CdfTrackSet Review Summary

Jim Kowalkowski
Marc Paterno

1 Introduction

This review covered the *CdfTrackSet* class and its associated classes, from the package **Tracking** package. We were concerned mostly with issues regarding the Event Data Model (EDM), and with the use of *CdfTrackSet* as an example of how other collections of objects in the *EventRecord* should be designed. We have looked at *CdfTrack* and the associated "physics" classes only insofar as their design reflects on the issues of the EDM. Our main concern was that the design integrate well with the new EDM design, and that the expected usage patterns were all supported with mechanism both "natural" and efficient. By "natural", we mean an interface that is consistent both within the EDM and with related concepts in the Standard Library. We viewed a natural interface as extremely important, because although this system may be designed and implemented by experts, it will be *used* by many who are not expert in OO software design or C++ programming.

Accompanying this document is a class diagram that illustrates many of the concepts outlined here. The class diagram shows several relevant use cases and the important relationship between EDM-supplied classes and the high-level track classes. Because of time constraints, this document does not go into detail about all aspects of the diagram. A follow-up meeting should be used to discuss the diagram.

2 Overview

In the current design of *CdfTrackSet*, it is clear that much effort has gone into producing a design that can be used as an example for other parts of the reconstruction system. The interface reflects the EDM related needs of the designers and users of a complex subsystem. Specific items we note are:

- Attention is given to the storage of "accounting" information, by which we mean information that describes how items in the event data were generated. The design has taken into account the difference between *CdfTrack*, a large object for which the overhead of storing accounting information is not excessive, and the various Hit classes, which are small object for which accounting overhead could be prohibitive.
- The concept of providing a view of data that is distinct from the actual container of the data is partially developed. Our design suggestions in this document expand upon this idea.
- Associations between related objects are partially developed. Again, in our design suggestions we expand upon this idea.
- The current design allows algorithms to operate given a view or a container. This was achieved by using a base class from which containers and views are derived. Effectively this base class acted as a view and the view subclass just exposed more parts of the base class interface.

We have attempted to extract from the design of *CdfTrackSet* and its accompanying classes a design that can be used by many parts of the reconstruction system.

2.1 Definitions

In this section, we present a few definitions for terms which we will use in later sections of this document.

Collection: A collection provides access to (*i.e.* iteration over) a group of elements. Collections can be either *containers* or *views*.

Container: A container is a collection that owns its elements, and is therefore responsible for managing their lifetimes.

View: A view is a collection that does not own its elements, and is therefore not responsible for managing their lifetimes.

The issues of persistence for both containers and views are somewhat complex, and are discussed in a later section of this document.

3 Major Concerns

Our major concern with *CdfTrackSet* and its associated classes is complexity. We believe that this complexity will cause difficulties for a less-expert designer trying to use *CdfTrackSet* as an example of the pattern to follow for the CDF Event Data Model. The designs we propose in this document try to manage this complexity, mainly through the use of class templates and function templates. While the implementation of the class templates is complex, the users (who are designers of other items in the Event Model) are not exposed to this complexity.

4 Design Recommendations

In this section we describe the design we recommend for collections in the CDF EDM, which includes but is not limited to *CdfTrackSet*. Our presentation is not complete in all details. Specifically, we have not dealt with the following details:

- The mechanism by which **ROOT** is used to perform input and output, and how this interacts with the *EventRecord*;
- Schema evolution, especially the effects of schema evolution on streamer methods for *Streamable* objects.

In addition, we present some classes which we do not recommend, but which might be necessary, depending upon decisions made about other classes. These classes are clearly indicated in the text.

4.1 Overview

We have divided the tasks performed by *CdfTrackSet* and its nested classes into several categories:

- Collection classes
- Link classes
- Accounting classes

In the following sections, and in the accompanying diagram, we describe the classes in each of these categories.

4.2 Description of Classes and Relationships

4.2.1 Collection Classes

We have identified two broad varieties of collection, and several subtypes for each of these. The two broad categories are *owning collections*, which we shall call *Containers*, and *non-owning collections*, which we shall call *Views*. It might be important to have all collection objects have, as a data member, the number of objects in the collection; this would allow the collection to carry some useful information even in the case in which all the contained (or pointed-to) objects are unavailable. All of these collection classes must provide for iteration over the collected elements. Because of the complexity inherent in providing a set of fully functional Standard Library iterators for each collection, we strongly recommend that the collection classes provide direct access to the underlying Standard Library collection upon which the *Containers* and *Views* are based. This means that the Standard Library iterators should be used directly in the views and containers. Of course, the appropriate `typedef` should be provided in each case, for uniformity and simplicity of interface.

