also your job to retrieve the object from the list. This is best done using the C++ RTTI mechanism.

`StDetectorState*  detectorState(StDetectorId det);`
`const StDetectorState*  detectorState(StDetectorId det) const;`
Returns pointer to detector "state" object referring to detector `det`. For `StDetectorId` see 2.2. Note that returned pointer might point to a class derived from `StDetectorState`. In this case a `dynamic_cast` is required unless you are only interested in the overall detector state which is obtained via `StDetectorState::good()`. If no "state" object for the referring detector is available a NULL pointer is returned.

`StPsd*  psd(StPwg pwg, int id);`
`const StPsd* psd(StPwg pwg, int id) const;`
Retrieve Physics Summary Data (PSD) from physics working group `pwg` (for `StPwg` see 2.2) with identifier `id`. Returns `null` if not present.

`unsigned int numberOfPsds();`
Returns number of all PSDs stored in `StEvent`.

`unsigned int numberOfPsds(StPwg pwg);`
Returns number of all PSDs from Physics Working Group `pwg` stored in `StEvent`.

`void setType(const char*);`

`void setRunId(int);`

`void setId(int);`

`void setTime(int);`

`void setTriggerMask(unsigned int);`

`void setBunchCrossingNumber(unsigned int, unsigned int);`

`void setSummary(StEventSummary*);`

`void setInfo(StEventInfo*);`

`void setRunInfo(StRunInfo*);`

`void setSoftwareMonitor(StSoftwareMonitor*);`

`void setTpcHitCollection(StTpcHitCollection*);`

`void setFtpcHitCollection(StFtpcHitCollection*);`

`void setSvtHitCollection(StSvtHitCollection*);`

`void setRichCollection(StRichCollection*);`

`void setTofCollection(StTofCollection*);`

`void setFpdCollection(StFpdCollection*);`

`void setPhmdCollection(StPhmdCollection*);`

`void setTriggerDetectorCollection(StTriggerDetectorCollection*);`

`void setTriggerIdCollection(StTriggerIdCollection*);`

`void setTriggerData(StTriggerData*);`

`void setL0Trigger(StL0Trigger*);`

`void setL3Trigger(StL3Trigger*);`

`void addPrimaryVertex(StPrimaryVertex*);`
Adds a primary vertex to the list. The vertices are automatically sorted according to their number of daughter tracks (descending order).

`void addCalibrationVertex(StCalibrationVertex*);`

`void addDetectorState(StDetectorState*);`

`void addPsd(StPsd* psd);`
Add PSD `*psd` to StEvent. This method checks if another PSD with identical identifiers is already stored. If so, it prints an error message and doesn't add the object pointed to by `psd`. Note that after this call the object is owned by StEvent. Don't delete it. However, you can still modify it.

`void removePsd(StPsd* psd);`
Removes PSD pointed to by `psd`. Please note that `psd` itself is not nulled. The pointer is not valid anymore once this method was invoked.

5.19 StEventInfo

Summary StEvent header info

Synopsis

```
#include "StEventInfo.h"
class StEventInfo;
```

Description Please note that some information this class provides is already accessible directly through the `StEvent` class itself. This class is used hold general event related information within `StEvent` such as event and run number, event size, and the time the event was recorded. For more information see 5.18.

Related Classes

Public Constructors

```
StEventInfo();
StEventInfo(const event_header_st&);
```

Public Member Functions

```
const TString& type() const;
int id() const;
int runId() const;
int time() const;
unsigned int triggerMask() const;
unsigned int eventSize() const;
```

```
unsigned int bunchCrossingNumber(Uint_t i) const;
```

Returns the bunch crossing numbers, i.e. two 32 bit words. `i = 0` returns the lower and `i = 1` the upper number.

```
void setType(const char*);
void setRunId(int);
void setId(int);
void setTime(int);
void setTriggerMask(unsigned int);
void setBunchCrossingNumber(unsigned int, unsigned int);
void setEventSize();
```

5.20 StEventScavenger

Summary Helper class to ease creating miniDSTs based on StEvent

Synopsis

```
#include "StEventScavenger.h"
class StEventScavenger;
```

Description The class contains only static member functions. These methods allow to delete, or better mark, classes which you do **not** want to end up on the miniDSTs. For more see section 4.

Related Classes none

Public Constructors

```
static bool removeEventSummary(StEvent*);
static bool removeSoftwareMonitor(StEvent*);
static bool removeTpcHitCollection(StEvent*);
static bool removeFtpcHitCollection(StEvent*);
static bool removeSvtHitCollection(StEvent*);
static bool removeSsdHitCollection(StEvent*);
static bool removeEmcCollection(StEvent*);
static bool removeRichCollection(StEvent*);
static bool removeTriggerDetectorCollection(StEvent*);
static bool removeL3Trigger(StEvent*);
static bool removeV0Vertices(StEvent*);
static bool removeXiVertices(StEvent*);
static bool removeKinkVertices(StEvent*);
static bool remove(StTrack*);
static bool removeFpdCollection(StEvent*);
static bool removeToFCollection(StEvent*);
static bool removeCalibrationVertices(StEvent*);
```

`removeTpcHitsNotOnTrack(StEvent*);`
Marks all TPC hits not associated with valid tracks. 'Valid' means that the track will be written to the miniDST (i.e. the track exist and is not a zombie). This method has to be called after the rejected tracks are removed. Do not remove the TPC hits via `removeTpcHitCollection()` if you use this method.

5.21 StEventSummary

Summary

Synopsis

```
#include "StEventSummary.h"
class StEventSummary;
```

Description

Related Classes

Public Constructors

```
StEventSummary();
StEventSummary(const dst_event_summary_st&,
               const dst_summary_param_st&);
```

Public Member Functions

```
int numberOfTracks() const;
int numberOfGoodTracks() const;
int numberOfGoodTracks(StChargeSign) const;
int numberOfGoodPrimaryTracks() const;
int numberOfExoticTracks() const;
int numberOfVertices() const;
int numberOfVerticesOfType(StVertexId) const;
int numberOfPileupVertices() const;
float meanPt() const;
float meanPt2() const;
float meanEta() const;
float rmsEta() const;
const StThreeVectorF& primaryVertexPosition() const;
unsigned int numberOfBins() const;
int tracksInEtaBin(unsigned int) const;
int tracksInPhiBin(unsigned int) const;
int tracksInPtBin(unsigned int) const;
float energyInEtaBin(unsigned int) const;
float energyInPhiBin(unsigned int) const;
float lowerEdgeEtaBin(unsigned int) const;
float upperEdgeEtaBin(unsigned int) const;
float lowerEdgePhiBin(unsigned int) const;
float upperEdgePhiBin(unsigned int) const;
float lowerEdgePtBin(unsigned int) const;
float upperEdgePtBin(unsigned int) const;
```

```
double magneticField() const;
```

Returns z-component of magnetic field in kGauss.

```
void setNumberOfTracks(int);
void setNumberOfGoodTracks(int);
void setNumberOfGoodTracks(StChargeSign, int);
void setNumberOfGoodPrimaryTracks(int);
void setNumberOfNegativeTracks(int);
void setNumberOfExoticTracks(int);
void setNumberOfVertices(int);
void setNumberOfVerticesForType(StVertexId, int);
```

```
void setNumberOfPileupVertices(int);
void setMeanPt(float);
void setMeanPt2(float);
void setMeanEta(float);
void setRmsEta(float);
void setPrimaryVertexPosition(const StThreeVectorF&);
void setMagneticField(double);
```

5.22 StEventTypes

Summary Header files which contains all type definition used in StEvent.

Synopsis `#include "StEventTypes.h"`

Description Since all StEvent classes contain only the minimum amount of declaration it could become very tedious to find the right set of header files in your application. This header files overcomes this problem. Include it and you are all set. See also section 2.1.

5.23 StFpdCollection

Summary

Synopsis

```
#include "StFpdCollection.h"
class StFpdCollection;
```

Description

Related Classes

Public Constructors

```
StFpdCollection();
```

Public Member Functions

```
unsigned int numberOfADC() const;
unsigned int numberOfTDC() const;
unsigned int numberOfRegisters() const;
unsigned int numberOfPedestal() const;
unsigned short* adc();
unsigned short* tdc();
unsigned short registers(unsigned int) const;
unsigned short* pedestal();
int token() const;
unsigned short north(unsigned int);
unsigned short south(unsigned int);
unsigned short top(unsigned int);
unsigned short bottom(unsigned int);
unsigned short smdx(unsigned int);
unsigned short smdy(unsigned int);
unsigned short pres1(unsigned int);
unsigned short pres2(unsigned int);
void setAdc(unsigned int, unsigned short);
void setTdc(unsigned int, unsigned short);
void setRegister(unsigned int, unsigned short);
void setPedestal(unsigned int, unsigned short);
void dump();
```

5.24 StFtpcHit

Summary

Synopsis

```
#include "StFtpcHit.h"
class StFtpcHit;
```

Description

Related Classes

Public Constructors

```
StFtpcHit();
```

```
StFtpcHit(const StThreeVectorF&,
          const StThreeVectorF&,
          unsigned int, float, unsigned char = 0);
```

```
StFtpcHit(const dst_point_st&);
```

Public Member Functions

```
unsigned int sector() const;
```

Returns sector number running from 1–6.

```
unsigned int plane() const;
```

Returns plane number running from 1–20.

```
unsigned int padsInHit() const;
unsigned int timebinsInHit() const;
```

5.25 StFtpcHitCollection

Summary

Synopsis

```
#include "StFtpcHitCollection.h"
class StFtpcHitCollection;
```

Description

Related Classes

Public Constructors

```
StFtpcHitCollection();
```

Public Member Functions

```
bool addHit(StFtpcHit*);
```

```
unsigned int numberOfHits() const;
```

Total number of FTPC hits stored in the collection.

```
unsigned int numberOfPlanes() const;
```

```
StFtpcPlaneHitCollection* plane(unsigned int i);
const StFtpcPlaneHitCollection* plane(unsigned int i) const;
```

Index `i` runs from 0–(n-1) where n = `numberOfPlanes()`.

5.26 StFtpcPlaneHitCollection

Summary

Synopsis

```
#include "StFtpcPlaneHitCollection.h"
class StFtpcPlaneHitCollection;
```

Description

Related Classes

Instance of `StFtpcPlaneHitCollection` are stored in the `StFtpcHitCollection`. The class holds a list of objects of type `StFtpcSectorHitCollection`.

Public Constructors

`StFtpcPlaneHitCollection();`
Default constructor.

Public Member Functions

`unsigned int numberOfHits() const;`
Number of hits stored in this FTPC plane.

`unsigned int numberOfSectors() const;`
Number of sectors in this FTPC plane.

`StFtpcSectorHitCollection* sector(unsigned int i);`
`const StFtpcSectorHitCollection* sector(unsigned int i) const;`
Returns the `i`’th sector, where `i = 0--(numberOfSectors()-1)`.

5.27 StFtpcSectorHitCollection

Summary

Synopsis

```
#include "StFtpcSectorHitCollection.h"
class StFtpcSectorHitCollection;
```

Description

Related Classes

Public Constructors

```
StFtpcSectorHitCollection();
```

Public Member Functions

```
StSPtrVecFtpcHit& hits();
```

```
const StSPtrVecFtpcHit& hits() const;
```

5.28 StFtpcSoftwareMonitor

Summary

Synopsis

```
#include "StFtpcSoftwareMonitor.h"
class StFtpcSoftwareMonitor;
```

Description

Related Classes

Public Constructors

```
StFtpcSoftwareMonitor();
```

```
StFtpcSoftwareMonitor(const dst_mon_soft_ftpc_st&);
```

Public Data Member

```
int n_clus_ftpc[2];
```

Total number of clusters in FTPC, east/west.

```
int n_pts_ftpc[2];
```

Total number of space points in FTPC, east/west.

```
int n_trk_ftpc[2];
```

Total number of tracks in FTPC east/west .

```
float chrg_ftpc_tot[2];
```

Total charge deposited in FTPC, east/west.

```
float hit_frac_ftpc[2];
```

Fraction of hits used in FTPC, east/west.

```
float avg_trkL_ftpc[2];
```

Average track length (cm) FTPC, east/west
or average number of points assigned.

```
float res_pad_ftpc[2];
```

Average residual, pad direction, FTPC east/west.

```
float res_drf_ftpc[2];
```

Average residual, drift direction, FTPC east/west.

5.29 StFunctional

Summary

Synopsis `#include "StFunctional.h"`

Description

5.30 StGlobalSoftwareMonitor

Summary

Synopsis

```
#include "StGlobalSoftwareMonitor.h"
class StGlobalSoftwareMonitor;
```

Description

Related Classes

Public Constructors

```
StGlobalSoftwareMonitor();
StGlobalSoftwareMonitor(const dst_mon_soft_glob_st&);
```

Public Data Member

`int n_trk_match[2];`
Total number of SVT-TPC tracks matched with tan(dip angle) < 0 (≥ 0).

`int prim_vrtx_ntrk;`
Number of tracks used in primary vertex fit.

`float prim_vrtx_cov[6];`
Primary vertex covariance matrix.

`float prim_vrtx_chisq;`
Primary vertex χ^2 of fit.

5.31 StGlobalTrack

Summary

Synopsis

```
#include "StGlobalTrack.h"
class StGlobalTrack;
```

Description

Related Classes `StGlobalTrack` is derived from `StTrack`. See also `StPrimaryTrack`.

Public Constructors

```
StGlobalTrack();
StGlobalTrack(const dst_track_st&);
StGlobalTrack(const StGlobalTrack&);
StGlobalTrack& operator=(const StGlobalTrack&);
```

Public Member Functions

```
StTrackType type() const;
```

Returns always `global`.

```
const StVertex* vertex() const;
```

Returns always `null`.

5.32 StHelixModel

Summary

Synopsis

```
#include "StHelixModel.h"
class StHelixModel;
```

Description

Related Classes Inherits directly from `StTrackGeometry`.

Public Constructors

```
StHelixModel();
```

```
StHelixModel(short q, float psi, float c, float dip,
             const StThreeVectorF& o, const StThreeVectorF& p);
```

```
StHelixModel(const dst_track_st&);
```

Public Member Functions

```
StTrackModel model() const;
```

```
short charge() const;
```

Charge in units of +e.

```
double curvature() const;
```

Curvature in cm^{-1}.

```
double psi() const;
```

Psi in radians.

```
double dipAngle() const;
```

Dip angle in radians.

```
const StThreeVectorF& origin() const;
```

Origin in cm.

```
const StThreeVectorF& momentum() const;
```

Momentum in GeV/c.

```
StPhysicalHelixD helix() const;
```

5.33 StHit

Summary Base class for hits.

Synopsis

```
#include "StHit.h"
class StHit;
```

Description

Related Classes `StHit` is derived form `StMeasuredPoint`. `StTpcHit`, `StSvtHit`, `StSsdHit`, `StRichHit`, `StFtpcHit`, and `StEmcPoint` inherit from this class.

Public Constructors

```
StHit();
StHit(const StThreeVectorF&,
      const StThreeVectorF&,
      unsigned int, float, unsigned char = 0);
```

Public Member Functions

`float charge() const;`
Total charge of hit.

