Physics Object Review

Craig Blocker
Jim Kowalkowski
Marc Paterno
Brian Winer

July 26, 2000

Abstract

We have reviewed the offline high level objects (*CdfJet*, *CdfEmObject*, *CdfTrack*, and *CdfMuon*) and their collection classes. This note gives our recommendations in several areas.

1 Introduction

In order for physicists to effectively analyze Run II data, it is important that the "physics objects" (*CdfJet*, *CdfEmObject*, *CdfTrack*, and *CdfMuon*) and their collection classes be well designed. It is essential that the user with an understanding of the detector, the physics she or he wishes to do, and the basics of C++ be able to quickly and easily access the needed reconstructed information. We have reviewed these objects with this and other criteria in mind (such as maintainability of the code and offline performance). This note contains our recommendations.

We were disappointed that we did not receive the information, summarizing the methods of each of the physics objects, which we requested from each of the groups. This information might have helped us produce a more complete review.

2 Physics Objects

The EET group has defined a set of what are known as High Level Objects. They differ from the objects we reviewed in that they contain many kinds of information from different sources, that is, they contain just about everything that the physicist would like to know about the event. We feel that there is a need for the level of object we reviewed (*CdfJet*, *CdfEmObject*, *CdfTrack*, and *CdfMuon* — we shall call them "physics objects" to distinguish them) in addition to the EET group's High Level Objects.

The physics objects contain direct information on the reconstructed physical objects in the event. The High Level Object contain a great deal of related information (such as trigger information, different fits, *etc*). This potentially leads to much physical coupling[1] and large stored objects. We briefly looked through the *HighLevelObjects* package. It appears to violate most of the rules established by the Offline software group. Development of these packages should be stopped, and the effort channeled towards the physics objects.

We do not consider it part of this review to cover these High Level Objects and, instead, focus on the physics objects.

It is difficult to review the design of a set of classes if the purpose for those classes is not clearly defined. It seems that not everyone agreed on the purpose of these physics objects. For the purpose of this review, we have accepted the following defined purpose:

> A "physics object" is an EDM object that is useful for performing analysis, even in the circumstance when other related EDM objects have been deleted. These objects are suitable for inclusion in secondary or tertiary data sets, and are complete enough for most analysis purposes. They may contain links to other EDM objects, but must retain significant usefulness even if the linked-to objects have been dropped from the *EventRecord*.

According to this definition, *CdfTrack* is a proper physics object, while *CdfJet* is a class of a different sort. *CdfJet* is designed to be compact in the case when the output of many jet algorithms are stored in one *EventRecord*. This is probably suitable for secondary data sets, but is less so for tertiary data sets. **We recommend the introduction of another class (we leave the naming to the jet group) with the same interface, but which caches the results necessary for implementing most of that interface.**

3 Uniformity and Ease of Access

It is vital the physicist user of these objects be able to simply, clearly, and efficiently access the information needed to do analysis. This implies both that the access methods to these objects be straight-forward and that the interfaces be as uniform as possible across the various physics object classes.

It is clear that the four groups have already cooperated well on establishing a uniform interface. However, there are some differences that remain that need to be addressed.

3.1 Finding Objects in the *EventRecord*

In order to simplify access to the stored collection objects, each collection class has a static `find` method. One problem is that there are differences in the var-

[1] For a description of physical coupling, and the reasons for our concern about minimizing it, see http://cdspecialproj.fnal.gov/examples/PhysicalCoupling.html.

ious `find` methods. A second is that these many functions introduce a maintenance burden; improvements in the EDM will only be realized if all of these functions are modified.

CdfTrack, while it is a storable object, has no `find` method to assist the user; if one wishes to look up an individual track, one must make use of the *EventRecord*'s iterator classes. No `find` method was supplied for *CdfTrack* because users are intended to look for *CdfTrackColl*, rather than directly for individual *CdfTrack*s. We agree with this decision.

Class *CdfTrackColl* has methods:

```
static CdfTrackColl::Error find(CdfTrackColl_ch&);
static CdfTrackColl::Error find(CdfTrackColl_ch&,
                                const StorableObject::Selector&);
```

Class *CdfEmObjectColl* has methods:

```
static CdfEmObjectColl::Error
       find(CdfEmObjectColl::const_handle&);
static CdfEmObjectColl::Error
       find(CdfEmObjectColl::const_handle&, const std::string&);
```

Class *CdfJetColl* has methods:

```
static CdfJetColl::Error find(EventRecord*, CdfJetColl_ch& ,
                              const std::string&);
static CdfJetColl::Error find(CdfJetColl_ch&,
                              const std::string&);
static CdfJetColl::Error find(EventRecord*, CdfJetColl_ch& , Id);
static CdfJetColl::Error find(CdfJetColl_ch&);
```