4.2.1.1 Containers

A container holds objects, and is responsible for managing their lifetime. A container is also responsible for providing keyed access to its contents. In most cases, the key can be an unsigned integer, and the *Container* can be implemented using an STL vector of pointers. In some cases, when some special feature of the underlying object should be modeled in the software, a specialized *Container* is more appropriate. We have actually identified four different types of containers: the *Container*, the *InstanceContainer*, the *SpecialContainer*, and the *TempContainer* (referred to as the *PumpingContainer* during the review). The accompanying class diagram shows all four of these containers and a few examples of there use. An important property of many of these containers is that the EDM can automatically handle the conversion to/from the persistent store.

- As mentioned earlier, the standard *Container* will hold and own pointers to instances. As shown on the diagram, it is templated on the type of element that it holds and the STL container on which it is based. The default STL container could very well be vector. If *CdfTracks* were not storable objects, then they could live in one of these containers, which could be defined as: `typedef Container<CdfTrack> CdfTrackSet`. An advantage to using this type of container is that it may be able to support any type of track that is derived from *CdfTrack*. A user may be able to mix derived track classes and *CdfTracks* in the same container. In order to minimize performance problems associated with allocating and freeing many objects from the heap in a long running program, special object buffer pool or custom allocators may be needed. These should be supplied by the EDM, or by a package upon which the EDM depends.
- The *InstanceContainer* is similar to the standard *Container*, except that the *InstanceContainer* holds instances of the elements which it contains by value, rather than by reference. The collection of Hits may require use of this type of container. It has the advantage that all the elements are stored close by each other in memory. An example declaration of one of these containers could be `typedef InstanceContainer<SiHit> SiHitContainer`.
- The *SpecialContainer* is a stereotype of a completely custom container. Typically, this is required when the methods and internal organization of a regular STL container will not do. The creator of a special container is responsible for streaming out the individual elements of the container. The class *CalorData* is an example meeting the *SpecialContainer* sterotype.
- The *TempContainer* holds pointers to instances like the standard *Container*. This container, however, expects that all the elements stored in it are actually *Storable* objects. The implications of this are that as soon as the *TempContainer* is stored into the event, it actually adds all of its elements to the event. This container essentially becomes a view after being inserted into the event, referring to its elements using *Links*.

All types of *Containers* are *Storable*, which means they can be held in the *EventRecord*, and that each is issued (by the *EventRecord*) a unique ID. Users interact with *Containers* primarily by using one to create a *View*.

4.2.1.2 Views

A *View* is a collection that does not own its objects. *Views* allow iteration over their contents. The user of a *View* is not able to change the objects to which the *View* points. In contrast, a user is allowed to alter the *View* itself, for example by adding or removing objects, or by sorting the objects. This assumes, of course, that the user has a non-`const` view. We have identified three types of views: the *SimpleView*, the *CompoundView*, and the *SOView*.

- A *SimpleView* is a view composed of objects owned by a single *Container*. A *SimpleView* is efficient because it needs to keep only one *Link* (to refer to the *Container* from which it was made) and one index per contained object. It is less flexible than a *CompoundView*. In order to facilitate iteration through the elements, the *SimpleView* may need to also hold a vector of pointers to the objects in the index array. These pointers would not be persistent directly, but would be reconstituted when the view is streamed in — or perhaps the first time the view is accessed. An example of a *SimpleView* could be the *Hits* referred by a *CdfTrack*. This view allow a user class to reference a subset of the full container. If the *SimpleView* is actually held as a *Link* to a *SimpleView*, then the data in the *SimpleView* can be dropped from the event without any repercussions, other then the user not being able to traverse that link.
- A *CompoundView* is a view composed of objects owned by more than one *Container*. It is more flexible than a *SimpleView*, but it is less efficient, because it needs to keep an *IndexLink* for each contained object. An example of a compound view could be the *BestTracksView*, which presumable refers to tracks produced from several different algorithms and owned by several different containers.
- The *SOView* is a high-level view that really holds each object as a *Link* to the object. This implies that all the elements in a *SOView* are stored directly in the event. If *CdfTracks* are stored directly into the event, then this view could be used to reference them.

4.2.2 Link classes

Link classes are "smart pointers"; they are produced from class templates that provide for persistent associations. While in memory, each link class contains an actual C++ pointer, and provides the member selection operator (`operator->()`) to allow the calling of member functions in the pointed-to class. When written to persistent storage, the link writes the information necessary to the reconstitution of this pointer.