`unsigned int flag() const;`
Returns status flag.

`unsigned int usedInFit() const;`
Returns flag providing info on if this hit was actually used in one of the various track fits.

`unsigned int trackReferenceCount() const;`
Returns the number of tracks associated with that hit.

`StDetectorId detector() const;`

`StThreeVectorF positionError() const;`

`StMatrixF covariantMatrix() const;`
Returns covariant matrix. In unknown (or not overwritten) simply fills the diagonal elements with the errors obtained via `positionError()`.

```
void setCharge(float);
void setFlag(unsigned char);
void setFitFlag(unsigned char);
void setTrackReferenceCount(unsigned char);
void setHardwarePosition(unsigned int);
void setPositionError(const StThreeVectorF&);
StPtrVecTrack relatedTracks(const StSPtrVecTrackNode&, StTrackType);
```

Public Member Operators

```
int operator==(const StHit&) const;
int operator!=(const StHit&) const;
```

5.34 StKinkVertex

Summary

Synopsis

```
#include "StKinkVertex.h"
class StKinkVertex;
```

Description

Related Classes

Public Constructors

```
StKinkVertex();
StKinkVertex(const dst_vertex_st&, const dst_tkf_vertex_st&);
```

Public Member Functions

```
StVertexId type() const;
unsigned int numberOfDaughters() const;
StTrack* daughter(unsigned int = 0);
const StTrack* daughter(unsigned int = 0) const;
StPtrVecTrack daughters(StTrackFilter&);
StParticleDefinition* pidParent() const;
StParticleDefinition* pidDaughter() const;
unsigned short geantIdParent() const;
unsigned short geantIdDaughter() const;
float dcaParentDaughter() const;
float dcaDaughterPrimaryVertex() const;
float dcaParentPrimaryVertex() const;
float hitDistanceParentDaughter() const;
float hitDistanceParentVertex() const;
float dE(unsigned int i) const;
float decayAngle() const;
float decayAngleCM() const;
const StThreeVectorF& parentMomentum() const;
StThreeVectorF& parentMomentum();
const StThreeVectorF& daughterMomentum() const;
StThreeVectorF& daughterMomentum();
void setGeantIdParent(unsigned short);
void setGeantIdDaughter(unsigned short);
void setDcaParentDaughter(float);
void setDcaDaughterPrimaryVertex(float);
void setDcaParentPrimaryVertex(float);
void setHitDistanceParentDaughter(float);
void setHitDistanceParentVertex(float);
void setdE(unsigned int, float);
void setDecayAngle(float);
void setDecayAngleCM(float);
void setParentMomentum(const StThreeVectorF&);
void setDaughterMomentum(const StThreeVectorF&);
void addDaughter(StTrack*);
void removeDaughter(StTrack*);
```

5.35 StL0Trigger

Summary

Synopsis

```
#include "StL0Trigger.h"
class StL0Trigger;
```

Description

Related Classes

Public Constructors

```
StL0Trigger();
StL0Trigger(const dst_L0_Trigger_st&);
```

Public Member Functions

```
unsigned int coarsePixelArraySize();
int coarsePixelArray(unsigned int i);
```

Index texttti runs from 0 to `coarsePixelArraySize()-1`.

```
int mwcCtbMultiplicity() const;
int mwcCtbDipole() const;
int mwcCtbTopology() const;
int mwcCtbMoment() const;
unsigned short dsmInput() const;
unsigned char detectorBusy() const;
unsigned short triggerToken() const;
unsigned short dsmAddress() const;
unsigned char addBits() const;
unsigned int lastDsmArraySize() const;
```

```
unsigned short lastDsmArray(unsigned int i);
```

Index texttti runs from 0 to `lastDsmArraySize()-1`.

```
unsigned int bcDataArraySize() const;
unsigned short bcDataArray(unsigned int i);
```

Index texttti runs from 0 to `bcDataArraySize()-1`.

```
unsigned int bunchCrossingId7bit(int run) const;
```

Takes run numbers as argument.

```
unsigned int bunchCrossingId() const;
```

```
void setMwcCtbMultiplicity(int);
void setMwcCtbDipole(int);
void setMwcCtbTopology(int);
void setMwcCtbMoment(int);
void setCoarsePixelArray(unsigned int i, int val);
void setDsmInput(unsigned short);
void setDetectorBusy(unsigned char);
void setTriggerToken(unsigned short);
void setDsmAddress(unsigned short);
void setAddBits(unsigned char);
void setLastDsmArray(unsigned int i, unsigned short val);
void setBcDataArray(unsigned int i, unsigned short val);
```

5.36 StL1Trigger

Summary

Synopsis

```
#include "StL1Trigger.h"
class StL1Trigger;
```

Description

Related Classes Inherits from StTrigger.

Public Constructors

```
StL1Trigger();
StL1Trigger(const dst_L0_Trigger_st&, const dst_L1_Trigger_st&);
```

Public Member Functions

```
unsigned int triggerWordPrime();

void setTriggerWordPrime(unsigned int);
```

5.37 StL3AlgorithmInfo

Summary L3 trigger algorithm information.

Synopsis

```
#include "StL3AlgorithmInfo.h"
class StL3AlgorithmInfo;
```

Description This class contains the complete information which describes a running L3 trigger algorithm and the algorithm results for this event.

Related Classes

Public Constructors

```
StL3AlgorithmInfo();
StL3AlgorithmInfo(Algorithm_Data*);
```

Public Member Functions

`int id() const;`
Unique algorithm id, defined in L3 include-files.

`bool on() const;`
Returns `true`, if algorithm was running for this event.

`bool accept() const;`
Returns `true`, if algorithm accepted this event.

`bool build() const;`
Returns `true`, if algorithm built this event.

`int numberOfProcessedEvents() const;`
Returns number of events this algorithm processed so far in this run. Returns `-1`, if this is undefined.

`int numberOfAcceptedEvents() const;`
Returns number of events this algorithm accepted so far in this run. Returns `-1`, if this is undefined.

`int numberOfBuildEvents() const;`
Returns number of events this algorithm built so far in this run. Returns `-1`, if this is undefined.

`int dataSize() const;`
Returns size of `data`-array filled by the algorithm.

`float data(int index) const;`
Returns `index`-th entry of `data`-array filled by the algorithm.

```
int preScale() const;
int postScale() const;
int intParameterSize() const;
int intParameter(int) const;
int floatParameterSize() const;
float floatParameter(int) const;
```

```
void setCounters(int, int, int);
void setParameters(int*, float*);
void setPreScale(int);
void setPostScale(int);
```

5.38 StL3EventSummary

Summary L3 event summary infomartion and access point to trigger algorithm information.

Synopsis

```
#include "StL3EventSummary.h"
class StL3EventSummary;
```

Description This class contains the L3 event summary information, e.g. global counters, and provides access to the trigger algorithm information. Member functions allow to check the event type (unbiased by L3, vertex triggered or triggered by an L3 algorithm) without the need to access the detailed trigger algorithm information.

Related Classes

Public Constructors

```
StL3EventSummary();
StL3EventSummary(Bank_L3_SUMD*);
```

Public Member Functions

```
int numberOfProcessedEvents() const;
```

Returns -1, if undefined for this event.

```
int numberOfReconstructedEvents() const;
```

Returns -1, if undefined for this event.

```
unsigned int numberOfTracks() const;
```

Total number of tracks found by L3. Note: This is not neccesarily the number tracks stored in StEvent.

```
unsigned int numberOfAlgorithms() const;
```

Number of L3 trigger algorithms switched on for this run.

```
bool unbiasedTrigger() const;
```

Returns `true`, if this event is unbiased by L3.

```
bool zVertexTrigger() const;
```

Returns `true`, if this event was accepted (and built) by the L3 vertex trigger.

```
StPtrVecL3AlgorithmInfo& algorithmsAcceptingEvent();
const StPtrVecL3AlgorithmInfo& algorithmsAcceptingEvent() const;
```

Returns vector of algorithms which accepted this event.

```
StSPtrVecL3AlgorithmInfo& algorithms();
const StSPtrVecL3AlgorithmInfo& algorithms() const;
```

Returns vector of all algorithms switched on for this run.

```
void addAlgorithm(StL3AlgorithmInfo*);
void setNumberOfTracks(int);
void setCounters(int, int);
```

5.39 StL3SoftwareMonitor

Summary

Synopsis

```
#include "StL3SoftwareMonitor.h"
class StL3SoftwareMonitor;
```

Description

Related Classes

Public Constructors

```
StL3SoftwareMonitor();
StL3SoftwareMonitor(const dst_mon_soft_l3_st&);
```

Public Data Member

`int id_algorithm;`
Id of the algorithm used in global L3.

`int id_hardware;`
Id of the hardware configuration.

`short triggermask;`
The result of the trigger inquiry.

`int nTotalHits ;`
Total number of clusters in the event.

`int nTotalTracks;`
Total number of tracks found by the tracker.

`int nTotalPrimaryTracks;`
Number of primary tracks found by the tracker.

`short processorId[24] ;`
Processor where the sector was reconstructed.

`float vertex[3][24];`
xyz coordinates of the vertex used for track finding.

`short id_param[24];`
The parameter set used in the tracker.

`int nHits[24];`
Number of clusters in the sector.

`int nTracks[24];`
Number of tracks found by the tracker.

`int nPrimaryTracks[24];`
Number of primary tracks found by the tracker.

`float cpuTime[24];`
CPU time used by the tracker.

5.40 StL3Trigger

Summary L3 header and entry point to the L3 part of StEvent tree.

Synopsis

```
#include "StL3Trigger.h"
class StL3Trigger;
```

Description This class is the entry point to the L3 tree of StEvent with exactly the same structure. In addition it provides access to the L3 event summary class.

Related Classes

Public Constructors

```
StL3Trigger();
```

Public Member Functions

```
StL3EventSummary* l3EventSummary();
const StL3EventSummary* l3EventSummary() const;

StTpcHitCollection* tpcHitCollection();
const StTpcHitCollection* tpcHitCollection() const;

StSPtrVecTrackDetectorInfo& trackDetectorInfo();
const StSPtrVecTrackDetectorInfo& trackDetectorInfo() const;

StSPtrVecTrackNode& trackNodes();
const StSPtrVecTrackNode& trackNodes() const;

unsigned int numberOfPrimaryVertices() const;
StPrimaryVertex* primaryVertex(unsigned int = 0);
const StPrimaryVertex* primaryVertex(unsigned int = 0) const;

void setL3EventSummary(StL3EventSummary*);
void setTpcHitCollection(StTpcHitCollection*);
void addPrimaryVertex(StPrimaryVertex*);
```

5.41 StMeasuredPoint

Summary

Synopsis

```
#include "StMeasuredPoint.h"
class StMeasuredPoint;
```

Description

Related Classes

Public Constructors

```
StMeasuredPoint();
StMeasuredPoint(const StThreeVectorF&);
```

Public Member Functions

```
const StThreeVectorF& position() const;
StThreeVectorF positionError() const = 0;
StMatrixF covariantMatrix() const = 0;
void setPosition(const StThreeVectorF&);
```

Public Member Operators

```
int operator==(const StMeasuredPoint&) const;
int operator!=(const StMeasuredPoint&) const;
```

5.42 StMwcTriggerDetector

Summary Interface to the MWC event data.

Synopsis

```
#include "StMwcTriggerDetector.h"
class StMwcTriggerDetector;
```

Description

Related Classes Inherits from `StObject`.

Public Constructors

```
StMwcTriggerDetector();
StMwcTriggerDetector(const dst_TrgDet_st&);
```

Public Member Functions

`unsigned int numberOfSectors() const;`
Number of MWC sectors (usually 24).

`unsigned int numberOfSubSectors() const;`
Number of MWC subsectors (usually 4).

`unsigned int numberOfPreSamples() const;`
Number of pre-samples available.

`unsigned int numberOfPostSamples() const;`
Number of post-samples available.

`unsigned int numberOfAuxWords() const;`
Number of auxiliary words (usually 32). No physical significance at present, these data are simple reserved for possible future use.

`float mips(unsigned int sec, unsigned int subsec, unsigned int evt = 0) c`
Return MIPS for sector `sec` and subsector `subsec`. The sectors run from 0 – `numberOfSectors()`-1, the subsectors from 0 – `numberOfSubSectors()`-1. The first half of the sector numbers are for the west side, the second half for the east side (usually 0-11 west, 12-23 east). Each sector has 4 subsectors, counting from the beam pipe outwards. The last argument `evt` has the following meaning: 0 is the triggered event, and `1-(1+npresamples+npostsamples-1)` are the other "events", in chronological order. For example, assuming 3 pre events and 2 post events:

evt = 0 triggered event at t = 0

evt = 1 1st pre event at t = -3

evt = 2 2nd pre event at t = -2

evt = 3 3rd pre event at t = -1

evt = 4 1st post event at t = +1

evt = 5 2nd post event at t = +2

`float aux(unsigned int i, unsigned int evt = 0) const;`
Argument `i` has no physical significance at present, these data are simple reserved (on request of Hank Crawford) for possible future use. For the second argument (`evt`) see `mips()` above.

```
void setMips(unsigned int, unsigned int, unsigned int, float);
void setAux(unsigned int, unsigned int, float);
void setNumberOfPreSamples(unsigned int);
void setNumberOfPostSamples(unsigned int);
```

5.43 StPhmdCluster

Summary Definitions of cluster objects used in PMD.

Synopsis
```
#include "StPhmdCluster.h"
class StPhmdCluster;
```

Description

Related Classes

Public Constructors
```
StPhmdCluster();
```

Public Member Functions

```
int module() const;
```
Returns the supermodule number where the cluster belong. It is assumed that clustering will be performed SM-wise.

```
int numberOfCells() const;
```
Returns the number of cells in the cluster.

```
float eta() const;
```
Returns eta value of the cluster.

```
float phi() const;
```
Returns phi value of the cluster.

```
float energy() const;
```
Returns the strength(cluster edep) of the cluster.

```
float sigma() const;
```
Returns spread(sigma) of the cluster.

```
int energyPid() const;
```
Returns cluster ID using MIP enegry cut.

```
int pid() const;
```
Returns cluster ID using CPV and PMD matching (or by other method).

```
int mcPid() const;
```
Returns Cluster ID from ManteCarlo track.

```
void setModule(int);
void setnumberOfCells(int);
void setEta(float);
void setPhi(float);
void setEnergy(float);
void setSigma(float);
void setEnergyPid(int);
void setPid(int);
void setMcPid(int);
```

Method for adding hits corresponding to that cluster. `void addHit(StPhmdHit*);`
```
void print(ostream& os = cout) const;
```

Method for obtaining hits corresponding to that cluster. `StPtrVecPhmdHit& hit();`
```
const StPtrVecPhmdHit& hit() const;
```

5.44 StPhmdClusterCollection

Summary Base class for PMD clusterCollecion.

Synopsis

```
#include "StPhmdClusterCollection.h"
#include "StPhmdHit.h"
class StPhmdClusterCollection;
```

Description The PMD cluster collection class `StPhmdClusterCollection` is the container for clusters in a subdetector.

