StEvent

User Guide and Reference Manual
for Version 2

Revision: 2.10
Date: 1999/12/06 23:01:54

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.

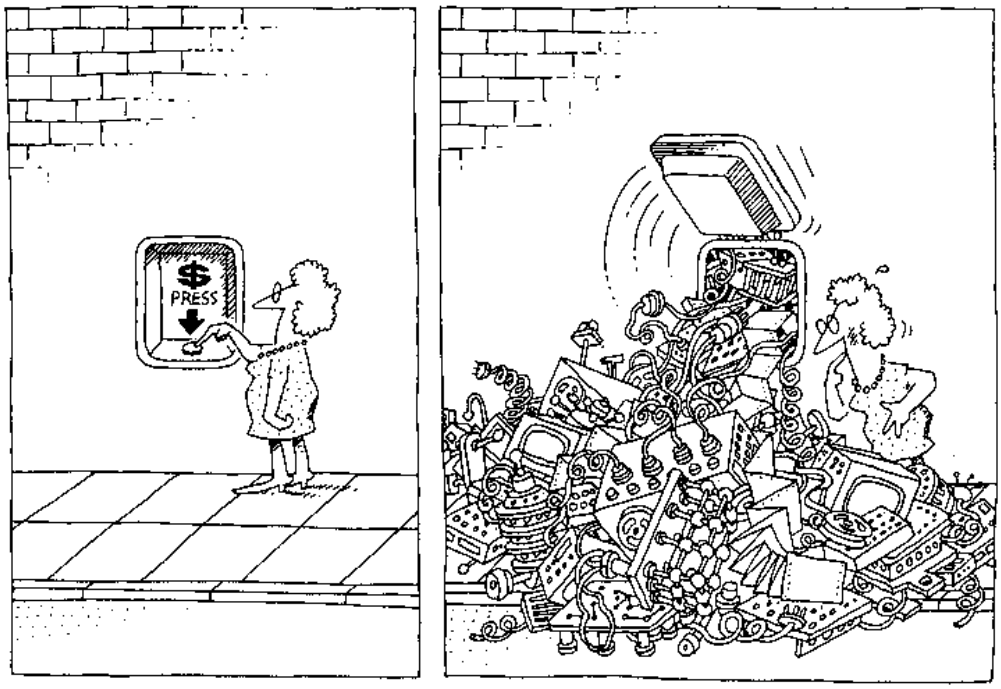

Figure 1.1: The task of the software development team is to engineer the illusion of simplicity.

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

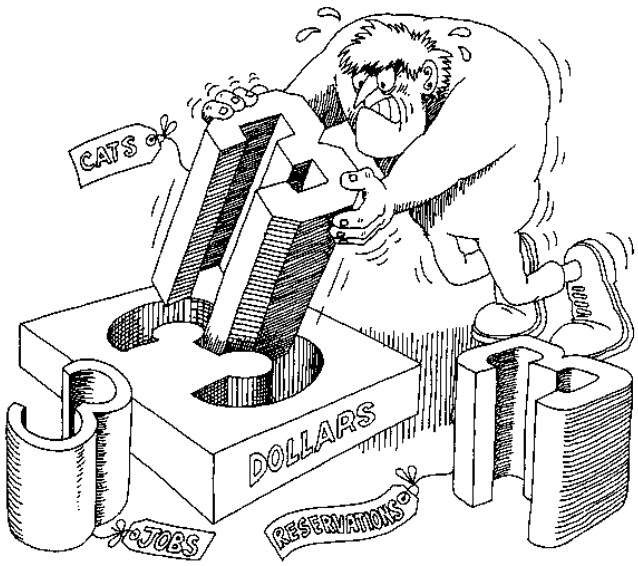

Figure 2.1: Strong typing avoids mixing abstractions.

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

```
enum StBeamDirection        {east, west};

enum StBeamPolarizationAxis {transverse, longitudinal};

enum StChargeSign           {negative, positive};

enum StTrackType            {global, primary, secondary};

enum StTrackModel           {helixModel, kalmanModel};

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

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

enum StDedxMethod           {kUndefinedMethodId,
```

```
                              kTruncatedMeanId,
                              kEnsembleTruncatedMeanId,
                              kLikelihoodFitId,
                              kWeightedTruncatedMeanId,
                              kOtherMethodId};

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

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

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. This includes not only indices as usual but also for example sector numbers, row numbers and wafer numbers. If you follow this rule things become less confusing for there is only one way of counting. This allows to follow the usual C/C++ syntax in all forms:

```
const int nSectors = 24;
for (int i=0; i<nSectors; i++)
    // ...
```

If you find a deviation from this rule it is a bug.

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;
```

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

```
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:

```
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).

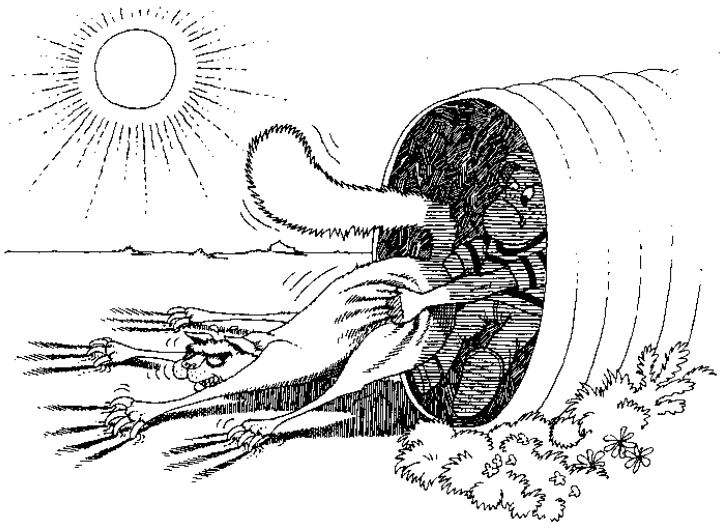

Figure 2.2: Persistence saves the state and class of an object across time or space.

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` $\leq$ `short` $\leq$ `int` $\leq$ `long` $\leq$ `long long` and `float` $\leq$ `double` $\leq$ `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
```

This is fine and good but there is absolutely no reason to use them in code other than in the definition of class data member. 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.

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(UInt_t nelem);`
`StSPtrVecClassName(UInt_t 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.

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

`void resize(UInt_t 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_t 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[](UInt_t 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_t doLoadTpcHits;`
Controls if TPC hits should be loaded (default=kTRUE).

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

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

`Bool_t doPrintRunInfo;`
Print or do not print a dump of the current `StRun` and `StRunSummary` instances (default=kFALSE).

`Bool_t 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_t 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_t 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. The object returned is actually of type `StBrowsableEvent`.

`StRun* run();`
Returns a pointer to the current `StRun` object. This object gets only updated for a new run, else you will get always the same instance.

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` or `StRun` object you get through these methods. They will 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 Run Header

The class `StRun` contains all information you would expect from a run header: run number, trigger settings, beam setup, and much more. Every living `StRun` object also contains a pointer to an instance of `StRunSummary` which contains - you guessed it - the run summary. The fact that the summary is contained by pointer already tells you that the pointer might be `NULL` if something went wrong. So better make sure it is non-zero before you dereference it.
The run summary contains information gathered during the DST reconstruction run like mean $p_\perp$, mean pseudo-rapidity, CPU time, and similar stuff. It does *not* contain all data on the experimental run but only on the events of the run which were handled in a single batch job. These are usually around 50 events, while an experimental run can contain thousands of events. Fig. 3.1 shows the UML class diagrams for the

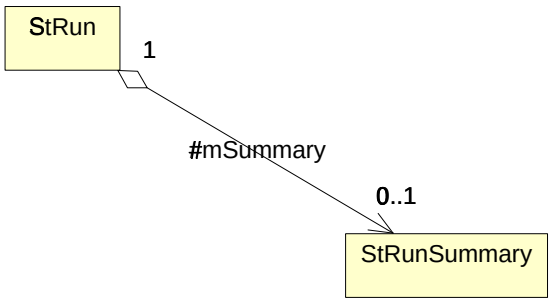

Figure 3.1: Class diagrams for `StRun` and `StRunSummary`.

two classes. Pretty simple. What is not shown is the relation to the `StEvent` class, simply because there is none. `StRun` and `StEvent` are completely separate entities. They don't know from each other. The `StRun` object is valid over many events, i.e. many generatins of StEvent events. `StEventMaker` (see Sec. 2.6 will give you the same one over and over again, unless there's a new run header on the DST.

Here is an example of a simple function `printRunInfo` which takes a pointer to an instance of `StRun` as argument and prints out some stuff.