Class *CdfMet* has methods:

```
static CdfMet::Error find(CdfMet_ch&);
static CdfMet::Error find(CdfMet_ch&,
                          const std::string&);
```

Class *CdfMuonColl* has methods:

```
static CdfMuonColl::Error find(CdfMuonColl::const_handle&);
static CdfMuonColl::Error find(CdfMuonColl::const_handle&,
                               const std::string&);
```

Each of these classes uses a return type of `Error`, which is a `typedef` within each class. However, the meanings of these `typedef`s differ. In some cases (*CdfEmObjectColl* and *CdfMuonColl*), the `typedef` is to `Bool_t`, which is itself a `typedef` provided by **ROOT**. **As a general note, we strongly recommend removing this dependence on ROOT. ROOT**'s `Bool_t` should everywhere in the CDF code be replaced by the standard built-in C++ type `bool`. In other cases, (*CdfJetColl* and *CdfTrackColl*), the `typedef` is an `enum` values with **OK** and **ERROR** (and they even differ on the order of the two values, although this probably wouldn't be noticed by the user). The return types should be uniform.

Also, the `find` methods differ in what additional information is needed, what form it takes, and whether a pointer to the event record is needed. More uniformity is essential here.

A possible solution to insuring uniformity of access methods to stored objects would be to make the `find` method part of the EDM. We propose the addition of several template member functions to the *EventRecord* class. The signatures could be something like the following suggestions.[2] The list of functions in our example is drawn from what we found in the existing code. The package authors should reach an agreement on which functions should be standard (including, perhaps, ones we did not think of).

Note that the template parameter SEQUENCE, as used in this example, is a back-insertion sequence. It can be any collection which understands the function `push_back`. In the Standard Library, the classes *vector*, *list*, and *deque* are all back-insertion sequences.

```
// Fill the given sequence with handles for all the
// StorableObjects of type SEQUENCE::value_type::value_type
// in this  EventRecord. Return the number of handles in the
// collection.
template <typename SEQUENCE>
size_t
findAll(SEQUENCE& results) const;


// Fill the given sequence with handles for all the
// StorableObjects of type SEQUENCE::value_type::value_type
// in this EventRecord that satisfy the given predicate. Return
// the number of handles in the collection.
template <typename SEQUENCE, typename PREDICATE>
size_t
findAll(SEQUENCE& results, const PREDICATE& pred) const;
```

[2] We have expressed these as template member functions of the class *EventRecord*; they could also be expressed as free function templates, making use of the existing *EventRecord* interface. Since both of these solutions would be an extension of the interface of *EventRecord*, not a change in an existing part of the interface, neither should break existing code. Marc and Jim will work with Robert Kennedy (the primary author of the EDM) to determine the exact signature and implementation. These functions would require minor additions to the handle classes, as described in §9.7.3.

```
// Get a single instance of the type of object specified by the
// given handle. Return true, and set the handle to contain the
// found object, if only one matching object is found. Return
// true, and set the handle to NULL, if no matching object is
// found. Return false, and set the handle to NULL, if more than
// one matching object is found.
// Note that use of findAll() is preferred.
template <typename HANDLE>
bool
findOne(HANDLE& h) const;

// Get a single instance of the type of object specified by the
// given handle, and matching the given predicate. Return true,
// and set the handle to contain the found object, if only one
// matching object is found. Return true, and set the handle to
// NULL, if no matching object is found. Return false, and set
// the handle to NULL, if more than one matching object is found.
// Note that use of findAll() is preferred.
template <typename HANDLE, typename PREDICATE>
bool
findOne(HANDLE& h, const PREDICATE& pred) const;

// Get a single instance of the type of object specified by the
// given handle, and identified by the given string. Return true,
// and set the hande to contain the found object, if only one
// matching object is found. Return true, and set the handle to
// NULL, if no matching object is found. Return false, and set
// the handle to NULL, if more than one matching object is found.
// Note that the findOne() that takes a predicate can fulfill
// this need as well.
template <typename HANDLE>
bool
findOne(HANDLE& h, const std::string& description) const;
```

3.2 Objects within Collections

The largest problem here has to do with the types of objects that are stored in the collections. *CdfTrack* and *CdfMuon* are storable objects and their collections contain links to them. *CdfJet* and *CdfEmObject* are streamable objects and the collections contain the actual objects. We believe that using streamable objects makes the code simpler and easier to maintain. In addition, collections own the streamable objects they contain, but once a collection of storable objects is stored in the event, the ownership of the collected storable objects is given to the event. This more complicated pattern of ownership is confusing for many users, and even some of the experts agreed it is easy to cause memory errors within the event.