Related Classes

Public Constructors

```
StPhmdClusterCollection();
```

Public Member Functions

```
void deleteClusters();
void deleteCluster(StPhmdCluster*);
void addCluster(StPhmdCluster*);
int numberOfClusters() const;
```

Returns the number of clusters.

```
StSPtrVecPhmdCluster& clusters();
const StSPtrVecPhmdCluster& clusters() const;
int clusterFinderId() const;
int clusterFinderParamVersion() const;
void setClusterFinderId(int);
void setClusterFinderParamVersion(int);
```

5.45 StPhmdCollection

Summary Base class for PMD which includes both the subdetectors CPV and PMD.

Synopsis

```
#include "StPhmdCollection.h"
#include "StPhmdDetector.h"
class StPhmdCollection;
```

Description The PMD Collection class `StPhmdCollection` keeps all information of both the subdetectors(CPV/PMD).

Related Classes

Public Constructors

```
StPhmdCollection();
```

Public Member Functions

```
StPhmdDetector* detector(StDetectorId);
```

Returns ID for subdetector.

```
const StPhmdDetector* detector(StDetectorId) const;
void setDetector(StPhmdDetector*);
```

5.46 StPhmdDetector

Summary Class for subdetector (CPV/PMD)

Synopsis

```
#include "StPhmdDetector.h"
#include "StPhmdHit.h"
#include "StPhmdModule.h"
#include "StPhmdClusterCollection.h"
class StPhmdDetector;
```

Description The PMD detector class `StPhmdDetector` contains the hit information inside `StPhmdModule` and the cluster information inside `StPhmdClusterCollection`.There are two instances of this class (CPV/PMD).

Related Classes

Public Constructors

```
StPhmdDetector();
StPhmdDetector(StDetectorId);
```

Public Member Functions

```
StDetectorId id() const;
```

Returns detector ID for CPV (25) and PMD (26).

```
unsigned int numberOfModules() const;
```

Returns number of modules in each subdetector.

```
bool addHit(StPhmdHit*);
unsigned int numberOfHits() const;
```

Return total number hits stored in each subdetector.

```
StPhmdModule* module(unsigned int);
int moduleHits(unsigned int);
```

Returns number of hits stored in each module.

```
const StPhmdModule* module(unsigned int) const;
StPhmdClusterCollection* cluster();
```

Returns number of clusters stored in each subdetector.

```
const StPhmdClusterCollection* cluster() const;
```

```
void setCluster(StPhmdClusterCollection*);
void setModule(StPhmdModule*, int);
```

5.47 StPhmdHit

Summary Definition of hit object used in PMD

Synopsis

```
#include "StPhmdHit.h"
class StPhmdHit;
```

Description The PMD hit class `StPhmdHit` is the basic class which contains the description of hit objects. The PMD consists of two subdetectors(CPV and PMD), each subdetectors has 12 supermodules of different sizes. Because of the PMD geometry the hits are stored in each supermodule in a given subdetector(CPV or PMD).

Related Classes

Public Constructors

```
StPhmdHit();
```

Public Member Functions

`int superModule() const;`
Returns supermodule number running from 1–12.

`int module();`
Returns Supermodule number.

`int subDetector()const;`
Returns subdetector number (CPV/PMD).

`int row()const;`
Returns row number in the supermodule.

`int column()const;`
Returns column number in the supermodule.

`float energy()const;`
Returns energy deposition in each cell.

`int adc()const;`
Returns ADC in each cell.

Given below the methods to access various data members.

```
void setSuperModule(int);
void setSubDetector(int);
void setRow(int);
void setColumn(int);
void setEnergy(int);
void setAdc(int);
```

5.48 StPhmdModule

Summary Class for each supermodule having the hit information

Synopsis

```
#include "StPhmdModule.h"
class StPhmdModule;
```

Description The PMD module class `StPhmdModule` takes hit information and stored in supermodulewise.

Related Classes

Public Constructors

```
StPhmdModule();
```

Public Member Functions

```
unsigned int numberOfHits() const;
```

Returns total number of hits in each supermodule.

```
StSPtrVecPhmdHit& hits();
const StSPtrVecPhmdHit& hits() const;
```

returns the container of hits.

5.49 StPrimaryTrack

Summary

Synopsis

```
#include "StPrimaryTrack.h"
class StPrimaryTrack;
```

Description

Related Classes

Public Constructors

```
StPrimaryTrack();
StPrimaryTrack(const dst_track_st&);
StPrimaryTrack(const StPrimaryTrack&);
StPrimaryTrack& operator=(const StPrimaryTrack&);
```

Public Member Functions

```
StTrackType type() const;
const StVertex* vertex() const;
```

Returns always `primary`.

```
void setVertex(StVertex*);
```

Returns pointer to actual primary (event) vertex.

5.50 StPrimaryVertex

Summary

Synopsis

```
#include "StPrimaryVertex.h"
class StPrimaryVertex;
```

Description

Related Classes Inherits directly from `StVertex`.

Public Constructors

```
StPrimaryVertex();
StPrimaryVertex(const dst_vertex_st&);
```

Public Member Functions

```
StVertexId type() const;
unsigned int numberOfDaughters() const;
```

```
int flag() const;
```

Inherited from StVertex. It his mentioned here since the return value in the context of the `StPrimaryVertex` has a special meaning.

For primary vertices `flag()` returns the fitting iteration and error reporting as follows (see also documentation for module `evr_am.F`):

+yx1 for normal, successfully found primary vertex during the 3rd iteration

-yx3 for initial seed value

-yx2 for first iteration value

-yx1 for second iteration value

-yx4 for failed fit with Determinant of G =0.0, occuring during any iteration.

-yx5 for failed error covariance matrix evaluation during fit with Determinant of E = 0.0, occuring during any iteration.

where

x is the event vertex id (for pileups). Zero means the triggered event vertex, x=1 is the next one etc.

y is the detector id for prevertex finding only. y=0 is not prevertex, y=1 TPC prevertex, y=2 SVT, y=3 FTPC etc. the detector id compliments this info.

```
StTrack* daughter(unsigned int);
const StTrack* daughter(unsigned int) const;
```

```
StPtrVecTrack daughters(StTrackFilter&);
```

```
StSPtrVecPrimaryTrack& daughters();
const StSPtrVecPrimaryTrack& daughters() const;
```

```
void addDaughter(StTrack*);
```

```
void removeDaughter(StTrack*);
```

```
void setParent(StTrack*); // overwrite inherited
```

5.51 StPsd

Summary Abstract base class for all Physics Summary Data classes.

Synopsis

```
#include "StPsd.h"
class StPsd;
```

Description All PSD classes must inherit from this base class.

Related Classes

Public Constructors

```
StPsd();
StPsd(StPwg, int);
```

Public Member Functions

```
StPwg pwg() const;
```

Returns the pwg to which this PSD belongs.

```
int id() const;
```

Returns the integer id of the PSD. One PWG can have several PSDs which can be distinguished by this ID.

```
void setPwg(StPwg);
void setId(int);
```

5.52 StRichCluster

Summary Description of a group of adjacent pixels from which hits are recontructed.

Synopsis

```
#include "StRichCluster.h"
class StRichCluster;
```

Description The cluster is an intermediate reconstructed object which allows the traceback from recontructed hits to single pixels. It is the level at which hit deconvolution algorithms work upon.

Related Classes

Public Constructors

```
StRichCluster();
```

Empty Constructor.

```
StRichCluster(int nPads, int nLocMax, int fPad,
       float ampSum, float amp2Sum, float rms2);
```

Constucted given the number of pads, number of local maximum within the cluster, first pad, sum of the amplitudes of the pads making up the cluster, the sum of the squared amplitude and the rms squared of the cluster.

Public Member Functions

```
bool operator==(const StRichCluster&) const;
```

Comparison of the pixels based on an exact match of all fields.

```
int numberOfPads() const;
```

Accesses the number of pads the cluster comprises.

```
int numberOfLocalMax() const;
```

Accesses the number of local maxima in the cluster.

```
int firstPad() const;
```

Accesses the first pad (in the pixel list).

```
float amplitudeSum() const;
```

Accesses the sum of the amplitudes of the pads identified with the cluster.

```
float amplitude2Sum() const;
```

Accesses the sum of the squares of the amplitudes of the pads identified with the cluster.

```
float rms2() const;
```

Accesses the square of the RMS of the cluster.

```
float rms2Calc();
```

Calculates the square rms of the cluster.

```
void increaseNumberOfLocalMax();
```

Modifies the cluster object by increasing the number of local maxima by 1.

```
void increaseNumberOfPads();
```

Modifies the cluster object by increasing the number of pads by 1.

```
void setFirstPad(int index);
```

Modifies the cluster object by setting the first pad of the cluster to index.

```
void setNumberOfPads(int newNPads);
```

Modifies the cluster object by setting the number of pads in the cluster to newNPads.

```
void updateAmplitude(float newamp);
```

Modifies the cluster object by setting the amplitude to newamp.

5.53 StRichCollection

Summary Contains all information of the Rich up until the point of hit reconstruction in generic STL like StEvent containers.

Synopsis
```
#include "StRichCollection.h"
class StRichCollection;
```

Description The raw and reconstructed pixel and hit information is contained in the StRichCollection. The module called StRchMaker is reponsible for creating a “RichCollection” and filling the pixels from the raw data and the clusters and hits from the cluster and hit finding algorithms.

Related Classes See StContainers.h.

Public Constructors

`StRichCollection();`
Empty constructor. All containers must be added by “set” member functions.

Public Member Functions

`const StSPtrVecRichPixel& getRichPixels() const;`
Returns the collection of pixels.

`StSPtrVecRichPixel& getRichPixels();`
Returns the collection of pixels.

`const StSPtrVecRichCluster& getRichClusters() const;`
Returns the collection of clusters.

`StSPtrVecRichCluster& getRichClusters();`
Returns the collection of clusters.

`const StSPtrVecRichHit& getRichHits() const;`
Returns the collection of reconstructed hits.

`StSPtrVecRichHit& getRichHits();`
Returns the collection of reconstructed hits.

`const StPtrVecTrack& getTracks() const;`
Returns the collection of `StTracks` that have an `StRichPidTrait` calculated in the reconstruction.

`StPtrVecTrack& getTracks();`
Returns the collection of `StTracks` that have an `StRichPidTrait` calculated in the reconstruction.

`void addPixel(const StRichPixel*);`
Adds a single pixel to the pixel collection.

`void addCluster(const StRichCluster*);`
Adds a single cluster to the cluster collection.

`void addHit(const StRichHit*);`
Adds a single hit to the hit collection.

`bool pixelsPresent() const;`
Returns the condition if the pixel container is present in the RichCollection.

`bool clustersPresent() const;`
Returns the condition if the cluster container is present in the RichCollection.

`bool hitsPresent() const;`
Returns the condition if the hit container is present in the RichCollection.

5.54 StRichHit

Summary Description of a reconstructed hit on the detector pad plane.

Synopsis
```
#include "StRichHit.h"
class StRichHit;
```

Description The hit object is the output from the cluster and hit finder. It specifies a recontructed position of a track (charged or photon) which passes through the detector pad plane.

Related Classes Inherits from `StMeasuredPoint` and `StHit`.

Public Constructors

`StRichHit();`
Empty Constructor.

`StRichHit(const StThreeVectorF& xg, const StThreeVectorF& dx);`
Specify a hit by the global position, xg, and the error on this position, dx.

```
StRichHit(const StThreeVectorF& xg, const StThreeVectorF& dx,
       unsigned int hp, float q, float maxAdc, unsigned char tc);
```
Specify a hit by the global position, xg, the error on this position, dx, the hardware position, hp, total integrated charge contained in the hit, q, the quantity of charge on the pixel with the maximum amplitude within the hit, maxAdc and the number of tracks associated with the hit, tc.

Public Member Functions

`const StThreeVectorF& local() const;`
Returns the position of the hit in the local coordinate system of the detector.

`const StThreeVectorF& internal() const;`
Returns the position of the hit in the interanl coordinate system of the detector.

`float maxAmplitude() const;`
Returns the amplitude of the pixel with the largest amplitude in the hit.

`int clusterNumber() const;`
Returns the number (index) of the cluster from which the hit was produced.

`unsigned short track() const;`
Returns the row of the track table from which the hit is associated.

`unsigned int reservedLong() const;`
Reserved for future use.

`float reservedFloat() const;`
Reserved for future use.

`StThreeVectorF& local();`
Returns the position of the hit in the local coordinate system of the detector.

`StThreeVectorF& internal();`
Returns the position of the hit in the internal coordinate system of the detector.

`unsigned short numberOfPads() const;`
Returns the number of pads from which a hit was constructed.

`void setNumberOfPads(unsigned short);`
Set the number of pads from which a hit was constructed.

`void setMaxAmplitude(float);`
Set the amplitude of the pixel with the maximum amplitude contained within the cluster.

```
void setClusterNumber(int);
```
Assign the cluster number (index) to which the it is associated.

```
void setTrack(unsigned int);
```
Set the track from which the hit is associated.

```
void setReservedLong(unsigned int);
```
Reserved for future use.

```
void setReservedFloat(float);
```
Reserved for future use.

5.55 StRichMCHit

Summary Description of a hit but adds Monte Carlo information to describe the origin of the hit.

Synopsis

```
#include "StRichMCHit.h"
class StRichMCHit;
```

Description This class inherits from `StRichHit` and contains the additional information of the origin of the hit in the `StRichMCInfo` class.

Related Classes See `StRichInfo`

Public Constructors

```
StRichMCHit();
```
Empty consturctor.

```
StRichMCHit(const StThreeVectorF& xg,
            const StThreeVectorF& dx);
```
Specify a hit by the global position, xg, and the error on this position, dx.

```
StRichMCHit(const StThreeVectorF& xg,
            const StThreeVectorF& dx,
            unsigned int hp, float q,
            float maxAdc, unsigned char tc);
```

```
StRichMCHit(const StThreeVectorF& xg,
            const StThreeVectorF& dx,
            unsigned int hp, float q,
            float maxAdc, unsigned char tc,
            StRichMCInfo& info);
```
Specify a hit by the global position, xg, the error on this position, dx, the hardware position, hp, total integrated charge contained in the hit, q, the quantity of charge on the pixel with the maximum amplitude within the hit, maxAdc, the number of tracks associated with the hit, tc, and the associated Monte Carlo information, info.

Public Member Functions

```
void setMCInfo(const StRichMCInfo&);
```
Returns the Monte Carlo information associated with the hit.

```
const StRichMCInfo& getMCInfo() const;
```
Returns the Monte Carlo information associated with the hit.

5.56 StRichMCInfo

Summary Contains the Monte Carle information associated with an object, be it a pixel, cluster, or hit.

Synopsis

```
#include "StRichMCInfo.h"
class StRichMCInfo;
```

Description This object contains Monte Carlo information so that the origin of an object, be it a pixel, cluster, or hit can be traced. This is necessary so that the feed-back photons may be accounted for and kept track.

Related Classes See `StRichMCHit` and `StRichMCPixel`.