```
void printRunInfo(StRun* run)
{
    if (!run) return;     // Oops, null pointer

    cout << "run id:                " << run->id() << endl;
```

```
    cout << "run type:              " << run->type() << endl;
    cout << "center of mass energy: " << run->centerOfMassEnergy() << endl;
    cout << "magnetic field:        " << run->magneticField() << endl;

    if (run->summary()) {
        cout << "# of events:           "
             << run->summary()->numberOfEvents() << endl;
        cout << "# of processed events:  "
             << run->summary()->numberOfProcessedEvents() << endl;
        cout << "CPU time used:         "
             << run->summary()->cpuSeconds() << endl;
        cout << "average luminosity:    "
             << run->summary()->averageLuminosity() << endl;
        cout << "<pt>:                  "
             << run->summary()->meanPt() << endl;
    }
}
```

This example also illustrates how useful it is to name the methods properly. This code actually could have been written simpler using the `PR(x)` macro defined in `StGlobals.hh` from the *StarClassLibrary*. Instead of

```
    cout << "magnetic field:        " << run->magneticField() << endl;
```

one then would write

```
    PR(run->magneticField());
```

which prints:

```
    run->magneticField() = 0.5
```

or whatever setting STAR was running at this time. As long as the names are descriptive this makes life much easier and provides the reader the same amount of information.

3.2 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 is a bit more complex than `StRun`. However, one thing is very similar: the summary. This is depicted in Fig. 3.2 which shows only a very small fraction of the class design around `StEvent`. The class `StEventSummary` contains lots of information 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. The same warning as for `StRun` applies here. Check the pointer to the event summary before you use it. It could be `NULL`.

The class `StBrowsableEvent` inherits from `StEvent` and therefore acts the same way but has some additional features which allow a better integration of StEvent into the ROOT framework. Since this is of

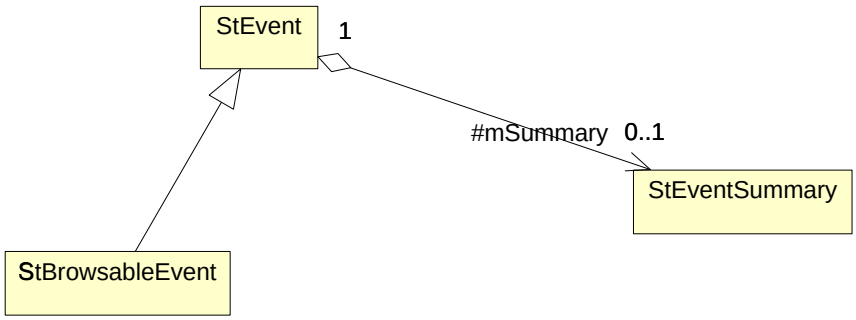

Figure 3.2: Class diagrams for `StEvent`, `StBrowsableEvent` and `StEventSummary`.

no big importance for the understanding of the data model as a whole we refer to section 2.6 and 4.1 for more details.

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 4.8.

3.3 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.

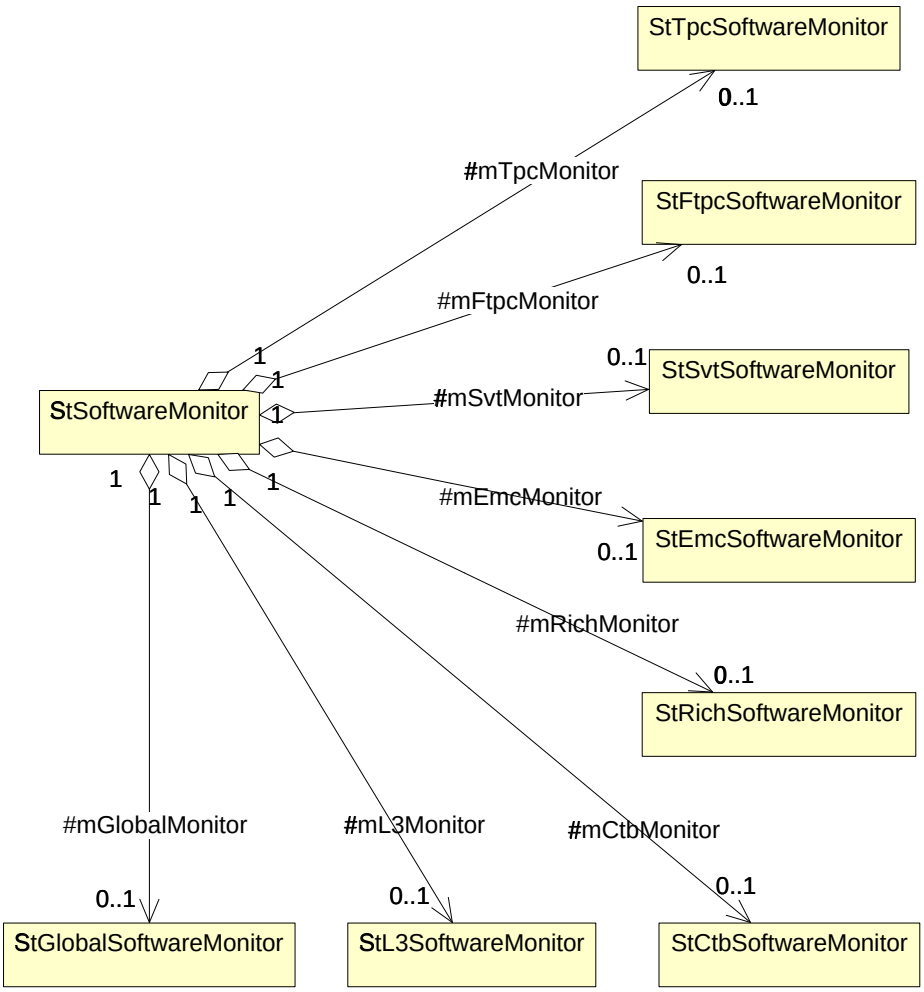

Figure 3.3: Class diagrams for the software monitors.

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. 3.3 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++) {
        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.

3.4 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. 3.4. The class contains

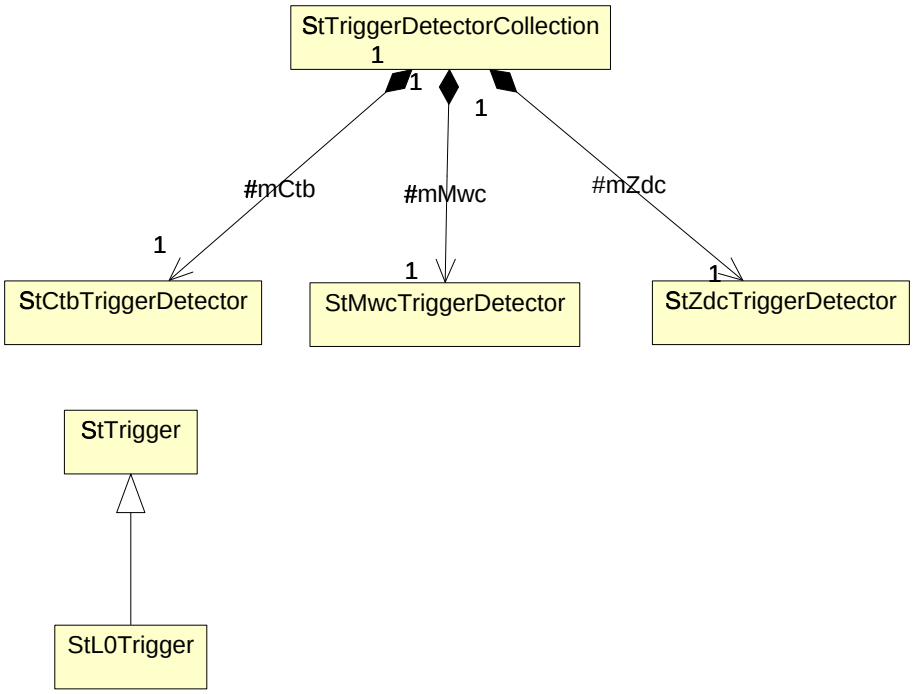

Figure 3.4: Class diagrams for the trigger detector collection and the `StTrigger` hierarchy.

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. 3.4. 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 col-

lection 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 << "  counter  |     mips    |     time   \n";
    cout << "-------------------------------------\n";

    for (int i=0; i<ctb.numberOfCtbCounters(); i++) {
        cout << setw(9) << i <<   "  | "
             << setw(10) << ctb.mips(i) << "  | "
             << ctb.time(i) << 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.

3.5 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.

3.5.1 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.

3.5.2 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

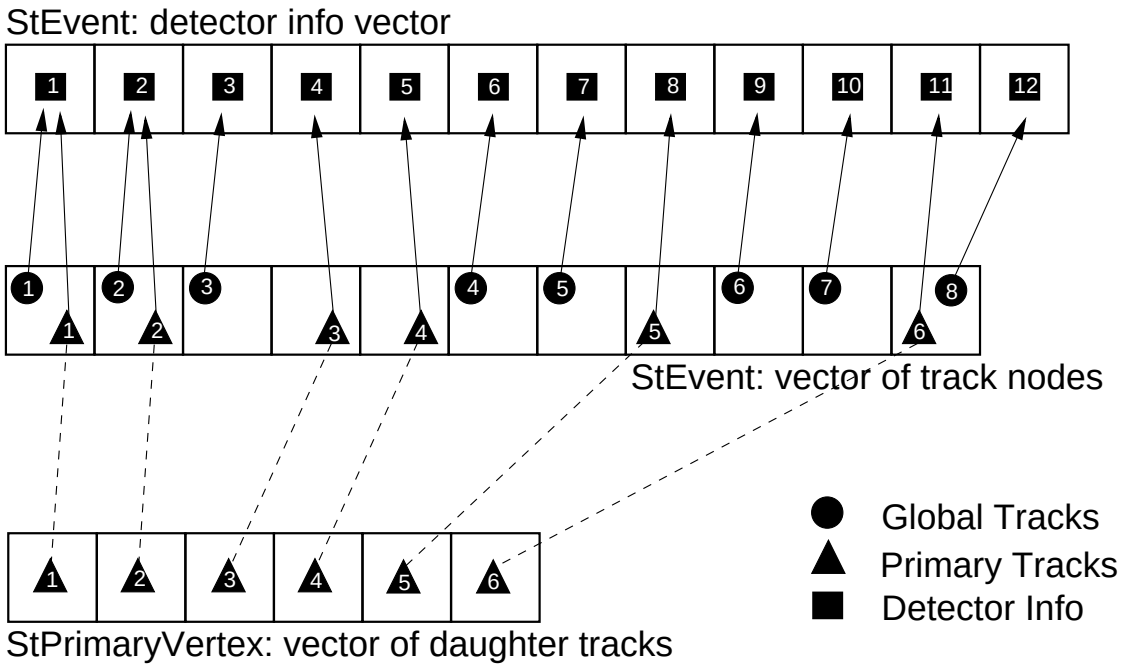

Figure 3.5: Schematic view of the track node collection and its relation to the detector info collection and the list of daughter tracks of the primary vertex.

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 tracks. This is depicted in Fig. 3.5. 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}.

3.5.3 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. 3.5. 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.

3.5.4 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. 3.6. 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. 3.6 `StTrack` is composed of several classes. It either contains them by value or by pointer. There are:

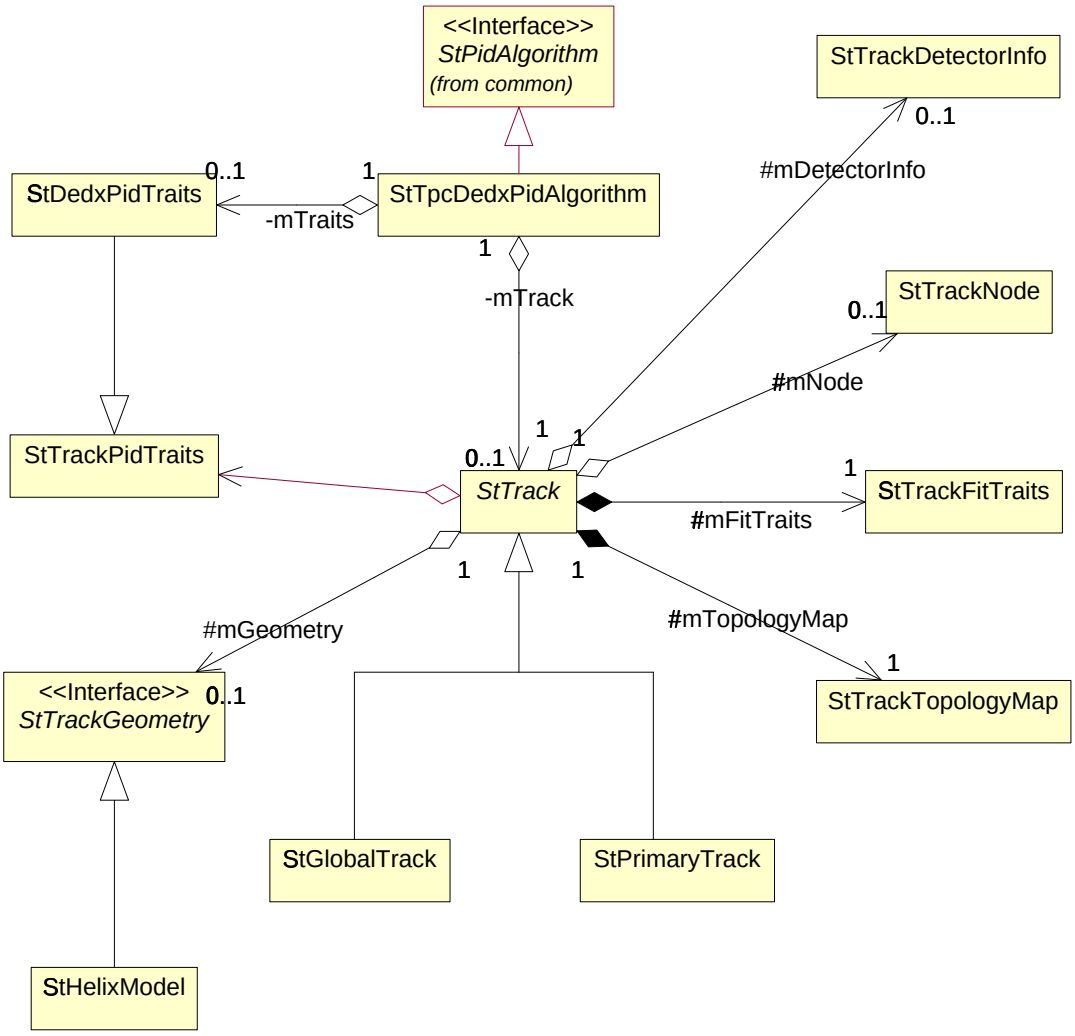

Figure 3.6: Class diagrams for `StGlobalTrack` and `StPrimaryTrack` including related classes and dependencies.

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`.

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 `StTrack-FitTraits` 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 3.5.2. `StTrack::node()` will return a pointer to the node the track belongs to.

[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.

StDetectorInfo See section 3.5.3. `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 3.5.5 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.

3.5.5 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 and sigma dE/dx, 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 placeholder 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;
}
```