Thus, we recommend that the physics objects be streamable. *CdfJet* and *CdfEmObject* are already streamable. Since *CdfMuon* is essentially unwritten at this point, it should be straight-forward to make it streamable. **For *CdfTrack*, we recommend that the tracking group look closely at the manpower, time, and consequences of converting to streamable objects.** If there is functionality of storable objects that are needed for *CdfTrack* objects, such as ID assignment, then we suggest that thought be given to also providing this functionality in streamable objects.

To consider what would be necessary if *CdfTrack* were converted to a streamable object, we discussed a few scenarios that would require additional EDM tools. These scenarios are also relevant whether *CdfTrack* remains a storable object, or if it is converted to a streamable object.

One scenario is the analysis of an event that has a track collection with 100 tracks and two electrons. On output we want to store the two electrons and their associated tracks, but not the remaining tracks. With the current classes, it is not clear how this is to be done. We consider it important for it to be easy for a user to do this.

In the current storable version of *CdfTrack*, dropping the track collection can be done without disturbing the individual tracks (we hope this is true), and the two we want will need to be marked to be saved while the rest are dropped. In the streamable case, we would need to copy the two tracks into a new track collection and add it to the event, then we would need to copy the electron collection, point the track links to the new track collection, and put it into the event. In either case, the process must be simple to use.

3.3 View Classes

CdfTrack and *CdfMuon* have associated collection objects known as Views (for example, *CdfTrackView*). These collections are derived from *StorableObject* and contain links to the physics objects. These behave basically as if they were collections of pointers to `const` objects — the objects may be queried, but not modified, through these pointers.[3] This has the advantage that the links in the View can be manipulated, that is, sorted, selected to not include all the objects, subtracted from and added to. The *CdfJet* documentation has an example of doing this by copying the stored *CdfJetColl* to a new collection. **We recommend this example be replaced by an example using Views. We recommend that all collections of physics objects be able to provide View objects.**

We recommend that analysis code and algorithm objects should prefer the use of Views to direct use of the collections.

[3]Formally, only member functions that have been declared `const` may be called on these objects. See also §8.

3.4 Standard `typedef`s and methods

Having methods on the collections that perform the function of `reserve` in *vector* is very important. While the `contents` method allows the user to call `reserve` directly on the contained collection, having `reserve` in the interface of the collection classes themselves may make it more obvious that `reserve` should be used. Giving count estimates to *vector*s, for example, can dramatically increase its speed of filling (`push_back`) under many situations. See §9.4.1 for more on this subject.

Creating a policy for standard `typedef`s in collections and objects is important. Choosing ones, whenever possible, that are similar to ones from the Standard Library is beneficial (*value_type*, *iterator*, *const_iterator*, *etc*). Any of the collection objects that have `typedef`s that will never be used (*iterator*, *const_iterator*, *etc.*) should be cleaned up and have these `typedef`s removed.

Using common methods names, such as `size`, `begin`, and `end`, are similarly important. As with the `typedef`s, it is critical to have them be common across the CDF classes, and important to have them be the same as the names in the Standard Library.

3.5 Association Between Objects

A forward-pointer is a pointer that points from an object created *earlier* in time to one created *later* in time. Setting such a pointer in an object that has already been added to the *EventRecord* requires casting away the `const`ness of the earlier object, and may cause a violation of the integrity of the event model. **We strongly recommend that any setting of forward pointers should be forbidden, and the existing occurrences should be removed from the code.**

Instead of inventing a way to deal with forward-pointers, such as a pointer in a *CdfTrack* to a *CdfMuon*, we recommend using association objects. That is, a simple collection that holds (*Link*<*CdfTrack*>, *Link*<*CdfMuon*>) pairs could be used to represent the associations. In addition to the benefit of not creating the opportunity for corrupting the integrity of the *EventRecord*, the use of an association class also allow the opportunity to associate multiple items together (a triplet rather than a pair), and also allows more than one instance of an associate (for example, different reconstruction algorithms may want to associate different instances of *CdfMuon* with the same *CdfTrack*).

Pointers from later objects to earlier objects (generally used to associate with the later object those objects used in its creation) cause no trouble and do not need the association class; for example, it is fine for *CdfMuon* to contain a *Link*<*CdfTrack*> which denotes the *CdfTrack* used in creating the muon.

4 Documentation

4.1 Documentation for Physics Objects

Documentation is extremely important because of the complexity of the physics objects and the EDM rules. It must be clear so that the user who understands the physics they wish to do and the basics of C++ can access and use the objects containing the information they need.