Public Constructors

`StRichMCInfo();`
Empty constructor.

```
StRichMCInfo(int id, int gid, int trk,
             float q, int process);
```

Specifies the Monte Carlo information of an object by specifying its track, id, GEANT particle id, gid, parent track, trk, charge that it deposited, q, and process of origin, process, be it unknown (0), a charged particle (1), a photon (2), a feedback photon (4), or noise (8).

Public Member Functions

`int operator==(const StRichMCInfo&) const;`
Checks the equivalence based on the id, trk, and process field.

`int operator!=(const StRichMCInfo&) const;`
The complement of the equality operator.

`int id() const;`
Returns the index the track in the track table.

`int gid() const;`
Returns the GEANT id of the track.

`int trackp() const;`
Returns the index of the the parent track in the track table.

`float charge() const;`
Returns the charge deposited by the track in question.

`int process() const;`
Returns the type of process, be it unknown (0), a charged particle (1), a photon (2), a feedback photon (4), or noise (8).

5.57 StRichMCPixel

Summary The raw data information from the detector along with the Monte Carlo information of the tracks which produced this information.

Synopsis

```
#include "StRichMCPixel.h"
class StRichMCPixel;
```

Description This class contains, in addition to the raw pixel information, the description of the simulated (Monte Carlo) processes which contributed to its production.

Related Classes See `StRichPixel`, and `StRichMCInfo`.

Public Constructors

`StRichMCPixel();`
Empty constructor.

`StRichMCPixel(unsigned int packedData);`
Coded data of the pixel where the first 8 bits are the pad number, the second 8 bits are the row, and the next 11 bits are the ADC value (11th bit indicates overflow).

```
StRichMCPixel(unsigned int packedData,
              const StSPtrVecRichMCInfo&);
```
Coded data of the pixel where the first 8 bits are the pad number, the second 8 bits are the row, and the next 11 bits are the ADC value (11th bit indicates overflow) as well as a list of the MC information.

Public Member Functions

`int operator==(const StRichMCPixel&) const;`
Equality operator based on the packed data ONLY.

`int operator!=(const StRichMCPixel&) const;`
Complement of the equality operator.

`unsigned short contributions() const;`
Returns the number of tracks/processes contributing to the pixel.

`void addInfo(const StRichMCInfo*);`
Adds information to the Monte Carlo list.

`void setInfo(const StSPtrVecRichMCInfo&);`
Specifies the Monte Carlo list.

`const StSPtrVecRichMCInfo& getMCInfo() const;`
Returns the Monte Carlo information.

`StSPtrVecRichMCInfo& getMCInfo();`
Returns the Monte Carlo information.

5.58 StRichPid

Summary This class contains information specific to a mass hypothesis of the track in question. It contains the photons associated with the ring or band of the tracks as well as parameters calculated for the photons. It also contains derived parameters which are used in the algorithm of identifying the particles.

Synopsis

```
#include "StRichPid.h"
class StRichPid;
```

Description

Related Classes

Public Constructors

```
StRichPid();
```

Public Member Functions

```
StRichPid(StParticleDefinition* particle, StThreeVectorD resid,
float totAzim, float totArea,
unsigned short totHits, float trunAzim,
float trunArea, unsigned short trunHits);
```

Constructor requires the particle type (particle), species dependent residual from the extrapolated track (resid), the total Azimuthal angle (totAzim), total Area(totArea) of the ring bands that are found on the pad plane, as well as the total number of photons/hits (totHits) as well as the corresponding quantities in the trucated ring bands.

```
int operator==(const StRichPid&) const;
```

```
void setRingType(StParticleDefinition* particle);
```

Sets the ring type according to the mass hypothesis.

```
void setMipResidual(StThreeVectorD t);
```

Sets the residual between the associated hit on the RICH pad plane and the extrapolated TPC track. This residual is calculated with the particle species dependent momentum loss.

```
void setTotalAzimuth(float);
```

Sets the total azimuthal angle of the ring seen on the pad plane.

```
void setTotalArea(float);
```

Sets the total area of the ring/bands seen on the pad plane.

```
void setTotalHits(unsigned short);
```

Sets the total number of photons/hits contained in the ring on the pad plane.

```
void setTruncatedAzimuth(float);
```

Sets the truncated azimuthal angle of the ring seen on the pad plane.

```
void setTruncatedArea(float);
```

Sets the truncated area of the ring/bands seen on the pad plane.

```
void setTruncatedHits(unsigned short);
```

Sets the total number of photons/hits contained in the truncated ring area on the pad plane.

```
StParticleDefinition* getRingType() const;
```

Returns the ring type according to the mass hypothesis.

```
const StPtrVecRichHit& getAssociatedRichHits() const;
```

Returns the photons/hits associated with the mass hypothesis.

`StPtrVecRichHit& getAssociatedRichHits();`
Returns the photons/hits associated with the mass hypothesis.

`int getParticleNumber() const;`
Returns the particle type (pdgEncoding) of the mass hypothesis.

`StThreeVectorD getMipResidual() const;`
Returns the residual between the associated hit on the RICH pad plane and the extrapolated TPC track. This residual is calculated with the particle species dependent momentum loss.

`float getTotalAzimuth() const;`
Returns the total azimuthal angle of the ring seen on the pad plane.

`float getTotalArea() const;`
Returns the total area of the ring/bands seen on the pad plane.

`unsigned short getTotalHits() const;`
Returns the total number of photons/hits contained in the ring on the pad plane.

`float getTotalDensity() const;`
Returns the total area of the ring/bands seen on the pad plane.

`float getTruncatedAzimuth() const;`
Returns the truncated azimuthal angle of the ring seen on the pad plane.

`float getTruncatedArea() const;`
Returns the truncated area of the ring/bands seen on the pad plane.

`unsigned short getTruncatedHits() const;`
Returns the total number of photons/hits contained in the truncated ring area on the pad plane.

`float getTruncatedDensity() const;`
Returns the areal density of the truncated number of hits and area.

`bool isSet(StRichPidFlag);`
Checks the bit flags, as defined in `StEnumerations`

`void setBit(StRichPidFlag);`
Sets a bit flag, as defined in `StEnumerations`

`void unSetBit(StRichPidFlag);`
Unsets a bit flag, as defined in `StEnumerations`

5.59 StRichPidTraits

Summary This is the PID Trait of a track that passes through the RICH detector. It contains track specific information as well as a container of `StRichPid` which in turn contain information about the Cerenkov radiation associated with the rings associated with a different mass hypothesis (π/K/p).

Synopsis
```
#include "StRichPidTraits.h"
class StRichPidTraits;
```

Description

Related Classes

Public Constructors
```
StRichPidTraits();
```

Public Member Functions
```
int operator==(const StRichPidTraits&) const;
void addPid(StRichPid* );
```
adds an `StRichPid` to the container. This is data related to a "particle hypothesis". In the most general case there would be 3 such structures (π/K/p).

```
StSPtrVecRichPid getAllPids();
```
Returns the container of the `StRichPid`s.

```
const StSPtrVecRichPid getAllPids() const;
```
Returns the container of the `StRichPid`s.

```
StRichPid* getPid(StParticleDefinition* t);
```
Returns the `StRichPid` associated to a the particle type **t**.

```
const StRichPid* getPid(StParticleDefinition* t) const;
```
Returns the `StRichPid` associated to a the particle type **t**.

```
StRichSpectra* getRichSpectra();
const StRichSpectra* getRichSpectra() const;
```

```
void setProductionVersion(int);
```
Label for the time which the `StRichPidTrait` was produced. The default version (for OFFICIAL production) has a value of -999. This allows us to keep track of modifications if "`StEvent` .root" files are subsequently reprocessed; another production version is assigned.

```
void setId(int);
```
Set the probable id of the particle. This is not filled in the reconstruction.

```
void setProbability(float);
```
Set the probability of it being identified as an "id()".

```
void setAssociatedMip(StRichHit*);
```
Set the `StRichHit` which is identified with being the match from the TPC track extrapolation.

```
void setMipResidual(const StThreeVectorF&);
```
Set the residual or distance between the associated Mip and the match from the TPC track extrapolation.

```
void setRefitResidual(const StThreeVectorF&);
```
Set the residual or distance between the associated Mip and the match from the TPC track extrapolation AFTER a refit of the TPC track has been done.

`void setSignedDca2d(float);`
Set the signed 2-dimensional dca (in bend-plane) as calculated from the parent global of the track in question.

`void setSignedDca3d(float);`
Set the signed 3-dimensional dca as calculated from the parent global of the track in question.

`int   productionVersion() const;`
Returns the production version under which the `StRichPidTrait` was created. Default or official STAR production is always -999.

`int   id() const;`
Returns the probable id of the particle.

`float probability() const;`
Returns the probability that the particle is assigned an "id()".

`StRichHit* associatedMip() const;`
Returns the `StRichHit` which has been associated with the TPC track extrapolation.

`const StThreeVectorF& mipResidual() const;`
Returns the residual of associated MIP and the extrapolated TPC track.

`const StThreeVectorF& refitResidual() const;`
Returns the residual of associated MIP and the extrapolated TPC track AFTER a refit of the track parameters has been done.

`float signedDca2d() const;`
Returns the signed 2-dimensional (bend-plane) distance of closest approach (dca) between the global partner of the track and the primary vertex position.

`float signedDca3d() const;`
Returns the signed 3-dimensional distance of closest approach (dca) between the global partner of the track and the primary vertex position.

`void setRichSpectra(StRichSpectra*);`

5.60 StRichPixel

Summary The raw data information from the detector.

Synopsis

```
#include "StRichPixel.h"
class StRichPixel;
```

Description This class contains the raw pixel information, which includes the pad, row, and integerized ADC value of a single pixel.

Related Classes

Public Constructors

`StRichPixel();`
Empty constructor.

`StRichPixel(unsigned int packedData);`
Specify the pixel utilizing the packed data where the first 8 bits are the pad number, the second 8 bits are the row, and the next 11 bits are the ADC value (11th bit indicates overflow).

Public Member Functions

`int operator==(const StRichPixel&) const;`
Equality operator based on the coded data.

`int operator!=(const StRichPixel&) const;`
Complement of the equality operator.

`void setPackedData(unsigned int);`
Specify the coded data.

`unsigned short pad() const;`
Returns the pad number.

`unsigned short row() const;`
Returns the row number.

`unsigned short adc() const;`
Returns the adc value.

`unsigned int reservedLong() const;`
Reserved for future use.

`void setReservedLong(unsigned int);`
Reserved for future use.

5.61 StRichSoftwareMonitor

Summary

Synopsis

```
#include "StRichSoftwareMonitor.h"
class StRichSoftwareMonitor;
```

Description

Related Classes

Public Constructors

```
StRichSoftwareMonitor();
StRichSoftwareMonitor(const dst_mon_soft_rich_st&);
```

Public Member Functions

```
void setNumberOfPixels(int);
void setNumberOfClusters(int);
void setNumberOfHits(int);
void setTotalCharge(int);
int numberOfPixels() const;
int numberOfClusters() const;
int numberOfHits() const;
int totalCharge() const;
void setNumberOfTracksCrossing(int);
void setNumberOfTracksPidCalculated(int);
void setNumberOfTracksAbove1Gev(int);
void setNumberOfHitsInRings(int);
void setNumberOfRings(int);
int numberOfTracksCrossing() const;
int numberOfTracksPidCalculated() const;
int numberOfTracksAbove1Gev() const;
int numberOfHitsInRings() const;
int mult_rich_tot;
```

5.62 StRichSpectra

Summary

Synopsis

```
#include "StRichSpectra.h"
class StRichSpectra;
```

Description

Related Classes

Public Constructors

```
StRichSpectra(int v=-999);
```

Public Member Functions

```
StRichSpectra(float,float,float,float,float,float,
float,float,int,float,int,int,
float,float,float,float,int,float,
float, float, float,
int, int, int,
int=-999);
void setExtrapolatedPosition(float, float);
void setExtrapolatedResidual(float, float);
void setCorrectedExtrapolatedResidual(float, float);
void setCherenkovAngle(float);
void setCherenkovSigma(float);
void setCherenkovPhotons(int);
void setPeakAngle(float);
void setPeakPhotons(int);
void setTotalPhotons(int);
void setMassSquared(float);
void setLineIntegralRatio(float);
void setLineIntegral(float);
void setAlpha(float);
void setFlag(int);
void setReserved(float);
void setMeanD(float pi=FLT_MAX, float k=FLT_MAX, float p=FLT_MAX);
void setNumberOfD(int pi=-100, int k=-100, int p=-100);
void setVersion(int);
float getExtrapolatedX() const;
float getExtrapolatedY() const;
float getExtrapolatedXResidual() const;
float getExtrapolatedYResidual() const;
float getCorrectedExtrapolatedXResidual() const;
float getCorrectedExtrapolatedYResidual() const;
float getCherenkovAngle() const;
float getCherenkovSigma() const;
int getCherenkovPhotons() const;
float getPeakAngle() const;
int getPeakPhotons() const;
int getTotalPhotons() const;
float getMassSquared() const;
float getLineIntegralRatio() const;
float getLineIntegral() const;
```

```
float getAlpha() const;
int getFlag() const;
float getMeanDpi() const;
float getMeanDk() const;
float getMeanDp() const;
int getMeanDnpi() const;
int getMeanDnk() const;
int getMeanDnp() const;
float getReserved() const;
int getVersion() const;
```

5.63 StRunInfo

Summary Parameters related to current run.

Synopsis

```
#include "StRunInfo.h"
class StRunInfo;
```

Description This class contains parameters related to the current run. Much of the information is redundant since some parameters can be obtained from the offline DB and other classes. Hint: If you are not sure about the beam direction you can also use `blue` and `yellow` for the blue and yellow ring respectively.