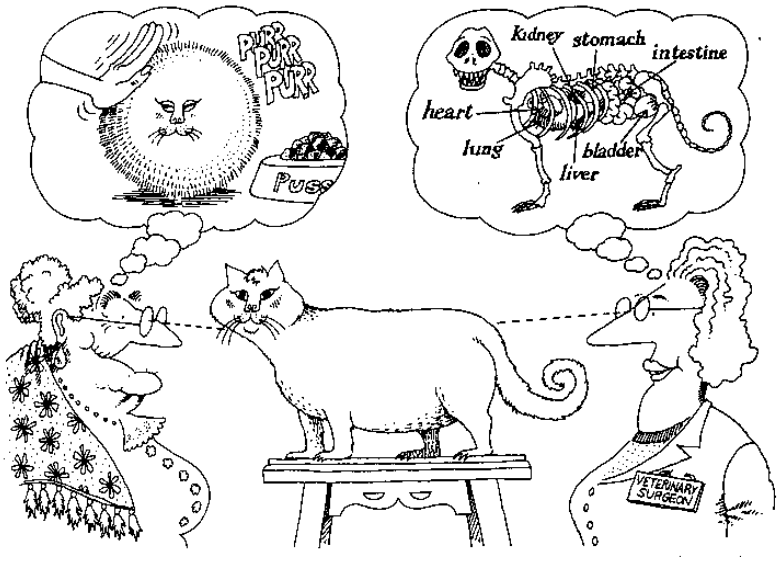

Figure 3.7: Abstraction focuses upon the essential characteristic of some object, relative to the perspective of the viewer.

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 (3.5.6) 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.)

3.5.6 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()->sigma() << 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 4.40 for details.

3.6 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 `StMeasured-Point` 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 four concrete vertex classes are:

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

`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. 3.8.

The primary vertex `StPrimaryVertex` class plays an important role since it holds all primary tracks. This is depicted in Fig. 3.5. 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.

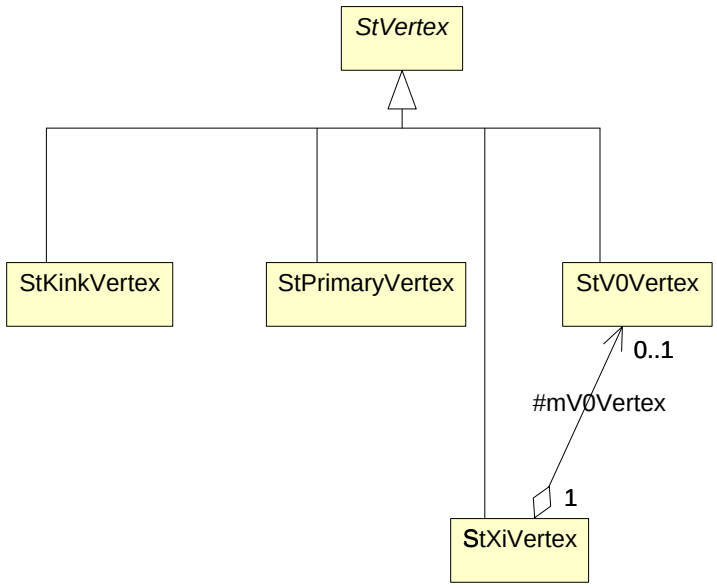

Figure 3.8: Vertex class diagrams and their dependencies.

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 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()`.

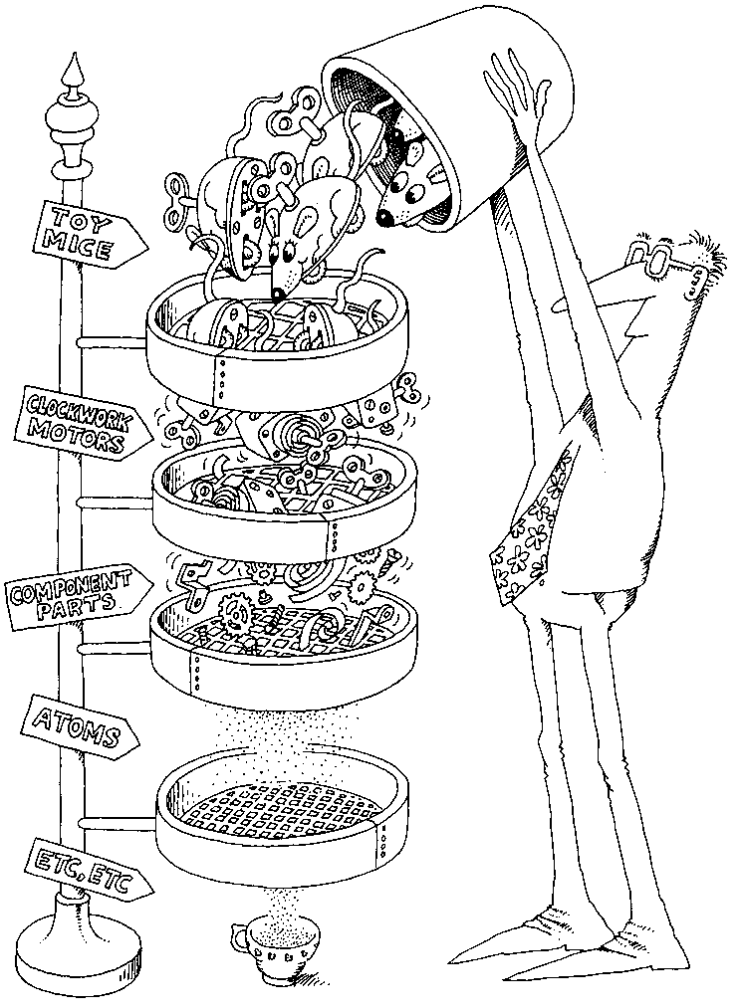

Figure 3.9: Abstraction from a hierarchy

3.7 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.