It is also important that the user know where to find the relevant, up-to-date documentation. **We strongly recommend that each offline package be required to have a */doc* directory.** The advantage of this is that as the code evolves, the documentation can be kept current and the correspondence between the version of the code and the version of the documentation can be maintained.

This documentation should be viewable by the CDF code browser (that is, text, HTML, PDF, or PostScript files). Of course, this documentation can refer to documentation in other locations (CDF notes, other web pages, etc.), but the need to maintain accurate documentation across offline versions should be kept in mind.

We strongly recommend that CDF establish (SRT) rules for building the documentation, and for installing the documentation into the release's */doc* directory. Having an installed release */doc* directory has the advantage of allowing access to the documentation in a well-defined place, without requiring going through the code browser. This may require enhancement of SRT by Jim Amundson.

We recommend that each package contain an example directory. This directory must be populated with a current, working AC++ module that illustrates how to use the classes in a meaningful way.

We recommend that the documentation for each of the physics objects include a UML class diagram, showing the relationships between the classes that are important to each physics object. This would include the inheritance hierarchy for each class, and each class to which links are held.

The tracking group is to be applauded for having significant documentation, consisting of a long (42 pages) CDF note on *CdfTrack* and its related classes. This note contains much detail, which is perhaps overwhelming to the user. **We recommend (as the tracking group suggested) a shorter summary that covers the important points for the user.**

The calorimetry group is also to be applauded for its documentation on both calorimetry and on jets. This documentation is in the form of a web page in a somewhat obscure location to the user. This needs to be remedied by either moving the html to the */doc* area or by putting an appropriate link there. Also, detailed documentation on the accessor methods needs to be added.

An upcoming CDF note on EM objects was promised.

There was no documentation on *CdfMuon*. This obviously must be remedied.

In summary, we recommend for each physics object, the following documentation:

- a */doc* directory in the package;
- an up-to-date user's guide in the */doc* directory;
- a UML diagram illustrating the relations between the important classes;
- an example directory, containing examples of how to use the important classes.

4.2 Documentation for the EDM

We recommend the production of a general document that explains the major concepts and classes of the EDM. Such a document should describe the purpose of each of the following classes and class categories. tasks:

- collections;
- views;
- handles (of all varieties); especially, distinguish between the different types of handles, and when the user should use each.

This document should also simply explain (with examples) how to do the following tasks:

- How does one select a subsample of objects to make a new collection?
- How does one include or exclude a collection from and output file?

This document should not contain information specific to any one kind of physics object; it would contain the information common to all the different physics objects.

5 Comments on Each Object

In these section, we address a range of issues for each of the physics object classes, and their related classes. We should stress that the coding recommendations made in one section often also apply to the other objects. We've generally placed the comments in the section which we thought needed them most.

5.1 Tracks

As a general comment, we find many `ifdefs` for `USE_CDFEDM2` in the tracking code. This makes the code difficult to read, and thus greatly increases the maintenance burden imposed on the code authors. We suspect these are no longer necessary; if so, they should be removed.

5.1.1 *CdfTrackColl*

The comments in the header for the class *CdfTrackColl* declare the copy constructor private, and state that it is not implemented, but the source file for the class does implement it. The implementation prints an error message to `cerr`, and then calls `assert(0)`, causing the program to crash.

If it is necessary to prevent the copying of *CdfTrackColl*, then the better solution is to declare the copy constructor private, and to leave it unimplemented — thus making code the tries to copy a *CdfTrackColl* produce a compile-time error, rather then letting it build an executable that reveals the error only through occasional crashes.

But the more important question is: why should it be illegal to copy one of these collections? It seems to us that this would be a natural thing to want to do, and we do not see any danger involved.

There are methods such as `_setHitSets` and `_setMCHMatches` that use COT singletons and cast away the `const`-ness of the hit collections. Do COT hits come from a singleton? If so, this must be changed so that they are stored using the EDM, and are accessed in the normal fashion.

Why are destroy, deallocate, activate, and deactivate present in this class? The comments of §9.7.2 apply for this class, and for others in the TrackingObjects package.

The `assignId` uses `const_cast` to remove the `const`-ness of *IdManager* object in the event. This should not be necessary. It is not clear whether the error is in the EDM (*i.e.* the *IdManager* or *EventRecord* needs to be changed), or if the error is in the style of use in *CdfTrackColl*.

The code authors commented that the Standard Library `typedef`s defined in this class are not used elsewhere. If this is true, they should be removed[4].

5.1.2 *CdfTrack*

Inlining simple methods like `CdfTrack::_setFitStatus` when possible is good to do.