Link: A *Link* is a smart pointer to a specific type of *Storable* object. Because the *EventRecord* provides a unique identifier for each *Storable* object, the persistent form of a *Link* needs only to record the identifier for the pointed-to object, and an indication of the class of the pointed-to object. *Link* has one template parameter: the type of the pointed-to object (which is required to be a subclass of *Storable*).

IndexLink: An *IndexLink* is a smart pointer to an object within a *Container* (which is, by definition, not a *Storable* object; *Storable* objects are held directly by the *EventRecord*). *IndexLink* has three template parameters: the type of the *Collection* into which it points; the type of the object to which it points, and the type of the index used by the *Collection* to indicate which object is pointed to.

4.2.3 Accounting Classes

In preparing an example use of accounting information, we noticed that there is likely to be a difference between how a high-level physics object or collection is stored in the event and how accounting information is stored and referenced. It is conceivable that many objects in the event, directly or indirectly, will need to refer to the same accounting information. The example use case we have in the *CdfTrack* and *CdfTrackSet*. Here the *CdfTrackSet* creates an accounting object, fills it, and stores it into the event; the *CdfTrackSet* rreferences the accounting object using a *Link*. Each track must also reference this same accounting object. We created a class called *Accountable*, which is derived from *Storable*. This class forces the user to install accounting information. The *Accounting* class is derived directly from *Storable*. If the *Accountable* layer did not exist, then *Storable* would need to enforce the accounting rules. If this were the case, then the *Accountable* class would be forced to have accounting information, causing an infinite loop. The ability to store accounting information directly into the event and reference it through a link it an important factor in reducing the number of accounting objects in the event.

4.2.4 How They Work Together

The container classes, the view classes, and the link classes work together to provide the functionality for which *CdfTrackSet* and its associated classes demonstrate the need. The containers hold the event data; reconstruction modules are responsible for creating containers and inserting them into the *EventRecord*. Views provide the access that users of reconstruction output require. Views can be created, and their contents sorted, extended or trimmed, as needed. Links provide the mechanism by which the contents of views are associated with containers.

One thorny remaining problem is that of the persistence of views. What does it mean to write out a view? Consider, for example, a revised *CdfTrack* which contains a *SimpleView<Hit>*, indicating the hits used to form this track. If the decision is made to write out the *CdfTrack*, what is done with the view? The simplest option is to merely write out the persistent form of the *Link* connecting the *SimpleView<Hit>* with the *Container<Hit>* to which it points, and the indices of the *Hits* to which reference is made. When the resulting file is read back in, the *Hits* would only be available if the *Container<Hit>* was also written to the file.

A more flexible, but more complex, approach would be to allow only some of the *Hits* in the *Container<Hit>* to be written to persistent storage; specifically, it could be arranged to write out those *Hits* to which some view actually refers, and only when such a view is written. We will not develop this theme further in this document. We raise it in part to demonstrate the degree of flexibility which such a component design gives the system. If such a scheme were actually employed, there would need to be an option to store the entire hit container, not just the ones referred to.

4.2.5 The GenericView class

It may be useful to have the algorithm objects contained with reconstruction modules be independent of the classes of the Event Data Model, such as the *Containers* and *Views*. This would allow them to avoid compile-time dependence upon **ROOT** class *TObject*, with all the other entanglements that entails. It could also provide the buffer between classes the **ROOT** command line C++ interpreter **ROOTCINT** can understand, and the template code that the current version of **ROOTCINT** cannot understand. It is possible that the *GenericView* classes might be no more than a Standard Library vector of pointers-to-T.

```
typedef std::vector<Track*> TrackVec;
typedef Container<Track, TrackVec> CdfTrackSet;
typedef View<Track, TrackVec> CdfTrackView;
typedef IndexedView<Track, TrackVec > CdfIndexedTrackView;
typedef GenericView<Track> CdfAlgorithmView;

func(Event& evt) {
  EdmHandle<CdfTrackSet> tset = evt.find(..1..);
  EdmHandle<CdfTrackView> tview = evt.find(..2..);
  CdfAlgorithmView av1(tset.begin(),tset.end());
  CdfAlgotithmView av2(tset.begin(),tset,end());
  CdfIndexTrackView tv(tset);
  CdfAlgorithmView av3(tv.begin(), tv.end());
  doFittingStuff(av1);
  doFittingStuff(av2);
  doFittingStuff(av3);
}
```