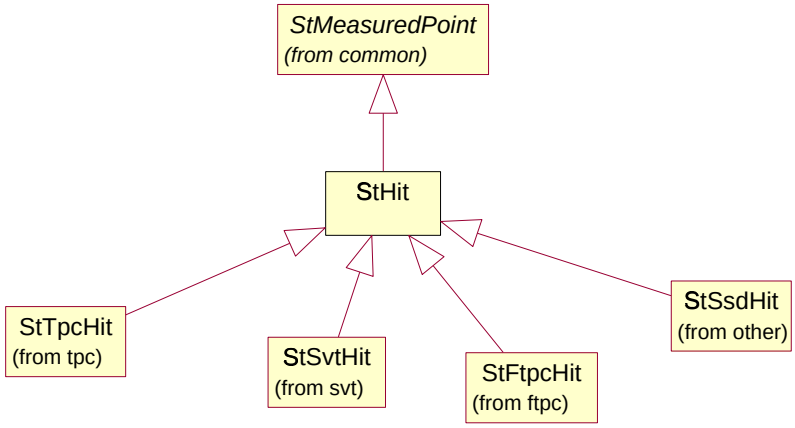

Figure 3.10: The `StHit` class and its subclasses and superclasses.

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. 3.10 shows their relation. In the following we describe the different classes in detail.

3.7.1 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 (0-23) and row number (0-44). Please read section 2.3.1 if you are puzzled about the numbering scheme. 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 4.41 for details.

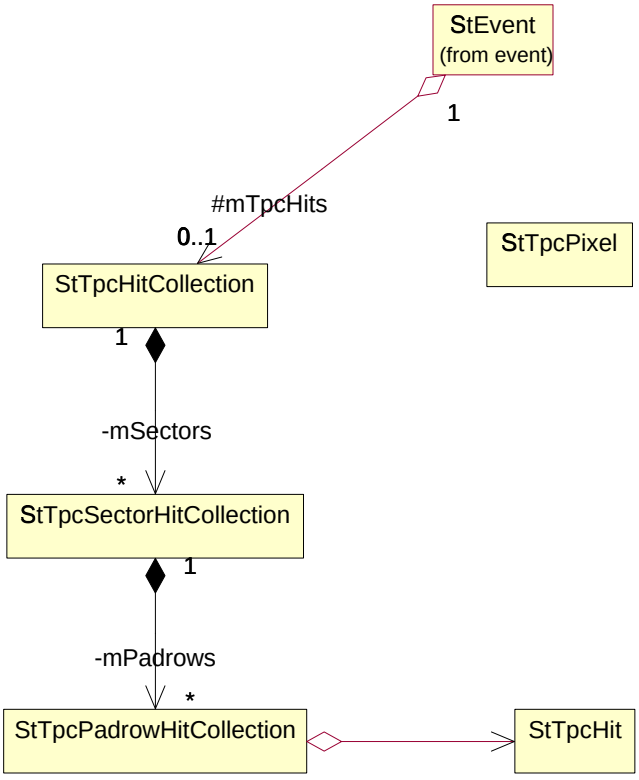

Figure 3.11: Class diagrams of the TPC hit storage scheme: sector/padrow.

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. 3.11. You get the idea when you look at the following example:

```
const int isec = 3;
const int irow = 24;

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.

3.7.2 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 (0–19) and then sectors (0–5). This is depicted in Fig. 3.12. 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 4.11 for details.

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

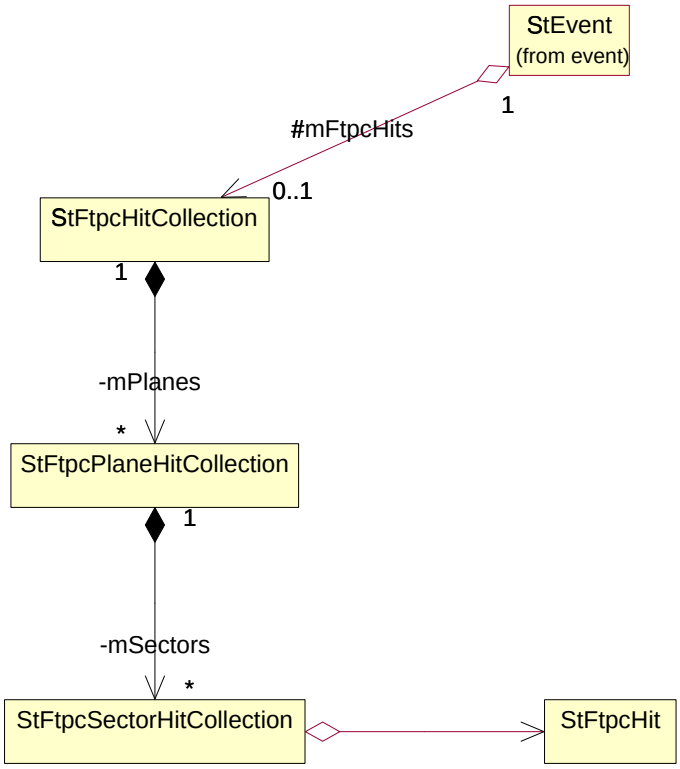

Figure 3.12: Class diagrams of the FTPC hit storage scheme: plane/sector.

```
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);
}
```

3.7.3 SVT hits

The SVT consist of 6 layers (or 3 barrels) with up to 8 ladders each. Each ladder has up to 7 wafers. Consequently the `StSvtHit` class provides method to return the local coordinates exactly in these "units": `barrel()` returns 0–2, `layer()` returns 0–5, `ladder()` returns 0–7, and `wafer()` returns 0–6. If you are puzzled why we start to count at 0 you better 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. 3.13. The top collection (`StSvtHitCollection`), the layer collection (`StSvtLayerHitCollection`), and the (`StSvtLadderHitCollection`) provide a method to return the number of hits stored in the referring subcomponent (`numberOfHits()`). The hits are stored per wafer.

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

```
const int ilayer   = 1;
const int iladder  = 2;
const int iwafer   = 3;

cout << "layer " << ilayer << " contains "
     << event->svtHitCollection()->layer(ilayer)->numberOfHits()
     << " hits" << endl;

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

cout << "layer " << ilayer << ", 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]->layer() == ilayer);
    assert(theHits[i]->wafer() == iwafer);
}
```

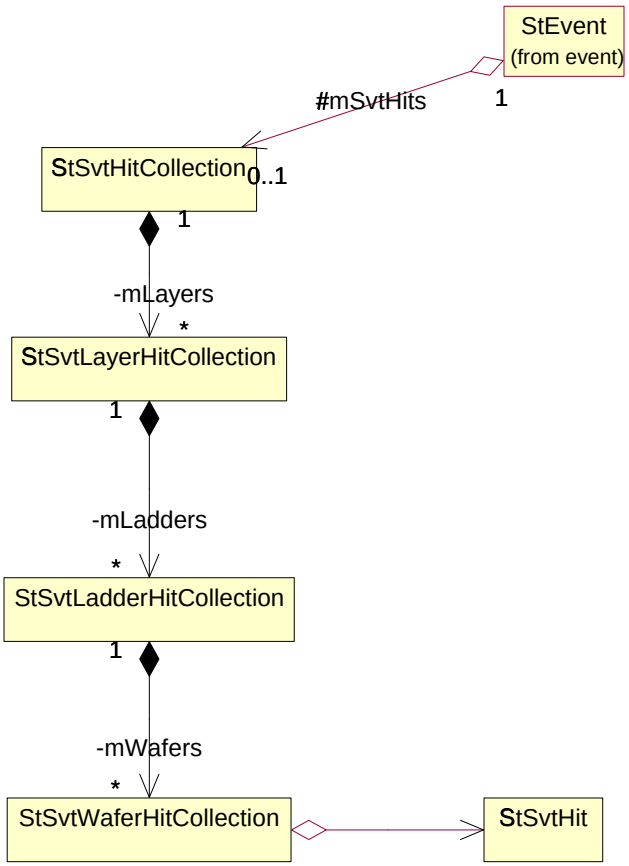

Figure 3.13: Class diagrams of the SVT hit storage scheme: layer/ladder/wafer.

3.8 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());
```