Each `setChild` does a `const_cast`; see comments in §8 concerning the excessive use of `const_cast`. **The member datum `_child` is also an example of a forward pointer; we strongly recommend against their use; see §3.5 for our explanation why, and our proposed solution.** Specifically, in this case, we understand (from conversations with Ken Bloom) that the restriction to zero or one children could be a problem; two different could be run, generating two children. While this problem hasn't yet been encountered, it is likely to in the future. The argument was made that, in setting this data member, "we are not changing anything fundamental about the track." But clearly some other code may have made a decision based upon the information about this data member in this track; if the data member is then altered, that other piece of code is not reproducible. This example illustrates why we strongly recommend prohibiting such forward pointers.

[4] See also the comments in §9.7.3.

We find many unprotected accesses to elements of arrays, such as in the functions `setResidual` and `covAx`. This sort of code is a primary source of memory corruption. It would be better to make use of appropriate classes for each circumstance.

Large functions, such as `beginSIHits`, should not be inlined; see also §5.1.8.

In `innerCTHitR`, these is a constant -999. What is the significance of this value and its origin? In general, “magic numbers” like this should be avoided, and replaced with meaningful error conditions. Their existence is frequently a sign of inadequate planning.

5.1.3 *CT_HitSet*

It appears that this class is both a storable object and a singleton. This does not appear to be a sensible combination; singletons are designed for global access, and have a policy for lifetime management that generally calls for destruction during static destruction time, while *StorableObject*s are accessed through the *EventRecord* interface, and have their lifetime controlled by the *EventRecord*. This cannot possibly work properly. It is not reasonable to expect a user to figure out how to use this class. **We strongly recommend removal of the singleton nature of this class.**

The copy and assignment operators for this class are forbidden. We do not understand why this is necessary. If it is not necessary, they should be provided.

5.1.4 *CdfTrackView*

We recommend that the static methods `allTracks` and `defTracks` be eliminated in favor of the standard access methods proposed in §3.1.

Beware that static strings class members should not be used until after `main` is entered.

Is the use of `AbsEnv::instanace()->theEvent()` really necessary?

5.1.5 Pairs of Tracks

Marc and Jim each have slightly different ways to manage iterating through pairs of tracks in a clean, simple way. We will try to make these publicly available very soon.

5.1.6 Use of bi-directional links

We strongly discourage the use of bi-directional links, and recommend instead the use of association objects, as described in §3.5.

5.1.7 Excessive use of `const_cast`

The code in the package TrackingObjects contains an excessive number of uses of `const_cast`.

5.1.8 Inappropriate Inlining

We observe frequent inlining of methods that are long and that have looping constructs. This is not likely to improve performance and it is more than likely to cause code bloat, which can lead to poor performance — at compile time, link time and run time.

5.1.9 A Plethora of Collections

It is unclear when one is supposed to use a *CdfTrackCollection* instead of a *CdfTrackColl* and when a *CdfTrackSetView* is important as opposed to a *CdfTrackView* — and when to use a *CdfTrackSet*. Do all these classes serve distinct purposes? Any unnecessary ones should be removed. A short document that compare the different purposes of the remaining classes should be provided to guide the user to the correct choice for his purpose.

5.1.10 Enumerations

Enumerations in the global namespace such as those in *FitStatus.hh* should be avoided unless prefixed with the package identifier, or confined to a class or namespace scope. It is very easy to clash on names such as "stale" and "ok", both of which appear in *FitStatus.hh*.

5.2 Jets

5.2.1 Linkage to *CalData*

One item that generated much discussion is whether *CdfJet* should always require *CalData* to be present, or if it should instead be "self-contained". **In the PAD, we recommend the use of *CdfJet* as it currently is — compact, but requiring the existence of *CalData*.** In more-derived datasets, one should be able to drop the *CalData* object and still have useful jet. There might be a simple way to achieve this by adding an interface (abstract) class (we call it *BasicJet*), and another jet class that is self-contained (we call it *SelfContainedJet*). We happily leave the choice of the real class names to the authors of the classes. We sketch our proposal below.

```
class BasicJet
{
  virtual HepLorentzVector fourMomentum() const = 0;
  ... etc, for the rest of the required interface
};

class CdfJet : public BasicJet
{
  ... as it is today ...
};
```

```
class SelfContainedJet : public BasicJet
{
... important quantities contained directly ...
};
```

We would then have two jet collection types — one to store *SelfContainedJets*, and one to store *CdfJets*.

```
class CdfJetColl : public StorableObject
{
  ... as it is today, plus additions ...
  typedef ViewVector<CdfJet>   CdfJetView;
  typedef ViewVector<BasicJet> BasicJetView;
  size_t fillView(ViewVector<CdfJet>& fill_me) const;
  size_t fillView(ViewVector<BasicJet>& fill_me) const;
};

class CdfSelfContainedJetColl : public StorableObject
{
  ... everything necessary ...
  typedef ViewVector<SelfContainedJet> SelfContainedJetView;
  typedef ViewVector<BasicJet>         BasicJetView;
  size_t fillView(ViewVector<SelfContainedJet>& fill_me) const;
  size_t fillView(ViewVector<BasicJet>& fill me) const;
};
```