Public Constructors

```
StRunInfo();
```

Public Member Functions

```
int runId() const;
```

Run number.

```
time_t productionTime() const;
```

Time when production of this run/file started. Since the production is done on a file-by-file basis and a run can be split in many files this time stamp is related to the beginning of the production of the file. The format is UNIX time.

```
TString productionVersion() const;
```

Library version of the production chain.

```
double centerOfMassEnergy() const;
int beamMassNumber(StBeamDirection) const;
float beamEnergy(StBeamDirection) const;
float initialBeamIntensity(StBeamDirection) const;
```

Only at beginning of run.

```
float beamLifeTime(StBeamDirection) const;
float beamFillNumber(StBeamDirection) const;
```

```
double magneticField() const;
```

Z-component of magnetic field at $(x, y, z) = (0, 0, 0)$.

```
double tpcDriftVelocity(StBeamDirection) const;
```

TPC drift velocity as obtained from the DB.

```
double zdcWestRate() const;
```

ZDC West scaler rate (counts per second) interpolated by a straight line between periodic online database entries.

```
double zdcEastRate() const;
```

ZDC East scaler rate (counts per second) interpolated by a straight line between periodic online database entries.

```
double zdcCoincidenceRate() const;
```

ZDC Coincidence scaler rate (counts per second) interpolated by a straight line between periodic online database entries.

```
double backgroundRate() const;
```

Mult scaler rate (counts per second) interpolated by a straight line between periodic online database entries.

```
double l0RateToRich() const;
```

L0 rate sent to the RICH from trigger rate (counts per second) interpolated by a straight line between periodic online database entries.

```
void setRunId(int);
void setProductionTime(time_t);
void setProductionVersion(const char*);
void setCenterOfMassEnergy(double);
void setBeamMassNumber(StBeamDirection, int);
void setBeamEnergy(StBeamDirection, float);
void setInitialBeamIntensity(StBeamDirection, float);
void setBeamLifeTime(StBeamDirection, float);
void setBeamFillNumber(StBeamDirection, float);
void setMagneticField(double);
void setTpcDriftVelocity(StBeamDirection, double);
void setZdcWestRate(double);
void setZdcEastRate(double);
void setZdcCoincidenceRate(double);
void setBackgroundRate(double);
void setL0RateToRich(double);
```

5.64 StSoftwareMonitor

Summary

Synopsis

```
#include "StSoftwareMonitor.h"
class StSoftwareMonitor;
```

Description

Related Classes

Public Constructors

```
StSoftwareMonitor();
StSoftwareMonitor(const dst_mon_soft_tpc_st*,
                  const dst_mon_soft_svt_st*,
                  const dst_mon_soft_ftpc_st*,
                  const dst_mon_soft_emc_st*,
                  const dst_mon_soft_ctb_st*,
                  const dst_mon_soft_rich_st*,
                  const dst_mon_soft_glob_st*,
                  const dst_mon_soft_l3_st*);
StSoftwareMonitor& operator=(const StSoftwareMonitor&);
StSoftwareMonitor(const StSoftwareMonitor&);
```

Public Member Functions

```
StTpcSoftwareMonitor* tpc();
const StTpcSoftwareMonitor* tpc() const;
StSvtSoftwareMonitor* svt();
const StSvtSoftwareMonitor* svt() const;
StFtpcSoftwareMonitor* ftpc();
const StFtpcSoftwareMonitor* ftpc() const;
StEmcSoftwareMonitor* emc();
const StEmcSoftwareMonitor* emc() const;
StRichSoftwareMonitor* rich();
const StRichSoftwareMonitor* rich() const;
StCtbSoftwareMonitor* ctb();
const StCtbSoftwareMonitor* ctb() const;
StGlobalSoftwareMonitor* global();
const StGlobalSoftwareMonitor* global() const;
StL3SoftwareMonitor* l3();
const StL3SoftwareMonitor* l3() const;
StTofSoftwareMonitor* tof();
const StTofSoftwareMonitor* tof() const;

void setTpcSoftwareMonitor(StTpcSoftwareMonitor*);
void setSvtSoftwareMonitor(StSvtSoftwareMonitor*);
void setFtpcSoftwareMonitor(StFtpcSoftwareMonitor*);
void setEmcSoftwareMonitor(StEmcSoftwareMonitor*);
void setRichSoftwareMonitor(StRichSoftwareMonitor*);
void setCtbSoftwareMonitor(StCtbSoftwareMonitor*);
void setGlobalSoftwareMonitor(StGlobalSoftwareMonitor*);
void setL3SoftwareMonitor(StL3SoftwareMonitor*);
void setTofSoftwareMonitor(StTofSoftwareMonitor*);
```

5.65 StSsdHit

Summary

Synopsis

```
#include "StSsdHit.h"
class StSsdHit;
```

Description

Related Classes

Public Constructors

```
StSsdHit();
StSsdHit(const StThreeVectorF&,
const StThreeVectorF&,
unsigned int, float, unsigned char = 0);
StSsdHit(const dst_point_st&);
```

Public Member Functions

```
unsigned int ladder() const;
```

Returns ladder number in the range 1–20.

```
unsigned int wafer() const;
```

Returns wafer number in the range 1–16.

```
unsigned int centralStripNSide() const;
unsigned int centralStripPSide() const;
unsigned int clusterSizeNSide() const;
unsigned int clusterSizePSide() const;
```

5.66 StSsdHitCollection

Summary

Synopsis

```
#include "StSsdHitCollection.h"
class StSsdHitCollection;
```

Description

Related Classes

Public Constructors

```
StSsdHitCollection();
```

Public Member Functions

```
bool addHit(StSsdHit*);
unsigned int numberOfHits() const;
unsigned int numberOfLadders() const;
StSsdLadderHitCollection* ladder(unsigned int);
const StSsdLadderHitCollection* ladder(unsigned int) const;
```

5.67 StSsdLadderHitCollection

Summary

Synopsis

```
#include "StSsdLadderHitCollection.h"
class StSsdLadderHitCollection;
```

Description

Related Classes

Public Constructors

```
StSsdLadderHitCollection();
```

Public Member Functions

```
unsigned int numberOfHits() const;
unsigned int numberOfWafers() const;
StSsdWaferHitCollection* wafer(unsigned int);
const StSsdWaferHitCollection* wafer(unsigned int) const;
```

5.68 StSsdWaferHitCollection

Summary

Synopsis

```
#include "StSsdWaferHitCollection.h"
class StSsdWaferHitCollection;
```

Description

Related Classes

Public Constructors

```
StSsdWaferHitCollection();
```

Public Member Functions

```
StSPtrVecSsdHit& hits();
const StSPtrVecSsdHit& hits() const;
```

5.69 StSvtBarrelHitCollection

Summary

Synopsis

```
#include "StSvtBarrelHitCollection.h"
class StSvtBarrelHitCollection;
```

Description

Related Classes

Public Constructors

```
StSvtBarrelHitCollection();
```

Public Member Functions

```
unsigned int numberOfHits() const;
unsigned int numberOfLadders() const;
StSvtLadderHitCollection* ladder(unsigned int);
const StSvtLadderHitCollection* ladder(unsigned int) const;
```

5.70 StSvtHit

Summary

Synopsis

```
#include "StSvtHit.h"
class StSvtHit;
```

Description

Related Classes

Public Constructors

```
StSvtHit();
StSvtHit(const StThreeVectorF&,
         const StThreeVectorF&,
         unsigned int, float, unsigned char = 0);
StSvtHit(const dst_point_st&);
```

Public Member Functions

`unsigned int layer() const;`
Layer in which hit is located. Layer number runs from 1–6.

`unsigned int ladder() const;`
Ladder number runs from 1–16. Ladders are counted per barrel.

`unsigned int wafer() const;`
Wafer number runs from 1–7.

`unsigned int barrel() const;`
Barrel number runs from 1–3.

`unsigned int hybrid() const;`

5.71 StSvtHitCollection

Summary

Synopsis

```
#include "StSvtHitCollection.h"
class StSvtHitCollection;
```

Description

Related Classes

Public Constructors

```
StSvtHitCollection();
```

Public Member Functions

```
bool addHit(StSvtHit*);
```

```
unsigned int numberOfHits() const;
```

```
unsigned int numberOfBarrels() const;
```

```
StSvtBarrelHitCollection* barrel(unsigned int);
const StSvtBarrelHitCollection* barrel(unsigned int) const;
```

5.72 StSvtLadderHitCollection

Summary

Synopsis

```
#include "StSvtLadderHitCollection.h"
class StSvtLadderHitCollection;
```

Description

Related Classes

Public Constructors

```
StSvtLadderHitCollection();
```

Public Member Functions

```
unsigned int numberOfHits() const;
unsigned int numberOfWafers() const;
StSvtWaferHitCollection* wafer(unsigned int);
const StSvtWaferHitCollection* wafer(unsigned int) const;
```

5.73 StSvtSoftwareMonitor

Summary

Synopsis

```
#include "StSvtSoftwareMonitor.h"
class StSvtSoftwareMonitor;
```

Description

Some of the data member in this class are arrays of size 4. The first 3 elements refer to the 3 SVT barrels, the 4^{th} element refers to the SSD.

Related Classes

Public Constructors

```
StSvtSoftwareMonitor();
StSvtSoftwareMonitor(const dst_mon_soft_svt_st&);
```

Public Data Member

`int n_clus_svt[4];`
Total number clusters in each SVT/SSD barrel/layer.

`int n_pts_svt[4];`
Total number of space points in each SVT/SSD barrel/layer.

`int n_trk_svt;`
Total number of tracks in SVT.

`float chrg_svt_tot[4];`
Total charge deposition in each SVT/SSD barrel/layer.

`float hit_frac_svt[4];`
Fraction of hits used in each SVT/SSD barrel/layer.

`float avg_trkL_svt;`
Average track length (cm) SVT
or average number of points assigned.

`float res_pad_svt;`
Average residual, pad direction, SVT
or average chisq(1) of fit.

`float res_drf_svt;`
Average residuals, drift direction, SVT
or average chisq(2) of fit.

5.74 StSvtWaferHitCollection

Summary

Synopsis

```
#include "StSvtWaferHitCollection.h"
class StSvtWaferHitCollection;
```

Description

Related Classes

Public Constructors

```
StSvtWaferHitCollection();
```

Public Member Functions

```
StSPtrVecSvtHit& hits();
const StSPtrVecSvtHit& hits() const;
```

5.75 StTofCell

Summary

Synopsis

```
#include "StTofCell.h"
class StTofCell;
```

Description

Related Classes

Public Constructors

```
StTofCell();
StTofCell(int, int, int, int, int, StTrack*);
```

Public Member Functions

```
int operator==(const StTofCell&) const;
int operator!=(const StTofCell&) const;
int trayIndex() const;
int moduleIndex() const;
int cellIndex() const;
int adc() const;
int tdc() const;
StTrack* associatedTrack();
const StTrack* associatedTrack() const;
void setTrayIndex(int);
void setModuleIndex(int);
void setCellIndex(int);
void setAdc(int);
void setTdc(int);
void setAssociatedTrack(StTrack*);
```

5.76 StTofCollection

Summary

Synopsis

```
#include "StTofCollection.h"
class StTofCollection;
```

Description

Related Classes

Public Constructors

```
StTofCollection();
```

Public Member Functions

```
const StSPtrVecTofCell& tofCells() const;
StSPtrVecTofCell& tofCells();
const StSPtrVecTofSlat& tofSlats() const;
StSPtrVecTofSlat& tofSlats();
const StSPtrVecTofHit& tofHits() const;
StSPtrVecTofHit& tofHits();
const StSPtrVecTofData& tofData() const;
StSPtrVecTofData& tofData();
void addSlat(const StTofSlat*);
void addCell(const StTofCell*);
void addHit(const StTofHit*);
void addData(const StTofData*);
bool cellsPresent() const;
bool slatsPresent() const;
bool hitsPresent() const;
bool dataPresent() const;
```

5.77 StTofData

Summary

Synopsis

```
#include "StTofData.h"
class StTofData;
```

Description

Related Classes

Public Constructors

```
StTofData();
StTofData(unsigned short, unsigned short, unsigned short, short, unsi
```

Public Member Functions

```
int operator==(const StTofData&) const;
int operator!=(const StTofData&) const;
unsigned short dataIndex() const;
unsigned short adc() const;
unsigned short tdc() const;
short tc() const;
unsigned short sc() const;
void setDataIndex(unsigned short);
void setAdc(unsigned short);
void setTdc(unsigned short);
void setTc(short);
void setSc(unsigned short);
```

5.78 StTofHit

Summary

Synopsis

```
#include "StTofHit.h"
class StTofHit;
```

Description

Related Classes

Public Constructors

```
StTofHit();
```

Public Member Functions

```
int trayIndex() const;
int moduleIndex() const;
int cellIndex() const;
int daqIndex() const;
float timeOfFlight() const;
float pathLength() const;
float beta() const;
StTrack* associatedTrack();
const StTrack* associatedTrack() const;
float tofExpectedAsElectron() const;
float tofExpectedAsPion() const;
float tofExpectedAsKaon() const;
float tofExpectedAsProton() const;
float sigmaElectron() const;
float sigmaPion() const;
float sigmaKaon() const;
float sigmaProton() const;
StParticleDefinition* particleHypothesis();
const StParticleDefinition* particleHypothesis() const;
void setTrayIndex(int);
void setModuleIndex(int);
void setCellIndex(int);
void setDaqIndex(int);
void setTimeOfFlight(float);
void setPathLength(float);
void setBeta(float);
void setAssociatedTrack(StTrack*);
void settofExpectedAsElectron(float);
void settofExpectedAsPion(float);
void settofExpectedAsKaon(float);
void settofExpectedAsProton(float);
void setsigmaElectron(float);
void setsigmaPion(float);
void setsigmaKaon(float);
void setsigmaProton(float);
void setparticleHypothesis(StParticleDefinition*);
```

5.79 StTofMCCell

Summary

Synopsis

```
#include "StTofMCCell.h"
class StTofMCCell;
```

Description

Related Classes

Public Constructors

```
StTofMCCell();
StTofMCCell(const StTofMCInfo&);
```

Public Member Functions

```
int operator==(const StTofMCCell&) const;
int operator!=(const StTofMCCell&) const;
const StTofMCInfo& mcInfo() const;
void setMCInfo(const StTofMCInfo&);
void setNHits(int nHits);
void setNPhe(int nPhe);
void setDe(float de);
void setDs(float ds);
void setTof(float tof);
```

5.80 StTofMCHit

Summary

Synopsis

```
#include "StTofMCHit.h"
class StTofMCHit;
```

Description

Related Classes

Public Constructors

```
StTofMCHit();
```

Public Member Functions

```
int trkId() const;
int gId() const;
void setTrkId(Int_t);
void setGId(Int_t);
```

5.81 StTofMCInfo

Summary

Synopsis

```
#include "StTofMCInfo.h"
class StTofMCInfo;
```

Description

Related Classes

Public Constructors

```
StTofMCInfo();
StTofMCInfo(int, int, int, float, int, float,
float, float, float, float, float, float,
float);
int operator==(const StTofMCInfo& MCInfo) const;
int operator!=(const StTofMCInfo& MCInfo) const;
Int_t mTrkId;
Int_t mGId;
Int_t mNHits;
Int_t mNPhe;
Float_t mDe;
Float_t mPTot;
Float_t mDs;
Float_t mSLength;
Float_t mPmLength;
Float_t mTof;
Float_t mTime;
Float_t mMTime;
Float_t mMTimeL;
```

5.82 StTofMCSlat

Summary

Synopsis

```
#include "StTofMCSlat.h"
class StTofMCSlat;
```

Description

Related Classes

Public Constructors

```
StTofMCSlat();
StTofMCSlat(const StTofMCInfo&);
```

Public Member Functions

```
int operator==(const StTofMCSlat&) const;
int operator!=(const StTofMCSlat&) const;
const StTofMCInfo& mcInfo() const;
void setMCInfo(const StTofMCInfo&);
void setNHits(int nHits);
void setNPhe(int nPhe);
void setDe(float de);
void setDs(float ds);
void setTof(float tof);
```

5.83 StTofPidTraits

Summary

Synopsis

```
#include "StTofPidTraits.h"
class StTofPidTraits;
```

Description

Related Classes

Public Constructors

```
StTofPidTraits();
```

Public Member Functions

5.84 StTofSlat

Summary

Synopsis

```
#include "StTofSlat.h"
class StTofSlat;
```

Description

Related Classes

Public Constructors

```
StTofSlat();
StTofSlat(unsigned short, unsigned short, unsigned short, StTrack*);
```

Public Member Functions

```
int operator==(const StTofSlat&) const;
int operator!=(const StTofSlat&) const;
unsigned short slatIndex() const;
unsigned short adc() const;
unsigned short tdc() const;
StTrack* associatedTrack();
const StTrack* associatedTrack() const;
void setSlatIndex(unsigned short);
void setAdc(unsigned short);
void setTdc(unsigned short);
void setAssociatedTrack(StTrack*);
```

5.85 StTofSoftwareMonitor

Summary

Synopsis

```
#include "StTofSoftwareMonitor.h"
class StTofSoftwareMonitor;
```

Description

Related Classes

Public Constructors

```
StTofSoftwareMonitor();
```

Public Member Functions

5.86 StTpcDedxPidAlgorithm

Summary Functor which allows to obtain PID information derived from the dE/dx of the track in the TPC.