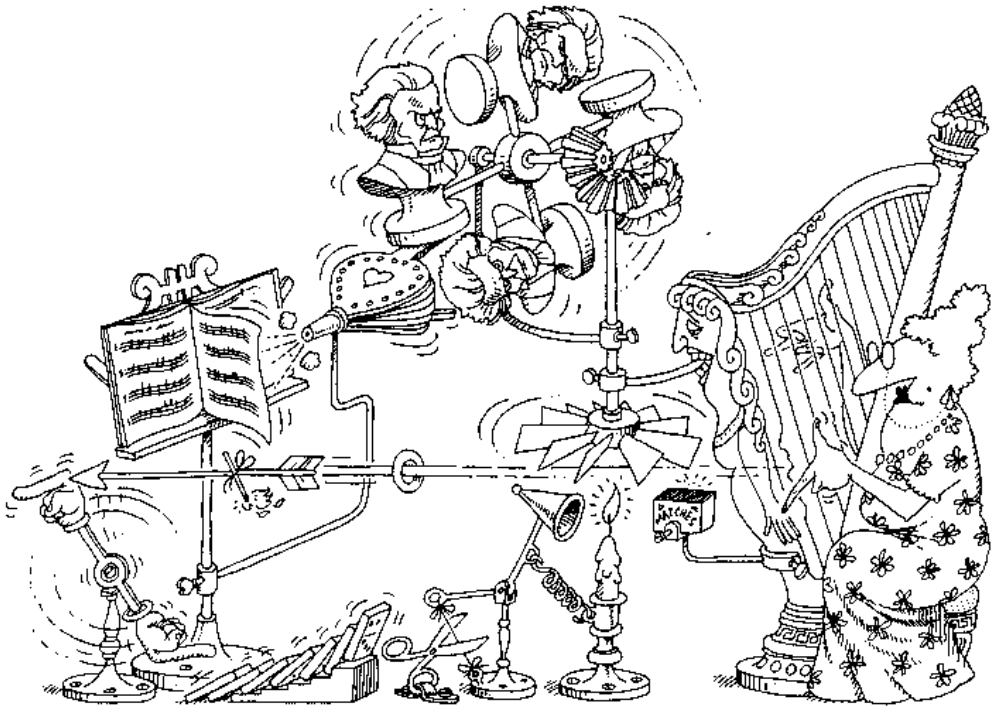

Figure 3.14: Mechanisms are the means whereby objects collaborate to provide some higher level behaviour.

Part II

Reference Manual

4 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.

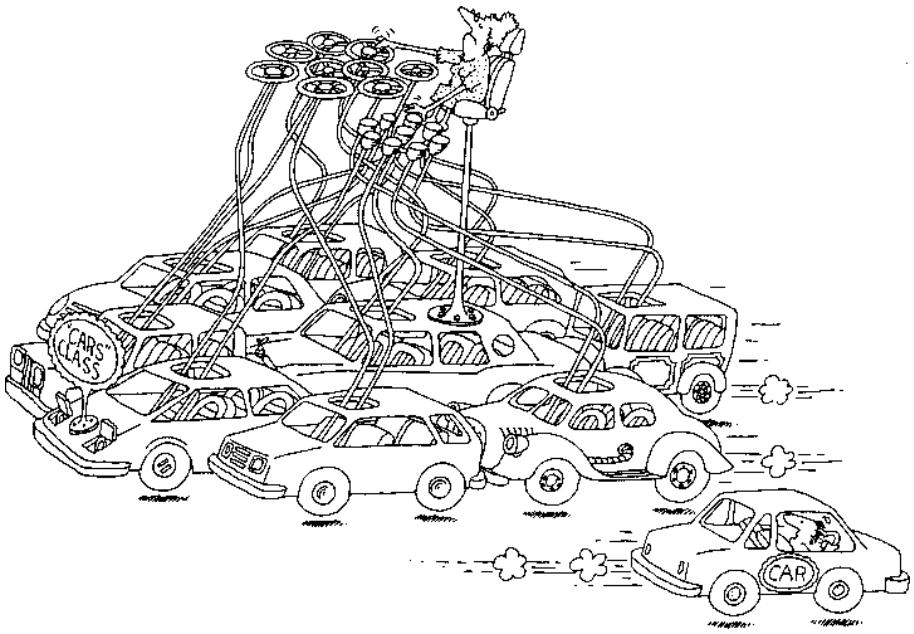

Figure 4.1: A class represents a set of objects that share a common structure and a common behavior.

4.1 StBrowsableEvent

Summary Version of `StEvent` with a higher degree of integration in ROOT.

Synopsis

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

Description `StBrowsableEvent` adds ROOT specific features to `StEvent` such as to allow the navigation through StEvent in a graphical data browser. `StEvent-Maker::event()` actually returns an instance of StBrowsableEvent not StEvent.

Related Classes `StBrowsableEvent` is derived from `StEvent`.

Public Constructors

```
StBrowsableEvent();

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

StBrowsableEvent(const event_header_st&);
```

Public Member Functions

```
void browse(TBrowser*);
```

4.2 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.

4.3 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

```
Long_t mult_ctb_tot;
```

Total multiplicity (or ADC sum) in CTB.

4.4 StCtbTriggerDetector

Summary Central Trigger Barrel (CTB) data.

Synopsis

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

Description

Related Classes

Public Constructors

```
StCtbTriggerDetector();

StCtbTriggerDetector(const dst_TrgDet_st&);
```

Public Member Functions

```
UInt_t numberOfCtbCounters() const;

Float_t mips(UInt_t) const;

Float_t time(UInt_t) const;

void setMips(UInt_t, Float_t);

void setTime(UInt_t, Float_t);
```

4.5 StDedxPidTraits

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StDedxPidTraits();
```
Default constructor.

```
StDedxPidTraits(StDetectorId det, Short_t emethod,
                UShort_t np, Float_t dedx, Float_t sig);
```
Create an instance of `StDedxPidTraits` for detector `det`, encoded method `emethod`, number of points `np`, dE/dx mean `dedx`, and sigma `sig`.

Public Member Functions

```
UShort_t numberOfPoints() const;
```
Number of points used to calculate the dE/dx value.

```
Float_t mean() const;
```
The derived dE/dx value.

```
Float_t sigma() const;
```
Returns the sigma of the dE/dx value.

```
StDedxMethod method() const;

Short_t encodedMethod() const;
```

4.6 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_t energy_emc;
```

Total energy (or ADC sum) in EMC.

4.7 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.

4.8 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`. `StBrowsableEvent` is derived from `StEvent`.

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
```
static const TString& cvsTag();
```
CVS tag of the version you are using.
```
const TString& type() const;
```
Character string which contains a short description of the type of the event you got.
```
Long_t id() const;
```
Unique event identifier.
```
Long_t runId() const;
```
Unique run identifier.
```
Long_t time() const;
```
Time when the event was taken.
```
ULong_t triggerMask() const;

ULong_t bunchCrossingNumber() const;

StEventSummary* summary();
const StEventSummary* summary() const;
```

Returns pointer to the event summary with many useful information for QA/QC and event characterization.

```
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.

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

```
StTriggerDetectorCollection* triggerDetectorCollection();
const StTriggerDetectorCollection* triggerDetectorCollec-
tion() const;
```
Returns pointer to the current trigger detector collection. Trigger detectors are CTB, ZDC, VPD, and MWC.

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

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

```
UInt_t 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(UInt_t i = 0);
const StPrimaryVertex* primaryVertex(UInt_t 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`).

```
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.

```
void setType(const Char_t*);
void setRunId(Long_t);
void setId(Long_t);
void setTime(Long_t);
void setTriggerMask(ULong_t);
void setBunchCrossingNumber(ULong_t);
void setSoftwareMonitor(StSoftwareMonitor*);
void setTpcHitCollection(StTpcHitCollection*);
void setFtpcHitCollection(StFtpcHitCollection*);
void setSvtHitCollection(StSvtHitCollection*);
void setTriggerDetectorCollection(StTriggerDetectorCollection*);
void setL0Trigger(StL0Trigger*);
void addPrimaryVertex(StPrimaryVertex*);
```

4.9 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

```
Long_t numberOfTracks() const;
Long_t numberOfGoodTracks() const;
Long_t numberOfGoodTracks(StChargeSign) const;
Long_t numberOfGoodPrimaryTracks() const;
Long_t numberOfExoticTracks() const;
Long_t numberOfVertices() const;
Long_t numberOfVerticesOfType(StVertexId) const;
Long_t numberOfPileupVertices() const;
Float_t meanPt() const;
Float_t meanPt2() const;
Float_t meanEta() const;
Float_t rmsEta() const;
const StThreeVectorF& primaryVertexPosition() const;
UInt_t numberOfBins() const;
Long_t tracksInEtaBin(UInt_t) const;
Long_t tracksInPhiBin(UInt_t) const;
Long_t tracksInPtBin(UInt_t) const;
Float_t energyInEtaBin(UInt_t) const;
Float_t energyInPhiBin(UInt_t) const;
Float_t lowerEdgeEtaBin(UInt_t) const;
Float_t upperEdgeEtaBin(UInt_t) const;
Float_t lowerEdgePhiBin(UInt_t) const;
Float_t upperEdgePhiBin(UInt_t) const;
Float_t lowerEdgePtBin(UInt_t) const;
Float_t upperEdgePtBin(UInt_t) const;
Double_t magneticField() const;
void setNumberOfTracks(Long_t);
void setNumberOfGoodTracks(Long_t);
void setNumberOfGoodTracks(StChargeSign, Long_t);
void setNumberOfGoodPrimaryTracks(Long_t);
```

```
void setNumberOfNegativeTracks(Long_t);
void setNumberOfExoticTracks(Long_t);
void setNumberOfVertices(Long_t);
void setNumberOfVerticesForType(StVertexId, Long_t);
void setNumberOfPileupVertices(Long_t);
void setMeanPt(Float_t);
void setMeanPt2(Float_t);
void setMeanEta(Float_t);
void setRmsEta(Float_t);
void setPrimaryVertexPosition(const StThreeVectorF&);
void setMagneticField(Double_t);
```

4.10 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.

4.11 StFtpcHit

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StFtpcHit();

StFtpcHit(const StThreeVectorF&,
          const StThreeVectorF&,
          ULong_t, Float_t, UChar_t = 0);

StFtpcHit(const dst_point_st&);
```

Public Member Functions

```
ULong_t sector() const;
```
Returns sector number running from 0–5.

```
ULong_t plane() const;
```
Returns plane number running from 0–19.

```
ULong_t padsInHit() const;
ULong_t timebinsInHit() const;
```

4.12 StFtpcHitCollection

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StFtpcHitCollection();
```

Public Member Functions

```
Bool_t addHit(StFtpcHit*);
```

```
ULong_t numberOfHits() const;
```
Total number of FTPC hits stored in the collection.