5.2.2 Standard `typedef`s

If the following `typedef`s in *CdfJetColl* are not used, they should be deleted. (We heard some comments that they were not used):

- `typedef CdfJet value_type`
- `typedef std::vector<CdfJet>::iterator iterator`
- `typedef std::vector<CdfJet>::const_iterator const_iterator;`
- `typedef CdfJet& reference;`
- `typedef const CdfJet& const_reference;`
- `typedef CdfJet* pointer;`
- `typedef const CdfJet* const_pointer;`
- `typedef std::vector<CdfJet>::difference_type difference_type;`
- `typedef std::vector<CdfJet>::size_type size_type;`

We recommend the `typedef CollType` be changed. The current version reads:

```
typedef std::vector<value_type> CollType;
typedef CollType collection_type;
```

The use of *vector* here assumes knowledge of the type used in the *ValueVector* class. This sort of code is fragile. **We recommend instead the use of a standard `typedef` in the collection class, as we propose in §9.7.3.**

5.3 EMObjects

5.3.1 Complex Calculations

The file *CdfEmObject.cc* is about 1400 lines long, and *EmCluster.cc* is about 1800 lines long. We are concerned that these classes are doing too much. We note the following items contained in these classes:

- A Gaussian integration algorithm implementation in *CdfEmObject.cc*.
- The large `EmCluster::checkDistance` algorithm implementation.
- The large `EmCluster::build`.
- The large `CdfEmObject::maxPtTrackLocalCoord` function.
- The large `CdfEmObject::bestMatchingCesCluster` function.
- The large `CdfEmObject::bestMatchingPes2dCluster` function.
- The large `CdfEmObject::lshr` function.

All of these seem more like algorithms or utilities than code that should live outside the "data objects" *EmCluster* and *CdfEmObject*.

Since these algorithms will almost certainly evolve, leaving them inside the classes will make it necessary for the code version to be carefully matched with the data read, so that old data uses the old algorithm versions, and new data uses the new algorithm versions. The complexity of dealing with this is reduced (but not eliminated) by having the algorithms brought out into their own classes[5].

We did not have time to look into this, but we wonder if the use of `postread` in *CdfEmObject.cc* causes all the linked objects to be brought in and converted to C++ objects? In *CdfEmObject.hh* we see:

```
ValueVectorLink<EmClusterColl> _ilEmCluster;  //Indexed link
EmCluster Link<CdfTrackView> _matchingTracks; //Link to tracks
IndexViewVector<CesClusterColl> _matchingCesClusters;
IndexViewVector<Pes2dClusterColl> _matchingPes2dClusters;
```

In *CdfEmObject.cc* we see:

```
bool cesread   = _matchingCesClusters.postread( p );
bool pes2dread = _matchingPes2dClusters.postread( p );
bool trackread = _matchingTracks.postread( p );
```

[5] See also §9.2.

5.3.2 Other Comments

We recommend that the EM objects follow a similar pattern as that outlined in the §5.2 regarding the use of Views. We stress encouraging the use of Views by adding `fillView` methods and providing (or correcting) `typedefs`. Please see also the comments in §9.2.

5.4 Muons

The muon code is well behind the other objects. The *CdfMuon* interface has not been clearly defined, and little code is written. The new group needs to quickly produce a viable *CdfMuon* class; this should be made somewhat easier by borrowing the most effective techniques from the other physics objects.

We recommend that, like *CdfJet* and *CdfEmObject*, *CdfMuon* should be a streamable object. It should contain a link to the associated track (or a collection of links to tracks, if it is possible to be associated with more than one). **We strongly recommend that *CdfMuon* not contain a host of methods that return components of contained quantities.** Instead, such components should be grouped into meaningful classes, and the *CdfMuon* should then have functions which return `const` references to these contained items.

The same recommendation we make for the other physics objects should also be applied to *CdfMuon*.

6 Links

6.1 Data Caching vs. Use of Links

One issue for all four types of objects is the question of which information should be cached in the object (that is, exist as a private data member) and which information should be calculable from links to other objects[6]. The issues here are efficiency of access, size of the stored object, and being able to drop sizeable lower levels objects in forming mini data sets from PAD's.

CdfTrack objects cache most of the track information. This is appropriate in this case since the lower level objects are primarily hits and recalculating the track parameters from hits and keeping the hits around for this would be highly inefficient.