Synopsis

```
#include "StTpcDedxPidAlgorithm.h"
class StTpcDedxPidAlgorithm;
```

Description This is an example of an `StPidAlgorithm`. Please handle it as such. The functor selects the first `StTrackPidTraits` it encounters that *(i)* is derived from the TPC and *(ii)* uses the method passed as an argument to the constructor. The various methods added to this class were implemented by Craig Ogilvie (MIT). For a more general overview see 3.4.6 and 3.4.7.

Related Classes `StTpcDedxPidAlgorithm` inherits directly from `StPidAlgorithm`.

Public Constructors

```
StTpcDedxPidAlgorithm(StDedxMethod = kTruncatedMeanId);
```

Creates instances of `StTpcDedxPidAlgorithm`. The argument is the dE/dx method of your choice. The default is `kTruncatedMeanId`, i.e. the `kTruncatedMeanId` method is selected in case you do not provide an argument. See 2.2 for possible methods. Note that the existence of an enumeration doesn't mean that it is available or even implemented.

Public Member Functions

```
StParticleDefinition* operator() (const StTrack&, const StSPtrVecTrackPid
const StDedxPidTraits* traits() const;
double numberOfSigma(const StParticleDefinition*) const;
double meanPidFunction(const StParticleDefinition*) const;
double sigmaPidFunction(const StParticleDefinition*) const;
```

5.87 StTpcHit

Summary

Synopsis

```
#include "StTpcHit.h"
class StTpcHit;
```

Description

Related Classes

Public Constructors

```
StTpcHit();
```

Default constructor.

```
StTpcHit(const StThreeVectorF&,
         const StThreeVectorF&,
         unsigned int, float, unsigned char = 0);
```

```
StTpcHit(const dst_point_st&);
```

Public Member Functions

```
unsigned int sector() const;
```

Returns sector number in the range 1–24.

```
unsigned int padrow() const;
```

Returns padrow number in the range 1–45.

```
unsigned int padsInHit() const;
```

```
unsigned int pixelsInHit() const;
```

5.88 StTpcHitCollection

Summary

Synopsis

```
#include "StTpcHitCollection.h"
class StTpcHitCollection;
```

Description

Related Classes

Public Constructors

```
StTpcHitCollection();
```

Public Member Functions

```
bool addHit(StTpcHit*);
```

```
unsigned int numberOfHits() const;
```

Total number of TPC hits in the collection.

```
unsigned int numberOfSectors() const;
```

```
StTpcSectorHitCollection* sector(unsigned int i);
const StTpcSectorHitCollection* sector(unsigned int i) const;
```

Index `i` runs from 0–(n-1) where n = `numberOfSectors()`.

5.89 StTpcPadrowHitCollection

Summary Holds all hits stored in a padrow.

Synopsis

```
#include "StTpcPadrowHitCollection.h"
class StTpcPadrowHitCollection;
```

Description

Related Classes

Public Constructors

```
StTpcPadrowHitCollection();
```

Public Member Functions

```
StSPtrVecTpcHit& hits();
const StSPtrVecTpcHit& hits() const;
```

5.90 StTpcPixel

Summary

Synopsis

```
#include "StTpcPixel.h"
class StTpcPixel;
```

Description

Related Classes

Public Constructors

```
StTpcPixel();
StTpcPixel(unsigned short, unsigned int);
StTpcPixel(const dst_pixel_st&);
```

Public Member Functions

```
unsigned short detector() const;
```

```
unsigned short sector() const;
```

Returns sector in the range 1–24.

```
unsigned short padrow() const;
```

Returns padrow in the range 1–45.

```
unsigned int pad() const;
unsigned int timebin() const;
unsigned int adc() const;
```

Public Member Operator

```
int operator==(const StTpcPixel&) const;
int operator!=(const StTpcPixel&) const;
```

5.91 StTpcSectorHitCollection

Summary

Synopsis

```
#include "StTpcSectorHitCollection.h"
class StTpcSectorHitCollection;
```

Description

Related Classes

Public Constructors

```
StTpcSectorHitCollection();
```

Public Member Functions

```
unsigned int numberOfHits() const;
```

Total number of hits in the sector.

```
unsigned int numberOfPadrows() const;
```

Always 45.

```
StTpcPadrowHitCollection* padrow(unsigned int i);
const StTpcPadrowHitCollection* padrow(unsigned int) const;
```

Returns padrow hit collection. Note that `i`=0–44.

5.92 StTpcSoftwareMonitor

Summary

Synopsis

```
#include "StTpcSoftwareMonitor.h"
class StTpcSoftwareMonitor;
```

Description

Related Classes

Public Constructors

```
StTpcSoftwareMonitor();
StTpcSoftwareMonitor(const dst_mon_soft_tpc_st&);
```

Public Data Member

`int n_clus_tpc_tot;`
Total number of clusters in TPC.

`int n_clus_tpc_in[24];`
Total number of clusters in inner TPC sectors.

`int n_clus_tpc_out[24];`
Total number of clusters in outer TPC sectors.

`int n_pts_tpc_tot;`
Total number of space points in TPC.

`int n_pts_tpc_in[24];`
Total number of space points in inner TPC sectors.

`int n_pts_tpc_out[24];`
Total number of space points in outer TPC sectors.

`int n_trk_tpc[2];`
Total number of tracks in TPC, tan(dip angle) < 0 (≥ 0).

`float chrg_tpc_drift[10];`
Charge deposited in TPC in along z.

`float chrg_tpc_tot;`
Total charge deposition in TPC.

`float chrg_tpc_in[24];`
Total charge deposition in inner TPC sectors.

`float chrg_tpc_out[24];`
Total charge deposition in outer TPC sectors.

`float hit_frac_tpc[2];`
Fraction of hits used in TPC, tan(dip angle) < 0 (≥ 0).

`float avg_trkL_tpc[2];`
Average track length (cm)
or average number of assigned, tan(dip angle) < 0 (≥ 0).

`float res_drf_tpc[2];`
Average residuals, drift direction,
or average chisq(2) of fit, tan(dip angle) < 0 (≥ 0).

5.93 StTptTrack

Summary

Synopsis

TPC tracks as generated by the “tpt” tracker. `#include "StTptTrack.h"`
`class StTptTrack;`

Description

`StTptTrack` is created using the original tracks generated in the `tpt` tracker. This is a pure TPC track. Use for debugging purposes only.

Related Classes

`StTptTrack` is derived from `StTrack`. See there for more info.

Public Constructors

```
StTptTrack();
StTptTrack(const dst_track_st&);
StTptTrack(const StTptTrack&);
StTptTrack& operator=(const StTptTrack&);
```

Public Member Functions

`StTptTrack type() const;`
Returns always `tpt`.

`const StVertex* vertex() const;`
Returns always `null`.

5.94 StTrack

Summary

Synopsis

```
#include "StTrack.h"
class StTrack;
```

Description

Related Classes

Public Constructors

```
StTrack();
StTrack(const dst_track_st&);
StTrack(const StTrack&);
StTrack& operator=(const StTrack&);
```

Public Member Functions

`StTrackType type() const = 0;`
Track type as defined in enumeration `StTrackType`. See section 2.2.

`const StVertex* vertex() const = 0;`
Pointer to parent vertex. Pure virtual function. To be implemented in concrete classes inheriting from `StTrack`.

`unsigned short key() const;`
Returns foreign key as defined in the dst_track table. The key can be used as track ID. All tracks in the same track node should carry the same ID.

`short flag() const;`
Track quality control flag. The most important point is: `flag()` > 0 is good, `flag()` < 0 is bad. The meaning of the negative return code depends on the actual implementation of the algorithms and will therefore vary in time. The best sources for more details are currently (June 2000):

- http://www.star.bnl.gov/STAR/html/all_l/html/dst_track_flags.html
- http://www.star.bnl.gov/STARAFS/comp/reco/kalerr.html

`unsigned short encodedMethod() const;`

`bool finderScheme(StTrackFinderScheme) const;`

`StTrackFittingMethod fittingMethod() const;`

`float impactParameter() const;`
Returns the impact parameter, or better the distance-of-closest aproach, of the track to the primary vertex.

`float length() const;`

`unsigned short numberOfPossiblePoints() const;`
Returns number of the (theoretically) maximum number of points along a track in TPC, SVT, and SSD. Does not include Tof, RICH, or EMC.

`unsigned short numberOfPossiblePoints(StDetectorId det) const;`
Returns number of the (theoretically) maximum number of points along a track in a given detector `det`. Applies only for tracking detectors: TPC, SVT, and SSD.

`const StTrackTopologyMap& topologyMap() const;`

`StTrackGeometry* geometry();`
`const StTrackGeometry* geometry() const;`
Returns pointer to track geometry, i.e. the track parameters as curvature, dip angles, momentum and more. Note that these are the parameters at the origin of the track. This can either be the first point, or better the point on the helix which is closest to the first point (global tracks), or the primary vertex (primary tracks). See also `outerGeometry()`.

`StTrackGeometry* outerGeometry();`
`const StTrackGeometry* outerGeometry() const;`
Same as above but for the last (outer-most) point on the track. For high-$p_\perp$ tracks the parameters from `geometry()` and `outerGeometry()` will be almost identical in terms of momentum and pointing but at lower $p_\perp$ they might differ considerably. Use the ones from the outer most point if you use the parameters to extrapolate to detectors at large radii (RICH, EMC, ToF, etc.). This was added after the Kalman fitter was implemented which incorporates effects like energy-loss and multiple scattering.

```
StTrackDetectorInfo* detectorInfo();
const StTrackDetectorInfo* detectorInfo() const;
const StTrackFitTraits& fitTraits() const;
const StSPtrVecTrackPidTraits& pidTraits() const;
StSPtrVecTrackPidTraits& pidTraits();
StPtrVecTrackPidTraits pidTraits(StDetectorId) const;
const StParticleDefinition* pidTraits(StPidAlgorithm&) const;
StTrackNode* node();
const StTrackNode* node() const;
void setFlag(short);
void setEncodedMethod(unsigned short);
void setImpactParameter(float);
void setLength(float);
void setTopologyMap(const StTrackTopologyMap&);
void setGeometry(StTrackGeometry*);
void setOuterGeometry(StTrackGeometry*);
void setFitTraits(const StTrackFitTraits&);
void addPidTraits(StTrackPidTraits*);
void setDetectorInfo(StTrackDetectorInfo*);
void setNode(StTrackNode*);
```

5.95 StTrackDetectorInfo

Summary Information on hits/points associated with a track.

Synopsis
```
#include "StTrackDetectorInfo.h"
class StTrackDetectorInfo;
```

Description This class holds all information related to hits used to reconstruct a given track. The main component is a list of pointers to those hits which are associated with the track. Several tracks can in principle share the same instance of `StTrackDetectorInfo`. Note, that in case there are no hits stored on a DST, this class won't provide a lot of info. The only methods still valid are:

- `firstPoint()`
- `lastPoint()`
- `numberOfPoints()`
- `numberOfPoints(StDetectorId)`

In this case the topology map (see 5.100) is the only source left to get more detailed info on hits.

Related Classes

Public Constructors
```
StTrackDetectorInfo();
StTrackDetectorInfo(const dst_track_st&);
```

Public Member Functions

`const StThreeVectorF& firstPoint() const;`
First point on the track.

`const StThreeVectorF& lastPoint() const;`
Last point on the track.

`unsigned short numberOfPoints() const;`
Returns the number of points assigned to the the track during reconstruction (*all* detectors). This number might or might not reflect the number of hits actually referenced by this class, i.e. it is not affected by calls to `addHit()` or `removeHit()`. Even if hits are not loaded, and thus no references to hits exist, this method will return the proper values. If you want to know the number of hits actually referenced use `numberOfReferencedPoints()` (see below). This method takes only the following hits into account: TPC, SVT, and SSD. Becomes obsolete when the DST tables are gone. The preferred way of getting this info is via the `StTrackTopologyMap` (see 5.100) available from class `StTrack` (see 5.94).

`unsigned short numberOfPoints(StDetectorId det) const;`
Returns the number of points in a given detector `det`. This is the number of hits from detector `det` assigned to this track at reconstruction time. This number might or might not reflect the number of hits actually referenced by this class, i.e. it is not affected by calls to `addHit()` or `removeHit()`. Even if hits are not loaded, and thus no references to hits exist, this method will return the proper values. If you want to know the number of hits actually referenced use `numberOfReferencedPoints()` (see below).
Because of the packing scheme of the variable used to store this information there's no way to distinguish between FTPC and TPC hits. That means that

```
numberOfPoints(kTpcId) == numberOfPoints(kFtpcWestId)
                       == numberOfPoints(kFtpcEastId)
```

If you are not sure you better check using the `StTrackTopologyMap` (see 5.100) available from class `StTrack` (see 5.94). This method does not apply for detectors other than: TPC, SVT, and SSD. Becomes obsolete when the DST tables are gone.

`unsigned short numberOfReferencedPoints() const;`
Returns the total number of points (hits) referenced. This is **not** necessarily the number of points assigned to the track during reconstruction. If the refrring hits are not loaded (e.g. on DST) this method will return 0. To find out about the number of hits used from the reconstruction chain use `numberOfPoints()`. This method will count all sort of hits including those from EMC, RICH, and ToF.

`unsigned short numberOfReferencedPoints(StDetectorId det) const;`
Returns the number of points (hits) referenced originating from detector `det`. This does **not** necessarily reflect the number of points assigned during reconstruction. If no hits are loaded for a given detector (e.g. on DST) this function will return 0. To find out about the number of hits as defined in reconstruction chain use `numberOfPoints()`. This method works for all sorts of hits including those from EMC, RICH, and ToF.

```
StPtrVecHit hits(StDetectorId) const;
StPtrVecHit hits(StHitFilter&) const;
StPtrVecHit& hits();
const StPtrVecHit& hits() const;
void setFirstPoint(const StThreeVectorF&);
void setLastPoint(const StThreeVectorF&);
void setNumberOfPoints(unsigned short);
void addHit(StHit*);
void removeHit(StHit*&);
```

5.96 StTrackFitTraits

Summary

Synopsis

```
#include "StTrackFitTraits.h"
class StTrackFitTraits;
```

Description

Related Classes

Public Constructors

```
StTrackFitTraits();
StTrackFitTraits(unsigned short, unsigned short, float[2], float[15]);
StTrackFitTraits(const dst_track_st&);
```

Public Member Functions

`unsigned short numberOfFitPoints() const;`
Total number of points used for the fit, i.e. summing the hits in all detectors the track crossed.