```
UInt_t numberOfPlanes() const;
```

```
StFtpcPlaneHitCollection* plane(UInt_t i);
const StFtpcPlaneHitCollection* plane(UInt_t i) const;
```
Index `i` runs from 0–(n-1) where n = `numberOfPlanes()`.

4.13 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

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

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

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

4.14 StFtpcSectorHitCollection

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StFtpcSectorHitCollection();
```

Public Member Functions

```
StSPtrVecFtpcHit& hits();

const StSPtrVecFtpcHit& hits() const;
```

4.15 StFtpcSoftwareMonitor

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StFtpcSoftwareMonitor();
```

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

Public Data Member

`Long_t n_clus_ftpc[2];`
Total number of clusters in FTPC, east/west.

`Long_t n_pts_ftpc[2];`
Total number of space points in FTPC, east/west.

`Long_t n_trk_ftpc[2];`
Total number of tracks in FTPC east/west .

`Float_t chrg_ftpc_tot[2];`
Total charge deposited in FTPC, east/west.

`Float_t hit_frac_ftpc[2];`
Fraction of hits used in FTPC, east/west.

`Float_t avg_trkL_ftpc[2];`
Average track length (cm) FTPC, east/west
or average number of points assigned.

`Float_t res_pad_ftpc[2];`
Average residual, pad direction, FTPC east/west.

`Float_t res_drf_ftpc[2];`
Average residual, drift direction, FTPC east/west.

4.16 StFunctional

Summary

Synopsis `#include "StFunctional.h"`

Description

4.17 StGlobalSoftwareMonitor

Summary

Synopsis

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

Description

Related Classes

Public Constructors

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

Public Data Member

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

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

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

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

4.18 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;
const StVertex* vertex() const;
```

4.19 StHelixModel

Summary

Synopsis

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

Description

Related Classes Inherits directly from `StTrackGeometry`.

Public Constructors

```
StHelixModel();
```

```
StHelixModel(Short_t q, Float_t psi, Float_t c, Float_t dip,
             const StThreeVectorF& o, const StThreeVec-
torF& p);
```

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

Public Member Functions

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

```
Short_t charge() const;
```
Charge in units of +e.

```
Double_t curvature() const;
```
Curvature in m^{-1}.

```
Double_t psi() const;
```
Psi in radians.

```
Double_t dipAngle() const;
```
Dip angle in radians.

```
const StThreeVectorF& origin() const;
```
Origin in cm.

```
const StThreeVectorF& momentum() const;
```
Momentum in GeV/c.

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

4.20 StHit

Summary

Synopsis

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

Description

Related Classes `StHit` is derived form `StMeasuredPoint`.

Public Constructors

```
StHit();
StHit(const StThreeVectorF&,
      const StThreeVectorF&,
      ULong_t, Float_t, UChar_t = 0);
```

Public Member Functions

```
Float_t charge() const;
UChar_t trackReferenceCount() const;
StDetectorId detector() const;
StThreeVectorF positionError() const; // overwrite inher-
ited
StMatrixF covariantMatrix() const; // overwrite inherited
void setCharge(Float_t);
void setTrackReferenceCount(UChar_t);
void setHardwarePosition(ULong_t);
void setPositionError(const StThreeVectorF&);
StPtrVecTrack relatedTracks(const StSPtrVecTrackNode&, StTrack-
Type);
```

Public Member Operators

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

4.21 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;
UInt_t numberOfDaughters() const;
StTrack* daughter(UInt_t = 0);
const StTrack* daughter(UInt_t = 0) const;
StPtrVecTrack daughters(StTrackFilter&);
StParticleDefinition* pidParent() const;
StParticleDefinition* pidDaughter() const;
UShort_t geantIdParent() const;
UShort_t geantIdDaughter() const;
Float_t dcaParentDaughter() const;
Float_t dcaDaughterPrimaryVertex() const;
Float_t dcaParentPrimaryVertex() const;
Float_t hitDistanceParentDaughter() const;
Float_t hitDistanceParentVertex() const;
Float_t dE(UInt_t i) const;
Float_t decayAngle() const;
Float_t decayAngleCM() const;
const StThreeVectorF& parentMomentum() const;
StThreeVectorF& parentMomentum();
const StThreeVectorF& daughterMomentum() const;
StThreeVectorF& daughterMomentum();
void setGeantIdParent(UShort_t);
void setGeantIdDaughter(UShort_t);
void setDcaParentDaughter(Float_t);
void setDcaDaughterPrimaryVertex(Float_t);
void setDcaParentPrimaryVertex(Float_t);
void setHitDistanceParentDaughter(Float_t);
void setHitDistanceParentVertex(Float_t);
void setdE(UInt_t, Float_t);
void setDecayAngle(Float_t);
void setDecayAngleCM(Float_t);
```

```
void setParentMomentum(const StThreeVectorF&);
void setDaughterMomentum(const StThreeVectorF&);
void addDaughter(StTrack*);
void removeDaughter(StTrack*);
```

4.22 StL0Trigger

Summary

Synopsis

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

Description

Related Classes

Public Constructors

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

Public Member Functions

```
UInt_t coarsePixelArraySize();
Long_t coarsePixelArray(UInt_t);
Long_t mwcCtbMultiplicity() const;
Long_t mwcCtbDipole() const;
Long_t mwcCtbTopology() const;
Long_t mwcCtbMoment() const;

void setMwcCtbMultiplicity(Long_t);
void setMwcCtbDipole(Long_t);
void setMwcCtbTopology(Long_t);
void setMwcCtbMoment(Long_t);
void setCoarsePixelArray(UInt_t, Long_t);
```

4.23 StL3SoftwareMonitor

Summary

Synopsis

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

Description

Related Classes

Public Constructors

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

Public Data Member

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

`Long_t id_hardware;`
Id of the hardware configuration.

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

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

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

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

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

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

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

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

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

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

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

4.24 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_t operator==(const StMeasuredPoint&) const;
Int_t operator!=(const StMeasuredPoint&) const;
```

4.25 StMwcTriggerDetector

Summary

Synopsis

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

Description

Related Classes

Public Constructors

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

Public Member Functions

```
UInt_t numberOfMwcSectors() const;
Float_t mips(UInt_t) const;
void setMips(UInt_t, Float_t);
```

4.26 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;
void setVertex(StVertex*);
```

4.27 StPrimaryVertex

Summary

Synopsis

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

Description

Related Classes

Public Constructors

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

Public Member Functions

```
StVertexId type() const;
UInt_t numberOfDaughters() const;
StTrack* daughter(UInt_t);
const StTrack* daughter(UInt_t) const;
StPtrVecTrack daughters(StTrackFilter&);
StSPtrVecPrimaryTrack& daughters();
const StSPtrVecPrimaryTrack& daughters() const;
void addDaughter(StTrack*);
void removeDaughter(StTrack*);
void setParent(StTrack*); // overwrite inherited
```

4.28 StRichPixel

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StRichPixel();
StRichPixel(UShort_t pad, UShort_t adc);
StRichPixel(const dst_rch_pixel_st&);
```

Public Member Functions

```
UShort_t module() const;
UShort_t channel() const;
UShort_t adc() const;
```

Public Member Operators

```
Int_t operator==(const StRichPixel&) const;
Int_t operator!=(const StRichPixel&) const;
```

4.29 StRichSoftwareMonitor

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StRichSoftwareMonitor();
StRichSoftwareMonitor(const dst_mon_soft_rich_st&);
```

Public Data Member

```
Long_t mult_rich_tot;
```

Total multiplicity (or ADC sum) in RICH.

4.30 StRun

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StRun();
StRun(const run_header_st&, const dst_run_summary_st&);
StRun(const run_header_st&);
```

Public Member Functions

```
Long_t id() const;
Long_t bfcId() const;
const TString& type() const;
Long_t triggerMask() const;
Double_t centerOfMassEnergy() const;
Short_t beamMassNumber(StBeamDirection) const;
Short_t beamCharge(StBeamDirection) const;
Double_t magneticField() const;
StRunSummary* summary();
const StRunSummary* summary() const;
static const TString& cvsTag();
void setId(Long_t);
void setBfcId(Long_t);
void setType(const Char_t*);
void setTriggerMask(Long_t);
void setCenterOfMassEnergy(Double_t);
void setBeamMassNumber(StBeamDirection, Short_t);
void setBeamCharge(StBeamDirection, Short_t);
void setSummary(StRunSummary*);
void setMagneticField(Double_t);
```

Public Member Operator

```
Int_t operator==(const StRun&) const;
Int_t operator!=(const StRun&) const;
```

4.31 StRunSummary

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StRunSummary();
StRunSummary(const dst_run_summary_st&);
```

Public Member Functions

```
ULong_t numberOfEvents() const;
ULong_t numberOfProcessedEvents() const;
Long_t startTime() const;
Long_t stopTime() const;
Float_t cpuSeconds() const;
Float_t averageBeamPolarization(StBeamDirection, StBeam-
PolarizationAxis) const;
Float_t averageLuminosity() const;
Float_t meanEta() const;
Float_t rmsEta() const;
Float_t meanPt() const;
Float_t rmsPt() const;
Float_t meanNumberOfVertices() const;
Float_t rmsNumberOfVertices() const;
Float_t meanMultiplicity(StDetectorId) const;
Float_t rmsMultiplicity(StDetectorId) const;
void setNumberOfEvents(ULong_t);
void setNumberOfProcessedEvents(ULong_t);
void setStartTime(Long_t);
void setStopTime(Long_t);
void setCpuSeconds(Float_t);
void setAverageBeamPolarization(StBeamDirection, StBeam-
PolarizationAxis, Float_t);
void setAverageLuminosity(Float_t);
void setMeanEta(Float_t);
void setRmsEta(Float_t);
void setMeanPt(Float_t);
void setRmsPt(Float_t);
void setMeanNumberOfVertices(Float_t);
void setRmsNumberOfVertices(Float_t);
void setMeanMultiplicity(StDetectorId, Float_t);
```