CdfEmObject objects are primarily links to *EmCluster* and *CdfTrack* objects. Most of the information needed by the users is cached in these lower levels objects. Given the relative small number of EM objects per event, this scheme is reasonable.

The *CdfJet* objects also are primarily links, in this case to *CalData*. In order to calculate anything about jets, it is necessary to have *CalData*available, which takes about 3.7 kbytes per event. However, given the relatively large number of

[6]See §9.2 for a discussion of caching vs. on-the-fly calculation.

jets per events (around 6 for jet50 data to around 18 for top data, due to the low 1 GeV threshold) and the presence of several different jet clustering algorithms, the amount of space needed store the 8–12 quantities needed to make *CdfJet* objects useful without *CalData* is roughly 2.7 kbytes per event. This is not significantly less than *CalData* itself. Thus, the present scheme for *CdfJet*s is reasonable.

However, it is possible that some people doing jet physics may want to be able to make intermediate data sets (that is, not ntuples) that contain a subset of jets (either a higher threshold or only some of the clustering algorithms). We encourage the QCD group to consider this and perhaps implement a version of *CdfJet* where the useful quantities are cached, and hence the jet objects can stand independently of *CalData*.

There does not presently exist an implementation of *CdfMuon*. The proposed class had the option to include several track candidates for each set of muon stubs. To the review committee, this seems like an unnecessary complication with little benefit. We believe that keeping the best track for each set of stubs is sufficient. However, if the Muon group feels that it is necessary (we don't believe it was done in Run 1), then keeping the second best track could also be done, although the user should have simple access to the quantities associated with the best track. Given the small number of muon candidates per event, we believe the best approach to muon is to cache most of the information (differences in x and z, χ-squares, calorimeter tower energies, and one or two isolation variables). In addition, there should be links to the track (or a *vector* of links, if more than one is needed) and to the muon stubs. There must be a four-momentum (not separate p_x, p_y, p_z, and E) method that returns the four-momentum of the associated track; this momentum should be cached in the *CdfMuon*.

6.2 Problems with Existing Use of Links

The EDM links have the problem that the `const` is dropped on the object that is being linked to. Robert Kennedy has proposed (during preparation of this document, has already begun to implement) a solution that removes the danger of allowing non-`const` access to objects in the *EventRecord*. This is a critical item, and Robert should be afforded whatever assistance (if any) he needs to see this problem solved.

We are also concerned with the degree of physical coupling[7] induced by use of the existing link classes. Robert has also begun work on decreasing this coupling. **We recommend that final goal should be a link class (or classes) that allows an instance of class *X* inside an instance of collection class *XC* to refer to an instance of class *Y* inside a collection of type *YC*, and which meets the following requirements**:

[7] See footnote 1 for a reference on physical coupling; see also §9.3 for more physical coupling issues.

- Class YC has compile-time or link-time coupling to neither class X nor class XC.
- Class Y has compile-time or link-time coupling to neither class X nor class XC.
- Coupling in name only is permitted.

7 Corrections

It is clear that several of these objects will require corrections to some quantities to make them optimally useful to do physics (for example, jet energy corrections, electron and photon energy corrections, beam constrained fits for tracks, and dE/dx corrections). None of the groups has considered these in detail yet.

We discussed two different options for the handling of corrections to reconstructed objects, each applicable to a different circumstance. Both recognize that the correction code will be changed more often than the code defining the classes for the reconstructed objects, and so it should be possible to change the corrections without modifying the reconstructed object's class.

The first option is to encapsulate the correction algorithm in a "correction object". One then gets at the corrected quantities by giving the original version of the reconstructed object to the correction object, and querying the correction object. This is most appropriate for those cases in which the correction depends only on constants within the correction object and the features of the object being corrected — and not on other features of the event.

The second option, which is appropriate when the corrections are substantial or require much additional information, is to treat the correction processes as another step in reconstruction. In this case, the correction algorithm should be encapsulated in a framework module; the input to this module is the pre-correction reconstructed object, and the output is a new reconstructed object, with appropriate bookkeeping information to indicate the parentage of the new object.

We recommend that these corrections be done by methods or functions that take as input the stored object (*CdfJet*, etc.) and returns the corrected quantity. The correction methods may have additional input concerning how the correction is to be done, but if this not supplied, should default to the group's best estimate of how to do the correction for the average user.

There should be uniformity between the various correction methods for the various objects. Since the corrected quantities vary, there will be differences in method names. However, the naming scheme should be similar for each, as should the order and meanings of arguments.