Here the implementation of *GenericView* might possibly be:

```
template <class T> class GenericView {
public:
  typedef std::vector<T> Vec;
  typedef Vec::iterator iterator;
    ...

  // implies that U returns T pointers
  template<class U> GenericView(U first, U last) {
    copy(first,last,back_insert_iterator(vec);
  }

  iterator begin() { return vec.begin(); }
  iterator end() { return vec.begin(); }
private:
  vector<const T*> vec; // so that the objects are immutable
};
```

This implies that a generic view copies out the pointers to thing each time it is made. This is probably the case anyway with the current *CdfTrackSet*, *CdfTrackCollection* and *CdfTrackView* because the Set/Collection holds three lists of Track pointers and/or objects. In many cases, the algorithms themselves are CPU intensive, and so the copying involved in creating a *GenericView* is comparatively inexpensive.

4.3 Creation of Contained Objects

We are concerned with having pointers to tracks flying about; specifically, we are concerned about memory ownership being assigned at all times, so that memory leaks do not occur – even when a function returns prematurely, because of an error condition. It is important that the ownership of each object (each *Hit*, each *CdfTrack*) be clearly allocated, so that the appropriate destructor will automatically release the resources allocated during the creation of that object. For this reason, we recommend that items like *CdfTracks* and *Hits* be owned upon creation by the appropriate *Container*. It is important to note that such objects can be modified while in the *Container*; they only become unmodifiable in the current version of the Event Data Model when the *Container* is inserted into the *EventRecord*.

4.4 Interchangeability of Containers and Views

The current *CdfTrackSet* design contains a very powerful feature: users of the base class interface do not care whether what they have is a “container” or a “view”. The user can perform the same iterations and other operations regardless of the true type they manipulate. In some cases, however, this had the bad effect of warping the internal interface of the base class to allow for overriding by subclasses. We found this the least “natural” part of the design, and expect that this is the part of the system in which designers with less expertise would flounder.

We considered several methods that would retain a similar flexibility on the part of users of containers and views:

1. make an abstract container base class with abstract iterators and let the various views and containers inherit from it;
2. have each algorithms templated on the type of container it takes;
3. take advantage of some common structure of view and containers and put that into the base class - basically the view is the base class;
4. have the algorithms make use of the iterator `typedefs` provided by the collections, which are uniformly named, and are really just Standard Library iterators;
5. have the algorithms use a simple class like *GenericView*, and support conversion from *Containers* and *Views* to *GenericViews*.

Choice (1) seems unwieldy to maintain and support, in part because of the complexity involved in implementing the necessary iterators, which has been described in Section . The use of virtual base classes for each iterator would also probably lead to inefficient performance when the container and view classes were actually used polymorphically. Choice (2), while powerful, would lead to very dense template code, which would probably only be understood by experts. Choice (3) is the tack taken in the current design of *CdfTrackSet*. As we noted above, this choice leads to significant complexity in the design, and some less-than-attractive implementation details.

We recommend choice (4) or (5), or both; they could be used in combination.

4.5 The Smart Iterator

The current **Tracking** package has the concept of smart iterators. The smart iterator is an iterator that takes a predicate and skips elements in the collection rejected by the predicate. We see very little practical need for such an iterator to be the standard iterator supplied by the containers and views. The containers and views should provide iterators that their corresponding STL containers provide. One of the uses of the smart iterator is to fill a view. We strongly recommend using the STL algorithms to do this, or the creation of functions that are built up from the STL components (*copy_if* and *back_insert_iterator* for example). If the smart iterators are to be supported, then there should be a single smart iterator class that the EDM provides, external to the collections which it iterates over. This iterator class could be constructed with a container type and a predicate. The use of this iterator is special; it really only allows for forward-style iteration over a collection. A method *done* could indicate that the end of the collection has been reached.

5 Coding Recommendations

5.1 Redundant methods

Redundant methods should be removed.

- push_back, pushBack, append

Caveat: for STL-like code, stick with STL conventions, in contrast to CDF conventions. We will be requesting to have the "Guidelines" modified to reflect this.

5.2 Iterators

The iterator classes of the Standard Library are an essential part of the "generic programming" aspect of the STL. They are what allow the algorithms to be written in a manner that makes them independent of the container classes. We would like to have the power of the algorithms of the Standard Library available for use with the collections in the CDF EDM.

Making an iterator class compatible with the algorithms of the Standard Library is a nontrivial task, involving inheritance from the correct classes, and implementation of the appropriate *iterator_traits* classes. Because this is not simple, we recommend using iterators provided by the Standard Library whenever possible. Custom iterators should only be implemented for special cases, and in those cases, the implementation should remain as simple as possible (for example, implement only a forward iterator, if that suffices).