```
void setRmsMultiplicity(StDetectorId, Float_t);
```

4.32 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;
void setTpcSoftwareMonitor(StTpcSoftwareMonitor*);
void setSvtSoftwareMonitor(StSvtSoftwareMonitor*);
void setFtpcSoftwareMonitor(StFtpcSoftwareMonitor*);
void setEmcSoftwareMonitor(StEmcSoftwareMonitor*);
void setRichSoftwareMonitor(StRichSoftwareMonitor*);
void setCtbSoftwareMonitor(StCtbSoftwareMonitor*);
```

```
void setGlobalSoftwareMonitor(StGlobalSoftwareMonitor*);
void setL3SoftwareMonitor(StL3SoftwareMonitor*);
```

4.33 StSsdHit

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StSsdHit();
StSsdHit(const StThreeVectorF&,
         const StThreeVectorF&,
         ULong_t, Float_t, UChar_t = 0);
StSsdHit(const dst_point_st&);
```

Public Member Functions

```
ULong_t centralStripNSide() const;
```

Runs from 0–767.

```
ULong_t centralStripPSide() const;
```

Runs from 0–767.

```
ULong_t clusterSizeNSide() const;
ULong_t clusterSizePSide() const;
ULong_t matchingQualityFactor() const;
```

4.34 StSvtHit

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StSvtHit();
StSvtHit(const StThreeVectorF&,
         const StThreeVectorF&,
         ULong_t, Float_t, UChar_t = 0);
StSvtHit(const dst_point_st&);
```

Public Member Functions

```
ULong_t layer() const;
```
Layer in which hit is located. Layer number runs from 0–5.

```
ULong_t ladder() const;
```
Ladder number runs from 0–7.

```
ULong_t wafer() const;
```
Wafer number runs from 0–6.

```
ULong_t barrel() const; // barrel=[0-2]
```
Barrel number runs from 0–2.

```
ULong_t hybrid() const;
```

4.35 StSvtHitCollection

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StSvtHitCollection();
```

Public Member Functions

```
Bool_t addHit(StSvtHit*);
ULong_t numberOfHits() const;
UInt_t numberOfLayers() const;
StSvtLayerHitCollection* layer(UInt_t);
const StSvtLayerHitCollection* layer(UInt_t) const;
```

4.36 StSvtLadderHitCollection

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StSvtLadderHitCollection();
```

Public Member Functions

```
ULong_t numberOfHits() const;
UInt_t numberOfWafers() const;
StSvtWaferHitCollection* wafer(UInt_t);
const StSvtWaferHitCollection* wafer(UInt_t) const;
void setLayerNumber(Int_t);
```

4.37 StSvtLayerHitCollection

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StSvtLayerHitCollection();
```

Public Member Functions

```
ULong_t numberOfHits() const;
UInt_t numberOfLadders() const;
StSvtLadderHitCollection* ladder(UInt_t);
const StSvtLadderHitCollection* ladder(UInt_t) const;
void setLayerNumber(Int_t);
```

4.38 StSvtSoftwareMonitor

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StSvtSoftwareMonitor();
StSvtSoftwareMonitor(const dst_mon_soft_svt_st&);
```

Public Data Member

`Long_t n_clus_svt[4];`
Total number clusters in each SVT layer.

`Long_t n_pts_svt[4];`
Total number of space points in each SVT layer.

`Long_t n_trk_svt;`
Total number of tracks in SVT.

`Float_t chrg_svt_tot[4];`
Total charge deposition in each SVT layer.

`Float_t hit_frac_svt[4];`
Fraction of hits used in each SVT layer.

`Float_t avg_trkL_svt;`
Average track length (cm) SVT
or average number of points assigned.

`Float_t res_pad_svt;`
Average residual, pad direction, SVT
or average chisq(1) of fit.

`Float_t res_drf_svt;`
Average residuals, drift direction, SVT
or average chisq(2) of fit.

4.39 StSvtWaferHitCollection

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StSvtWaferHitCollection();
```

Public Member Functions

```
StSPtrVecSvtHit& hits();
const StSPtrVecSvtHit& hits() const;
```

4.40 StTpcDedxPidAlgorithm

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTpcDedxPidAlgorithm();
```

Public Member Functions

```
StParticleDefinition* operator() (const StTrack&, const StSP-
trVecTrackPidTraits&);
const StDedxPidTraits* traits() const;
double numberOfSigma(const StParticleDefinition*) const;
double meanPidFunction(const StParticleDefinition*) const;
double sigmaPidFunction(const StParticleDefinition*) const;
```

4.41 StTpcHit

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTpcHit();
StTpcHit(const StThreeVectorF&,
         const StThreeVectorF&,
         ULong_t, Float_t, UChar_t = 0);
StTpcHit(const dst_point_st&);
```

Public Member Functions

```
ULong_t sector() const; // 0-23
ULong_t padrow() const; // 0-44
ULong_t padsInHit() const;
ULong_t pixelsInHit() const;
```

4.42 StTpcHitCollection

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTpcHitCollection();
```

Public Member Functions

```
Bool_t addHit(StTpcHit*);
```

```
ULong_t numberOfHits() const;
```
Total number of TPC hits in the collection.

```
UInt_t numberOfSectors() const;
```

```
StTpcSectorHitCollection* sector(UInt_t i);
const StTpcSectorHitCollection* sector(UInt_t i) const;
```
Index `i` runs from 0–(n-1) where n = `numberOfSectors()`.

4.43 StTpcPadrowHitCollection

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTpcPadrowHitCollection();
```

Public Member Functions

```
StSPtrVecTpcHit& hits();
const StSPtrVecTpcHit& hits() const;
```

4.44 StTpcPixel

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTpcPixel();
StTpcPixel(UShort_t, ULong_t);
StTpcPixel(const dst_pixel_st&);
```

Public Member Functions

```
UShort_t detector() const;
UShort_t sector() const;
UShort_t row() const;
ULong_t pad() const;
ULong_t timebin() const;
ULong_t adc() const;
```

Public Member Operator

```
Int_t operator==(const StTpcPixel&) const;
Int_t operator!=(const StTpcPixel&) const;
```

4.45 StTpcSectorHitCollection

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTpcSectorHitCollection();
```

Public Member Functions

```
ULong_t numberOfHits() const;
```
Total number of hits in the sector.

```
UInt_t numberOfPadrows() const;
```
Always 45.

```
StTpcPadrowHitCollection* padrow(UInt_t i);
const StTpcPadrowHitCollection* padrow(UInt_t) const;
```
Returns padrow hit collection. Note that `i`=0–44.

4.46 StTpcSoftwareMonitor

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTpcSoftwareMonitor();
StTpcSoftwareMonitor(const dst_mon_soft_tpc_st&);
```

Public Data Member

`Long_t n_clus_tpc_tot;`
Total number of clusters in TPC.

`Long_t n_clus_tpc_in[24];`
Total number of clusters in inner TPC sectors.

`Long_t n_clus_tpc_out[24];`
Total number of clusters in outer TPC sectors.

`Long_t n_pts_tpc_tot;`
Total number of space points in TPC.

`Long_t n_pts_tpc_in[24];`
Total number of space points in inner TPC sectors.

`Long_t n_pts_tpc_out[24];`
Total number of space points in outer TPC sectors.

`Long_t n_trk_tpc[2];`
Total number of tracks in TPC, tan(dip angle) < 0 (≥ 0).

`Float_t chrg_tpc_drift[10];`
Charge deposited in TPC in along z.

`Float_t chrg_tpc_tot;`
Total charge deposition in TPC.

`Float_t chrg_tpc_in[24];`
Total charge deposition in inner TPC sectors.

`Float_t chrg_tpc_out[24];`
Total charge deposition in outer TPC sectors.

`Float_t hit_frac_tpc[2];`
Fraction of hits used in TPC, tan(dip angle) < 0 (≥ 0).

`Float_t avg_trkL_tpc[2];`
Average track length (cm)
or average number of assigned, tan(dip angle) < 0 (≥ 0).

`Float_t res_drf_tpc[2];`
Average residuals, drift direction,
or average chisq(2) of fit, tan(dip angle) < 0 (≥ 0).

4.47 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;
const StVertex* vertex() const = 0;
UShort_t key() const;
Short_t flag() const;
UShort_t encodedMethod() const;
Bool_t finderScheme(StTrackFinderScheme) const;
StTrackFittingMethod fittingMethod() const;
Float_t impactParameter() const;
Float_t length() const;
UShort_t numberOfPossiblePoints() const;
UShort_t numberOfPossiblePoints(StDetectorId) const;
const StTrackTopologyMap& topologyMap() const;
StTrackGeometry* geometry();
const StTrackGeometry* geometry() const;
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_t);
void setEncodedMethod(UShort_t);
void setImpactParameter(Float_t);
void setLength(Float_t);
void setTopologyMap(const StTrackTopologyMap&);
void setGeometry(StTrackGeometry*);
```

```
void setFitTraits(const StTrackFitTraits&);
void addPidTraits(StTrackPidTraits*);
void setDetectorInfo(StTrackDetectorInfo*);
void setNode(StTrackNode*);
```

4.48 StTrackDetectorInfo

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTrackDetectorInfo();
StTrackDetectorInfo(const dst_track_st&);
```

Public Member Functions

```
const StThreeVectorF& firstPoint() const;
const StThreeVectorF& lastPoint() const;
UShort_t numberOfPoints() const;
UShort_t numberOfPoints(StDetectorId) const;
StPtrVecHit hits(StDetectorId) const;
StPtrVecHit hits(StHitFilter&) const;
StPtrVecHit& hits();
const StPtrVecHit& hits() const;
void setFirstPoint(const StThreeVectorF&);
void setLastPoint(const StThreeVectorF&);
void setNumberOfPoints(UShort_t);
void addHit(StHit*);
void removeHit(StHit*&);
```

4.49 StTrackFitTraits

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTrackFitTraits();
StTrackFitTraits(UShort_t, UShort_t, Float_t[2], Float_t[15]);
StTrackFitTraits(const dst_track_st&);
```

Public Member Functions

```
UShort_t numberOfFitPoints() const;
```
Total number of points used for the fit.