`unsigned short numberOfFitPoints(StDetectorId id) const;`
Number of points in detector `id` used for the fit.

`StParticleDefinition* pidHypothesis() const;`
PID hypothesis used for the fit. This variable is only useful if the fitting algorithms takes energy loss and multiple scattering into account.

`StMatrixF covariantMatrix() const;`

`double chi2(unsigned int i = 0) const;`
Depending on the fitting method and the type of track there is either one or two values. If a real 3D fit is performed (e.g. Kalman) the first value (`i = 0`) is the χ^2 per degree of freedom and the second value (`i = 1`) is the upper tail probability of χ^2 distribution for this value. If two 2-dimensional fits are performed (circle in xy plane and straight line in path-z) the χ^2 per degree of freedom for each fit is stored.

5.97 StTrackGeometry

Summary Abstract base class. All descriptions of track geometry inherit from this class.

Synopsis

```
#include "StTrackGeometry.h"
class StTrackGeometry;
```

Description The concrete implementations of this class (e.g. `StHelixModel`) are used to describe the geometry of a track, i.e., it's direction, shape, origin, and momentum.

Related Classes Class `StHelixModel` inherits from `StTrackGeometry`.

Public Constructors

```
StTrackGeometry();
StTrackGeometry(const dst_track_st&);
```

Public Member Functions

```
StTrackModel model() const = 0;
short charge() const = 0;
double curvature() const = 0;
double psi() const = 0;
double dipAngle() const = 0;
const StThreeVectorF& origin() const = 0;
const StThreeVectorF& momentum() const = 0;
StPhysicalHelixD helix() const = 0;
```

5.98 StTrackNode

Summary

Synopsis

```
#include "StTrackNode.h"
class StTrackNode;
```

Description

Related Classes

Public Constructors

```
StTrackNode();
```

Public Member Functions

```
void addTrack(StTrack*);
void removeTrack(StTrack*);
unsigned int entries() const;
StTrack* track(unsigned int);
const StTrack* track(unsigned int) const;
unsigned int entries(StTrackType) const;
StTrack* track(StTrackType, unsigned int = 0);
const StTrack* track(StTrackType, unsigned int = 0) const;
```

5.99 StTrackPidTraits

Summary Abstract interface to all track related PID information.

Synopsis

```
#include "StTrackPidTraits.h"
class StTrackPidTraits;
```

Description This class is the abstract interface to all track related PID information. Since the implementations of the various concrete PID classes differ strongly this class is incomplete and thus a `dynamic_cast` is required each time when concrete implementations are accessed through this interface (e.g. in `StTrack`).

Related Classes All PID classes (e.g. `StDedxPidTraits`) inherit from this class.

Public Constructors

```
StTrackPidTraits();
StTrackPidTraits(StDetectorId);
StTrackPidTraits(const dst_dedx_st&);
```

Public Member Functions

```
short detector() const;
```

Returns the detector from which the PID is derived.

5.100 StTrackTopologyMap

Summary Provides details on the topology of hits along a track trajectory.

Synopsis

```
#include "StTrackTopologyMap.h"
class StTrackTopologyMap;
```

Description Represents packed 64-bits used to indicate either the continuity or gaps in tracks as they cross different active elements of the SVT, SSD, TPC and FTPC. Also indicates if track extrapolates to other detectors (MWC, CTB, TOF, RICH, EMC barrel, EMC end cap) and if a vertex constraint was used or not. In case no hits are available on the DST this is the only piece of information available on the topology of the hits belonging to a track.
The class provides methods that allow to access information without detailed knowledge on how the bits are packed.

Related Classes

Public Constructors

```
StTrackTopologyMap();
StTrackTopologyMap(unsigned int, unsigned int);
StTrackTopologyMap(const unsigned int*);
```

Public Member Functions

`bool primaryVertexUsed() const;`
Indicates wether the primary vertex was used or not. Also used to indicate that a secondary vertex constraint was used for special track fitting methods for decay vertices (see method variable).

`unsigned int numberOfHits(StDetectorId id) const;`
Returns the number of hits assigned to the track in detector `id`.

`bool hasHitInRow(StDetectorId det, unsigned int row) const;`
Row numbering starts at 1 following STAR conventions.

`bool hasHitInSvtLayer(unsigned int layer) const;`
Layer numbering starts at 1 following STAR conventions.

`bool turnAroundFlag() const;`
Indicates if a track spirals in which case the information stored is incomplete.

`unsigned int data(unsigned int i) const;`
Returns the "raw" data in case you want to figure out yourself what bit is set (in case you know what it stands for). The map needs 2 long words (64 bits) hence one has to provide an argument to request the first or the second (`i`=0,1). The class actually stores the data in the form of unsigned ints so that they may be stored in a TTree (TBranch does not support ULong_t), but the interface effectively hides this from the user.

`int largestGap(StDetectorId id) const;`
Returns the largest gap in units of rows/layers in detector id. Please note, that the method does **not** return a path length but an integer number representing the maximum number of contiguous rows/layers without a hit. This is most useful for the TPC and FTPC although the method will also work for the SVT. In case there are less than 2 hits in total or in case something else is wrong the method returns `-1` to indicate failure.
Example:
Assume a track has hits in row 24, 25, 26, 30, 35, 36, 38 the method will return 4

since the largest gap is formed by the missing rows 31, 32, 33, and 34. All other gaps are smaller.

Global Operators

```
ostream& operator<< (ostream& os,
                     const StTrackTopologyMap& m);
```

Prints the topology map `m` to output stream `os`, i.e. the whole bit pattern starting at the least significant bit (0) up to bit 63.

Example:

```
cout << track->topologyMap() << endl;
```

prints

```
0000000000000000001011111111111111111111111111000011000000000000
```

5.101 StTrigger

Summary

Synopsis

```
#include "StTrigger.h"
class StTrigger;
```

Description

Related Classes

Public Constructors

```
StTrigger();
StTrigger(unsigned short aw, unsigned short w);
```

Public Member Functions

```
unsigned short triggerActionWord() const;
unsigned short triggerWord() const;
void setTriggerActionWord(unsigned short);
void setTriggerWord(unsigned short);
```

Public Member Operator

```
int operator==(const StTrigger&) const;
int operator!=(const StTrigger&) const;
```

5.102 StTriggerData

Summary Abstract base class to serve as interface for all trigger data starting Run III (2003).

Synopsis
```
#include "StTriggerData.h"
class StTriggerData;
```

Description

Related Classes StTriggerData2003

Public Constructors
```
StTriggerData();
```

Public Member Functions

```
virtual void dump() const = 0;
```
Dump data as text on screen.

```
virtual int year() const;
```
Year of the data.

```
virtual unsigned int version() const = 0;
```
TrgDataType Version Number.

```
virtual unsigned int numberOfPreXing() const = 0;
```
Number of pre-xing data for detectors.

```
virtual unsigned int numberOfPostXing() const = 0;
```
Number of pre-xing data for detectors.

```
virtual unsigned int token() const = 0;
virtual unsigned int triggerWord() const = 0;
virtual unsigned int actionWord() const = 0;
virtual unsigned int busyStatus() const;
virtual unsigned int bunchId48Bit() const;
virtual unsigned int bunchId7Bit() const;
virtual unsigned int spinBit() const;
virtual unsigned short tcuBits() const;
virtual unsigned short lastDSM(int channel) const;
virtual unsigned short vertexDSM(int channel) const;
virtual unsigned short ctbLayer1DSM(int channel) const;
virtual unsigned short ctbLayer2DSM(int channel) const;
virtual unsigned short emcLayer2DSM(int channel) const;
virtual unsigned short fpdLayer2DSM(int channel) const;
virtual unsigned short ctb(int pmt, int prepost=0) const;

virtual unsigned short
mwc(int sector, int prepost=0) const;

virtual unsigned short
zdcUnAttenuated(int eastwest, int prepost=0) const;

virtual unsigned short
zdcAttenuated(int eastwest, int prepost=0) const;
```

```
virtual unsigned short
zdcADC(int eastwest, int pmt, int prepost=0) const;

virtual unsigned short
zdcTDC(int eastwest, int prepost=0) const;

virtual unsigned short
bemcHighTower(int eta, int phi, int prepost=0) const;

virtual unsigned short
bemcJetPatch (int eta, int phi, int prepost=0) const;

virtual unsigned short
eemcHighTower(int eta, int phi, int prepost=0) const;

virtual unsigned short
eemcJetPatch (int eta, int phi, int prepost=0) const;

virtual unsigned short
bbcADC(int eastwest, int pmt, int prepost=0) const;

virtual unsigned short
bbcTDC(int eastwest, int pmt, int prepost=0) const;

virtual unsigned short
bbcADCSum(int eastwest, int prepost=0) const;

virtual unsigned short
bbcADCSumLargeTile(int eastwest, int prepost=0) const;

virtual unsigned short
bbcEarliestTDC(int eastwest, int prepost=0) const;

virtual unsigned short bbcTimeDifference() const;

virtual unsigned short
fpd(int eastwest, int module, int pmt, int prepost=0) const;

virtual unsigned short
fpdSum(int eastwest, int module) const;

virtual char* getTriggerStructure() = 0;
```

5.103 StTriggerData2003

Summary
Concrete class (inheriting from `StTriggerData`) that contains the trigger data information for Run III in 2003.

Synopsis
```
#include "StTriggerData2003.h"
class StTriggerData2003;
```

Description
Users should not use this class but rather go through the abstract interface class `StTriggerData`. Not all methods are listed here. Those inherited but not overwritten are only listed in the base class (see 5.102).

Related Classes
Inherits from `StTriggerData`.

Public Constructors
```
StTriggerData2003();
StTriggerData2003(char*);
StTriggerData2003(TrgDataType2003*);
void dump() const;
unsigned int version() const;
unsigned int numberOfPreXing() const;
unsigned int numberOfPostXing() const;
unsigned int token() const;
unsigned int triggerWord() const;
unsigned int actionWord() const;
unsigned int bunchId48Bit() const;
unsigned int bunchId7Bit() const;
unsigned int spinBit() const;
unsigned short ctb(int pmt, int prepost=0) const;
unsigned short mwc(int pmt, int prepost=0) const;

unsigned short
bbcADC(int eastwest, int pmt, int prepost=0) const;

unsigned short
bbcTDC(int eastwest, int pmt, int prepost=0) const;

unsigned short
bbcADCSum(int eastwest, int prepost=0) const;

unsigned short
bbcADCSumLargeTile(int eastwest, int prepost=0) const;

unsigned short
bbcEarliestTDC(int eastwest, int prepost=0) const;

unsigned short
bbcTimeDifference() const;

unsigned short
fpd(int eastwest, int module, int pmt, int prepost=0) const;
```

```
unsigned short
fpdSum(int eastwest, int module) const;
```

```
char* getTriggerStructure();
TrgDataType2003* getTriggerStructure2003();
```
Experts only.

5.104 StTriggerDetectorCollection

Summary

Synopsis

```
#include "StTriggerDetectorCollection.h"
class StTriggerDetectorCollection;
```

Description

Related Classes

Public Constructors

```
StTriggerDetectorCollection();
StTriggerDetectorCollection(const dst_TrgDet_st&);
```

Public Member Functions

```
StBbcTriggerDetector& bbc();
const StBbcTriggerDetector& bbc() const;
StCtbTriggerDetector& ctb();
const StCtbTriggerDetector& ctb() const;
StMwcTriggerDetector& mwc();
const StMwcTriggerDetector& mwc() const;
StVpdTriggerDetector& vpd();
const StVpdTriggerDetector& vpd() const;
StZdcTriggerDetector& zdc();
const StZdcTriggerDetector& zdc() const;
StEmcTriggerDetector& emc();
const StEmcTriggerDetector& emc() const;
```

5.105 StTriggerId

Summary

Synopsis

```
#include "StTriggerId.h"
class StTriggerId;
```

Description

Related Classes

Public Constructors

```
StTriggerId();
```

Public Member Functions

```
unsigned int mask() const;
bool isTrigger(unsigned int id) const;
unsigned int version(unsigned int id) const;
unsigned int nameVersion(unsigned int id) const;
unsigned int thresholdVersion(unsigned int id) const;
unsigned int prescaleVersion(unsigned int id) const;
vector<unsigned int> triggerIds() const;
void setMask(unsigned int);
void addTrigger(unsigned int, unsigned int,
unsigned int, unsigned int, unsigned int);
```

5.106 StTriggerIdCollection

Summary Collection that holds all relevant trigger information starting RunIII (2003).

Synopsis

```
#include "StTriggerIdCollection.h"
class StTriggerIdCollection;
```

Description

Related Classes

Public Constructors

```
StTriggerIdCollection();
```

Public Member Functions

```
const StTriggerId* nominal() const;
const StTriggerId* l1() const;
const StTriggerId* l2() const;
const StTriggerId* l3() const;

void setL1(StTriggerId*);
void setL2(StTriggerId*);
void setL3(StTriggerId*);
void setNominal(StTriggerId*);
```

5.107 StV0Vertex

Summary

Synopsis

```
#include "StV0Vertex.h"
class StV0Vertex;
```

Description

Related Classes

Public Constructors

```
StV0Vertex();
StV0Vertex(const dst_vertex_st&, const dst_v0_vertex_st&);
```

Public Member Functions

```
StVertexId type() const;

unsigned int numberOfDaughters() const;

StTrack* daughter(StChargeSign sign);

const StTrack* daughter(StChargeSign sign) const;

StTrack* daughter(unsigned int);

const StTrack* daughter(unsigned int) const;

StPtrVecTrack daughters(StTrackFilter&);

void addDaughter(StTrack*);

void removeDaughter(StTrack*);

float dcaDaughterToPrimaryVertex(StChargeSign sign) const;

float dcaDaughters() const;

float dcaParentToPrimaryVertex() const;

const StThreeVectorF& momentumOfDaughter(StChargeSign sign) const;

StThreeVectorF momentum() const;

void setDcaDaughterToPrimaryVertex(StChargeSign, float);

void setMomentumOfDaughter(StChargeSign, const StThreeVectorF&);
```

```
void setDcaDaughters(float);

void setDcaParentToPrimaryVertex(float);
```

5.108 StVertex

Summary

Synopsis

```
#include "StVertex.h"
class StVertex;
```

Description

Related Classes

Public Constructors

```
StVertex();
StVertex(const dst_vertex_st&);
```

Public Member Functions

```
StVertexId type() const = 0;
```

```
int flag() const;
```

For primary vertex this indicates the fitting iteration and error reporting as follows.

+yx1 for normal, successfully found primary vertex during the 3rd iteration

-yx3 for initial seed value

-yx2 for first iteration value

-yx1 for second iteration value

-yx4 for failed fit with Determinant of G =0.0, occuring during any iteration.

-yx5 for failed error covariance matrix evaluation during fit with determinant of E = 0.0, occuring during any iteration.