5.3 Inappropriate `const`-ness

The `const` sorting method is not appropriate. A `const` method should not modify the object on which it is called. If the *CdfTrackSet* were really an unordered set, then it should not have a member function called *sort*. If the *CdfTrackSet* is an ordered collection, then a *sort* function which changes the ordering of the elements should not be declared `const`. A clear indication of this is the warning in the comment of the *sort* function: calling this function invalidates all iterators into the *CdfTrackSet*. Calling a `const` function on a collection of the Standard Library cannot have this effect; the collection of the Event Data Model should behave similarly.

Sorting is provided for in our suggested model by the creation of a non-`const` view, and then the calling of a *sort* function on that view. The collections and view should not have a `const` sorting function, and their data members should not be declared `mutable`.

5.4 Style

5.4.1 Obsolete declaration style

Many functions use an old style of declaring a function with no arguments. Newcomers to C++ may find this confusing, because it is not seen in the major texts (*c.f.* The C++ Programming Language, 3rd edition, Strouvstrup). The extraneous `void` should be removed: `func(void)` $\longrightarrow$ `func( )`.

5.4.2 Inherited function documentation

While the documentation of the **Tracking** package is, in general, excellent, we think it is counterproductive to include in a subclass a commented-out version of the base class's functions. It is too easy for this to get out of synchronization; to be sure of the correctness, a reader would still have to go to the header of the base class. We recommend removal of all "inherited function" documentation of this sort.

5.5 CdfTrack

We suggest the introduction of a class *CdfTrackInfo* that includes the "accounting" information in the current in the current *CdfTrack* class. This would allow removal of many of the nested classes defined within *CdfTrack*.

If *CdfTrack* does inherit from *StorableObject*, we recommend that the ID manager of the *EventRecord* be used to assign track IDs. If *CdfTrack* does not inherit from *Storable Object* (which is what we prefer), then *CdfTrack* instances do not need individual IDs. In either case, a special ID manager for *CdfTrack* is not needed.

5.6 Miscellanea

- Use constructors to initialize values (especially "special" values, such as a value indicating an invalid state), rather than making the outside world aware of the "special" value.
- The predicate classes in *CdfTrack* need to be changed from nested classes to regular classes inside a namespace. We recommend getting rid of the abstract base class comparison class; it appears to be needed only for sorting.
- Reduce the number of custom iterators to a bare minimum; use them only for special cases. Reuse the STL iterators using `typedefs` wherever possible. Get users accustomed to using **iter->method( )* for reference collections and **iter.method ()* for value collection objects when using iterators. In other words, use *vector<T*>* iterators as they are and do not attempt to write special iterators that make it look like *vector<T>*.

6 Physical Design

The implementation of the various link classes must not force any compile time or link time dependencies on the class to which they point. In other words, definition of the link classes should cause name-only binding to the classes to which they refer. Only if a user chooses to write code that traverses the link should that user be dependent on the classes and library to which the link refers. If a user of *CdfTrack* is not interested in looking at a *Hits*, then the user's code should not need to link in the library that contains the *Hit* code.

An *IndexLink* implies that if the link is traversed, then the code will depend on the container to which the *IndexLink* points, and thus on the class to which the *IndexLink* refers.

7 Documentation

The documentation present in the header files of the various CdfTrack classes is extremely useful for discovering what purpose the class serves and what the methods do. We would like to see all the code we walk through have documentation as good as this.

8 Other Notes

8.1 Class Hierarchies

Several time we have heard from collaborators who are reluctant to deal with deep inheritance hierarchies or complex template code. What is needed is an implementation that is powerful enough to make its use simple, even if the implementation itself is not simple. The classes of the Standard Library are often quite complex, but the simplicity of their use hides this complexity very well. This is the design we should strive to emulate.

8.2 Modifiable Objects in the Event

The need for modifiable objects in the event was brought up at the review. This was also brought up by the RRL3 group. Several example scenarios were given as to where and why this would be useful. One possible way to provide this functionality was discussed: optionally allow objects to be inserted into the event unlocked, an unlocked object cannot be pulled out of the event in a read-only form. When the object to ready for normal use (done being massaged and manipulated by a series of modules), it is locked. At the point it is locked, it becomes available for read-only use, which means that users downstream can safely referred to elements within it. An unlocked object in the event cannot be safely referred to in any way.

It might be worthwhile to have many classes of objects be manipulated at all times through reference counted pointers, to help avoid resource leaks.