```
UShort_t 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_t chi2(UInt_t i = 0) const;
```
Depending on the fitting method there is one or two χ^2 values.

4.50 StTrackGeometry

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTrackGeometry();
StTrackGeometry(const dst_track_st&);
```

Public Member Functions

```
StTrackModel model() const = 0;
Short_t charge() const = 0;
Double_t curvature() const = 0;
Double_t psi() const = 0;
Double_t dipAngle() const = 0;
const StThreeVectorF& origin() const = 0;
const StThreeVectorF& momentum() const = 0;
StPhysicalHelixD helix() const = 0;
```

4.51 StTrackNode

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTrackNode();
```

Public Member Functions

```
void addTrack(StTrack*);
void removeTrack(StTrack*);
UInt_t entries() const;
StTrack* track(UInt_t);
const StTrack* track(UInt_t) const;
UInt_t entries(StTrackType) const;
StTrack* track(StTrackType, UInt_t = 0);
const StTrack* track(StTrackType, UInt_t = 0) const;
```

4.52 StTrackPidTraits

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTrackPidTraits();
StTrackPidTraits(StDetectorId);
StTrackPidTraits(const dst_dedx_st&);
```

Public Member Functions

```
Short_t detector() const;
```

4.53 StTrackTopologyMap

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTrackTopologyMap();
StTrackTopologyMap(ULong_t, ULong_t);
StTrackTopologyMap(const ULong_t*);
```

Public Member Functions

```
Bool_t primaryVertexUsed() const;
UInt_t numberOfHits(StDetectorId) const;
```

```
Bool_t hasHitInRow(StDetectorId det, UInt_t row) const;
```
Row numbering starts at 0.

```
Bool_t hasHitInSvtLayer(UInt_t layer) const;
```
Layer numbering starts at 0.

```
Bool_t turnAroundFlag() const;
ULong_t data(UInt_t 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).

4.54 StTrigger

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTrigger();
StTrigger(UShort_t aw, UShort_t w);
```

Public Member Functions

```
UShort_t triggerActionWord() const;
UShort_t triggerWord() const;
void setTriggerActionWord(UShort_t);
void setTriggerWord(UShort_t);
```

Public Member Operator

```
Int_t operator==(const StTrigger&) const;
Int_t operator!=(const StTrigger&) const;
```

4.55 StTriggerDetectorCollection

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StTriggerDetectorCollection();
StTriggerDetectorCollection(const dst_TrgDet_st&);
```

Public Member Functions

```
StCtbTriggerDetector& ctb();
const StCtbTriggerDetector& ctb() const;
StMwcTriggerDetector& mwc();
const StMwcTriggerDetector& mwc() const;
StVpdTriggerDetector& vpd();
const StVpdTriggerDetector& vpd() const;
StZdcTriggerDetector& zdc();
const StZdcTriggerDetector& zdc() const;
```

4.56 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;
UInt_t numberOfDaughters() const;
StTrack* daughter(StChargeSign sign);
const StTrack* daughter(StChargeSign sign) const;
StTrack* daughter(UInt_t);
const StTrack* daughter(UInt_t) const;
StPtrVecTrack daughters(StTrackFilter&);
void addDaughter(StTrack*);
void removeDaughter(StTrack*);
Float_t dcaDaughterToPrimaryVertex(StChargeSign sign) const;
Float_t dcaDaughters() const;
Float_t dcaParentToPrimaryVertex() const;
const StThreeVectorF& momentumOfDaughter(StChargeSign sign) const;
StThreeVectorF momentum() const;
void setDcaDaughterToPrimaryVertex(StChargeSign, Float_t);
void setMomentumOfDaughter(StChargeSign, const StThree-
VectorF&);
void setDcaDaughters(Float_t);
void setDcaParentToPrimaryVertex(Float_t);
```

4.57 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;
ULong_t flag() const;
Float_t chiSquared() const;
StMatrixF covariantMatrix() const;
StThreeVectorF positionError() const;
const StTrack* parent() const;
UInt_t numberOfDaughters() const = 0;
StTrack* daughter(UInt_t i) = 0;
const StTrack* daughter(UInt_t i) const = 0;
StPtrVecTrack daughters(StTrackFilter&) = 0;
void setFlag(ULong_t);
void setCovariantMatrix(Float_t[6]);
void setChiSquared(Float_t);
void setParent(StTrack*);
void addDaughter(StTrack*) = 0;
void removeDaughter(StTrack*) = 0;
```

Public Member Operator

```
Int_t operator==(const StVertex&) const;
Int_t operator!=(const StVertex&) const;
```

4.58 StVpdTriggerDetector

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StVpdTriggerDetector();
StVpdTriggerDetector(const dst_TrgDet_st&);
```

Public Member Functions

```
UInt_t numberOfVpdCounters() const;
Float_t adc(UInt_t) const;
Float_t time(UInt_t) const;
Float_t minimumTime(StBeamDirection) const;
Float_t vertexZ() const;
void setAdc(UInt_t, Float_t);
void setTime(UInt_t, Float_t);
void setMinimumTime(StBeamDirection, Float_t);
void setVertexZ(Float_t);
```

4.59 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;
UInt_t numberOfDaughters() const;
StTrack* daughter(UInt_t = 0);
const StTrack* daughter(UInt_t = 0) const;
StPtrVecTrack daughters(StTrackFilter&);
Float_t dcaBachelorToPrimaryVertex() const;
Float_t dcaV0ToPrimaryVertex() const;
Float_t dcaDaughters() const;
Float_t dcaParentToPrimaryVertex() const;
const StThreeVectorF& momentumOfBachelor() const;
StThreeVectorF momentumOfV0() const;
StThreeVectorF momentum() const;
StV0Vertex* v0Vertex() const;
StTrack* bachelor();
Double_t chargeOfBachelor();
void setDcaBachelorToPrimaryVertex(Float_t);
void setMomentumOfBachelor(const StThreeVectorF&);
void setDcaDaughters(Float_t);
void setDcaParentToPrimaryVertex(Float_t);
void setV0Vertex(StV0Vertex*);
void addDaughter(StTrack*);
void removeDaughter(StTrack*);
```

4.60 StZdcTriggerDetector

Summary

Synopsis

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

Description

Related Classes

Public Constructors

```
StZdcTriggerDetector();
StZdcTriggerDetector(const dst_TrgDet_st&);
```

Public Member Functions

```
UInt_t numberOfZdcCounters() const;
Float_t adc(UInt_t) const;
Float_t tdc(UInt_t) const;
Float_t adcSum(StBeamDirection) const;
Float_t adcSum() const;
void setAdc(UInt_t, Float_t);
void setTdc(UInt_t, Float_t);
void setAdcSum(StBeamDirection, Float_t);
void setAdcSum(Float_t);
```

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

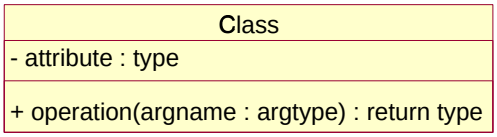

Figure A.1: The class icon in UML.

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 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

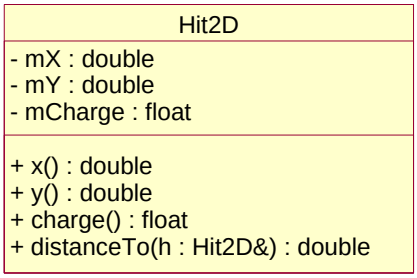

Figure A.2: Hit2D class. Attributes and operations are shown.

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.

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 {
```

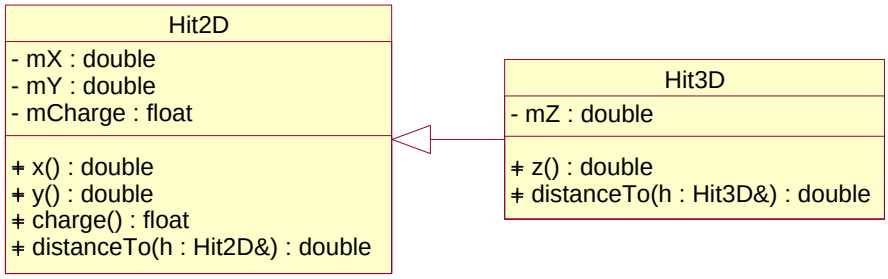

Figure A.5: Inheritance.

```
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

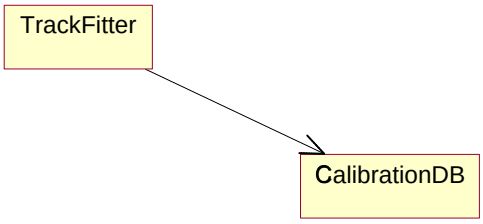

Figure A.8: Dependency.

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