Definitions:

x is the event vertex id (for pileups). Zero means the triggered event vertex, x=1 is the next one etc

y is the detector ID for prevertex finding only y=0 is not prevertex, y=1 TPC prevertex, y=2 SVT, y=3 FTPC etc.

```
float chiSquared() const;
```

Returns χ^2 per degree of freedom.

```
float probChiSquared() const;
```

Upper tail probability of χ^2 distribution.

```
StMatrixF covariantMatrix() const;
```

```
StThreeVectorF positionError() const;
```

```
StTrack* parent();
const StTrack* parent() const;
```

Return pointer to parent track.

```
unsigned int numberOfDaughters() const = 0;
```

```
StTrack* daughter(unsigned int i) = 0;
const StTrack* daughter(unsigned int i) const = 0;
```

```
StPtrVecTrack daughters(StTrackFilter&) = 0;

void setFlag(unsigned int);
void setCovariantMatrix(float[6]);
void setChiSquared(float);
void setProbChiSquared(float);
void setParent(StTrack*);
void addDaughter(StTrack*) = 0;
void removeDaughter(StTrack*) = 0;
```

Public Member Operator

```
int operator==(const StVertex&) const;
int operator!=(const StVertex&) const;
```

5.109 StVpdTriggerDetector

Summary

Synopsis

```
#include "StVpdTriggerDetector.h"
class StVpdTriggerDetector;
```

Description

Related Classes

Public Constructors

```
StVpdTriggerDetector();
StVpdTriggerDetector(const dst_TrgDet_st&);
```

Public Member Functions

```
unsigned int numberOfVpdCounters() const;
float adc(unsigned int) const;
float time(unsigned int) const;
float minimumTime(StBeamDirection) const;
float vertexZ() const;
void setAdc(unsigned int, float);
void setTime(unsigned int, float);
void setMinimumTime(StBeamDirection, float);
void setVertexZ(float);
```

5.110 StXiVertex

Summary

Synopsis

```
#include "StXiVertex.h"
class StXiVertex;
```

Description

Related Classes

Public Constructors

```
StXiVertex();
StXiVertex(const dst_vertex_st&, const dst_xi_vertex_st&);
```

Public Member Functions

```
StVertexId type() const;
unsigned int numberOfDaughters() const;
StTrack* daughter(unsigned int = 0);
const StTrack* daughter(unsigned int = 0) const;
StPtrVecTrack daughters(StTrackFilter&);
float dcaBachelorToPrimaryVertex() const;
float dcaV0ToPrimaryVertex() const;
float dcaDaughters() const;
float dcaParentToPrimaryVertex() const;
const StThreeVectorF& momentumOfBachelor() const;
StThreeVectorF momentumOfV0() const;
StThreeVectorF momentum() const;
StV0Vertex* v0Vertex() const;
StTrack* bachelor();
double chargeOfBachelor();
void setDcaBachelorToPrimaryVertex(float);
void setMomentumOfBachelor(const StThreeVectorF&);
void setDcaDaughters(float);
void setDcaParentToPrimaryVertex(float);
void setV0Vertex(StV0Vertex*);
void addDaughter(StTrack*);
void removeDaughter(StTrack*);
```

5.111 StZdcTriggerDetector

Summary Zero Degree Calorimeter (ZDC) data

Synopsis

```
#include "StZdcTriggerDetector.h"
class StZdcTriggerDetector;
```

Description

Related Classes

Public Constructors

```
StZdcTriggerDetector();
StZdcTriggerDetector(const dst_TrgDet_st&);
```

Public Member Functions

```
unsigned int numberOfZdcWords() const;
```

Currently, and probably for the rest of its lifetime, this method will return 16.

```
float adc(unsigned int) const;
```

The interpretation of the different ADC values can vary as a function of time. The meaning of the various words (status Sep 2001) is as follows:

- unattenuated signals (mostly useful for looking at the single neutron peak)

  adc[7] = ZDC East Module 1

  adc[6] = ZDC East Module 2

  adc[5] = ZDC East Module 3

  adc[4] = ZDC East Sum

  adc[3] = ZDC West Module 1

  adc[2] = ZDC West Module 2

  adc[1] = ZDC West Module 3

  adc[0] = ZDC West Sum

- attenuated signals

  adc[10] = ZDC West Attenuated Sum

  adc[11] = ZDC West Module 2 Attenuated

  adc[12] = ZDC West Module 1 Attenuated

  adc[13] = ZDC East Attenuated Sum

  adc[14] = ZDC East Module 2 Attenuated (until Aug 29, 2001)

  adc[14] = East+West Attenuated Analog Sum (since Aug 29, 2001)

  adc[15] = ZDC East Module 1 Attenuated

- other

  adc[8] = TDC East (since run 2001)

  adc[9] = TDC West (since run 2001)

The reason why there are only 2 single modules in the attenuated part is that we don't have enough ADC channels. However, the sum is however still the sum of all 3 channels per module. The most relevant are the sum channels.

```
float tdc(unsigned int) const;
```

```
float adcSum(StBeamDirection) const;
```

Sum for ZDC east and west.

```
float adcSum() const;
```
Total ZDC sum.

```
float vertexZ() const;
```
Returns z position of vertex obtained from ZDC timing information.

```
void setAdc(unsigned int, float);
void setTdc(unsigned int, float);
void setAdcSum(StBeamDirection, float);
void setAdcSum(float);
void setVertexZ(float);
```

A Brief Introduction to UML

A.1 Introduction

UML stands for Unified Modelling Language. It is the current standard modelling language used to design object oriented software. It is a unification of the concepts and notations used in earlier models such as Booch and OMT.

Although the complexity and theoretical concept behind UML is certainly not of great use for most of the developer and user of HENP software it provides one important component which is gaining more and more importance: its notation, i.e., a set of rules on how to present complex software in form of simple graphic symbols. There is a notation for static elements of a design such as classes, attributes, and relationships and a notation for modelling the dynamic elements such as objects, messages, and, state machines. In this appendix we present only the basic aspects of the static modelling notation – the class diagrams.

A.2 Class diagrams

The purpose of a class diagram is to depict the classes within a model. In an object oriented application, classes have attributes (member variables), operations (member functions) and relationships with other classes.The fundamental element of the class diagram is an icon that represents a class. This icon is shown

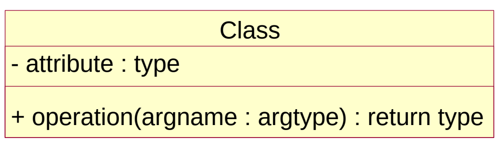

Figure A.1: The class icon in UML.

in Fig. A.1. A class icon is simply a rectangle divided into three compartments. The topmost compartment contains the name of the class. The middle compartment contains a list of attributes (member variables), and the bottom compartment contains a list of operations (member functions). In many diagrams, the bottom two compartments are omitted. Even when they are present, they typically do not show every attribute and operations. The goal is to show only those attributes and operations that are useful for the particular diagram. There is typically never a need to show every attribute and operation of a class on any diagram. Fig. A.2 shows a typical UML description of a class that represents a Hit (here fictitious Hit2D). Notice

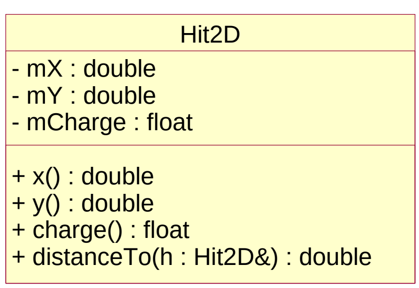

Figure A.2: Hit2D class. Attributes and operations are shown.

that each member variable is followed by a colon and by the type of the variable. If the type is redundant, or otherwise unnecessary, it can be omitted. Notice also that the return values follow the member functions in a similar fashion. Again, these can be omitted. Finally, notice that the member function arguments also have a name and type. Again one can omit the name or the arguments altogether.

At the beginning of each attribute and operations the visibility of the class is indicated through a simple tag. UML provides three tags:

+ public

protected

– private

These abbreviations match exactly the three levels of visibility provided in C++. The class shown in Fig. A.2 is then translated into C++ code as follows:

```
class Hit2D {
public:
    double x();
    double y();
    double distanceTo(Hit2D& h);
private:
    double mX, mY;
    float  mCharge;
};
```

A.3 Composition Relationships

Each instance of type Hit usually contains an instance of type Position. One also says the Hit *has* a Position. This is a relationship known as composition. It can be depicted in UML using a class relationship. Fig. A.3 shows the *composition* relationship. The black diamond represents composition. It is placed on the Hit

Figure A.3: Class Hit has a Position.

class because it is the Hit that is composed of (or has) a Position. The arrowhead on the other end of the relationship denotes that the relationship is navigable in only one direction. That is, Position does not know about Hit. In UML relationships are presumed to be bidirectional unless the arrowhead is present to restrict them. Composition relationships are a strong form of containment or aggregation. Aggregation is a whole/part relationship. In this case, Hit is the whole, and Position is part of Hit. However, composition is more than just aggregation. Composition also indicates that the lifetime of Position is dependent upon Hit. This means that if Hit is destroyed, Position will be destroyed with it. In C++ we would represent this as:

```
class Hit {
     Position mPos;
};
```

In this case we have represented the composition relationship as a member variable. We could also have used a pointer so long as the destructor of Hit deleted the pointer. A more realistic example can be found in StEvent. There the *StHit* class has a member of type *StThreeVector* which represents a position.

A.4 Inheritance

The inheritance relationship in UML is depicted by a triangular arrowhead which points to the base class. One or more lines proceed from the base of the arrowhead connecting it to the derived classes.

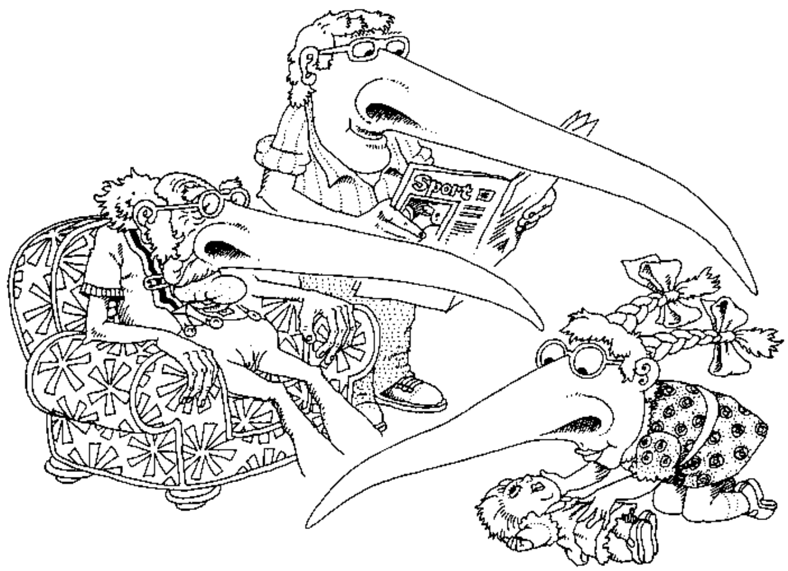

Figure A.4: A subclass may inherit the structure and behaviour of its superclass.

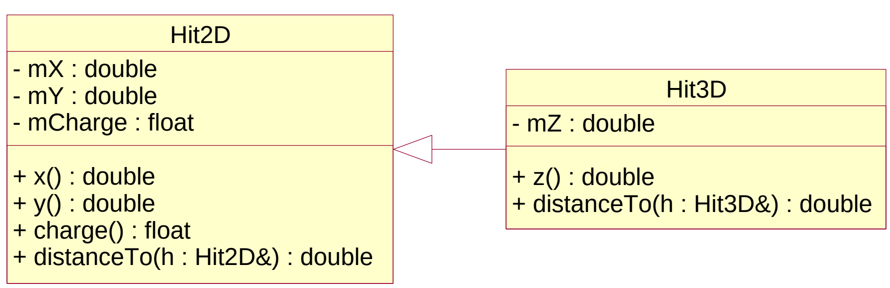

Figure A.5: Inheritance.

Fig. A.5 shows the form of the *inheritance* relationship. In this diagram we see that Hit3D is derived from Hit2D. If the name of a class would be shown in italics, it would indicate that the class is an abstract class. Note also that operations shown in italics indicate that they are pure virtual. The corresponding C++ code for the Hit3D class from Fig. A.5 would look like:

```
class Hit3D : public Hit2D {
public:
    double z();
    double distanceTo(Hit3D& h);
private:
    double mZ;
};
```

A.5 Aggregation and Association

The weak form of aggregation is denoted with an open diamond. This relationship denotes that the aggregate class (the class with the white diamond touching it) is in some way the ”whole”, and the other class in the relationship is somehow ”part” of that whole. Fig. A.6 shows an *aggregation* relationship. In this

Figure A.6: Aggregation.

case, the Track class contains many Hit instances. In UML the ends of a relationship are referred to as its ”roles”. Notice that the role at the Hit end of the aggregation is marked with a ”$*$”. This indicates that the Track contains many Hit instances. The following Listing shows how Fig. A.6 might be implemented in C++ as:

```
class Track {
public:
    // ...
private:
    vector<Hit*> mHits;
};
```

There are other forms of containment that do not have whole/part implications. For example, each *Vertex* refers back to its parent Track. This is not aggregation since it is not reasonable to consider a parent Track to be part of a child Vertex. We use the *association* relationship to depict this.

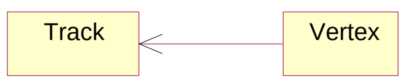

Figure A.7: Association.

Fig. A.7 shows how we draw an association. An association is nothing but a line drawn between the participating classes. In Fig. A.7 the association has an arrowhead to denote that Track does not necessarily know anything about Vertex. This relationship will almost certainly be implemented with a pointer of some kind.

What is the difference between an aggregation and an association? Aggregation denotes whole/part relationships whereas associations do not. However, there is not likely to be much difference in the way that the two relationships are implemented. That is, it would be very difficult to look at the code and determine whether a particular relationship ought to be aggregation or association. Aggregation and Association both correspond to the *has-by-reference* relationship.

A.6 Dependency

Sometimes the relationship between a two classes is very weak. They are not implemented with member variables at all. Rather they might be implemented as member function arguments.

Consider, for example, the fit function of a TrackFitter class. Suppose that this function takes an argument of type CalibrartionDB since it requires information from it (e.g. if the magnetic field was on or off) in order to perform the fit. Fig. A.8 shows a dashed arrow between the TrackFitter class and the CalibrartionDB

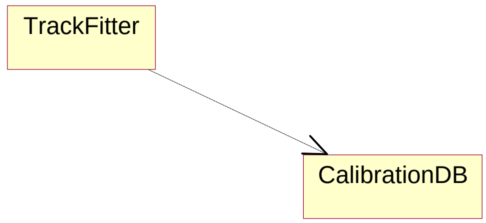

Figure A.8: Dependency.

class. This is the *dependency* relationship. This is often called a *using* relationship. This relationship simply means that TrackFitter somehow depends upon CalibrartionDB. In C++ this almost always results in a #include:

```
#include "CalibrartionDB.hh"
class TrackFitter {
public:
    // ...
    void fit(CalibrartionDB &db);
private:
    // ...
};
```

Index