8 `const` Correctness

We are very concerned with the `const`-correctness of the code we reviewed. We remind all the code authors that care with `const`-correctness is critical. The design of the EDM allows access only to the `const` functions of items that have been stored in the *EventRecord*. Since this is the model accepted by CDF (which we support), it is important to assure that this policy is not subverted, either by error or by design.

The signs in the code leading us to concern about `const`-correctness were widespread use of `const_cast` and widespread use of `mutable` data members. Both of these are generally a sign of poor design or misunderstanding how to use a design — either in the EDM or in the physics objects themselves.

These problems are concentrated especially in the tracking objects code. **We recommend the authors of the tracking objects meet with Robert Kennedy, the author of the EDM classes, to discuss where the problems have arisen, and what can be done to remove them.**

Group leaders should perform periodic reviews, to be sure no inappropriate casting away of `const`-ness is being done.

9 Miscellanea

This section contains a selection of comments that did not seem to fit neatly into one of the previous categories. This should not be taken to mean that we think these are unimportant — they are just hard to classify.

9.1 Doubles vs. Floats

The built-in type `double` should be returned from accessor methods of the objects to prevent round-off error and give the best results for intermediate calculations. The actual value written out could be a `float`, to save space, or a `double`, if the extra precision (or range) is required.

9.2 Calculation vs. Cached Values

Methods (in physics objects) that calculate values must be kept simple. If a method does extensive calculations, it is likely to be more like an algorithm then a simple transformation of existing information. The most significant problem with algorithmic methods in the data objects is that the methods will need to be versioned, in a fashion similar to the **ROOT** streamer model. Each of the objects created will need to be tagged with a data object version so that the proper part of the method can be invoked for older data. The algorithms in complex methods are likely to be changed and can cause reproducibility problems. **We recommend that algorithm methods in physics objects be forbidden.**

9.3 Convenience Methods

Class member functions which call through to functions in pointed-to (or linked-to) contained objects should be avoided. That is, if *ThingWithTrack* contains a *Link<CdfTrack>*, and if the momentum of *ThingWithTrack* is determined by the contained *CdfTrack*, then *ThingWithTrack* should either cache the momentum of the associated track, and have a member function that returns that momentum, or it should not have a member function to return the momentum. It should *not* have a member function that call the member function `momentum` of the *CdfTrack* it contains; it may have a member function that returns the *Link<CdfTrack>* itself.

The reason for this prohibition is that having such functions forces a compile-time dependency on the linked-to object. This makes the high-level object tightly coupled to the linked-to object's interface and ties it to the linked-to object library. Beyond the physical coupling, it also adds a unnecessary maintenance burden to the author of the class which has this sort of function; each time the interface of the pointed-to function is changed, the other class must be modified to adapt to the new interface.

The Links should only cause a compile- or link-time coupling if they are traversed by the user application code. Merely returning a link should not cause any such coupling.

9.4 Efficiency and Correctness Issues

9.4.1 Steamers

Care should be given to reserving space in *vector*s when streaming in objects. This can improve object conversion performance.

9.4.2 Copy constructors

In several places there are copy constructors written that just copy the private members from one place to another. The default copy constructor (that will be written for you by the compiler) does this, and may be more efficient than such code. More important is the reduction of the maintenance burden: allowing the compiler to generate the copy constructor means there is no chance of forgetting to update the copy when a new member variable is added. Why write (and maintain) code that you do not need to write?

9.4.3 Unnecessary Callthroughs

There are several methods that do nothing useful. An example is:

```
void CdfTrack::deallocate()
{
  StorableObject::deallocate();
}
```

Since *CdfTrack* publicly inherits from *StorableObject*, this call just wastes time and creates an additional maintenance burden. If the function was not there, the result would be that the base class's `deallocate` would be called[8].

9.5 RCS Identifiers

Care must be taken about using RCS identifiers as code version identifiers. We have not searched through all the code to find out if they were or not. Since algorithms are often split across several header and source files, the RCS identifier of any individual file it not very meaningful. The CVS tags may be more useful.

9.6 Magnetic Field

Although the issue of how the magnitude of the magnetic field is accessed is not directly part of this review committee's charge, we must comment on the importance of having an implementation soon to stop the promulgation of hard coding the value where needed.

9.7 EDM Issues

These section includes a few comments that are directed more toward the EDM than toward the physics objects classes themselves.

9.7.1 Compressed Datasets

The discussion of the physics objects has brought up several scenarios that make building compressed datasets difficult. It appears that the EDM is on its way to addressing them, and the issues are understood. We mention the more important issues here, so that they are not forgotten.

- It must be made easy to copy subsets of objects from collections, including linked information into new instances and have that information written out.
- It could be there is an option or collection type where only objects in the collection are only written out if they are referenced by a link.
- Utilities must be made available either through the EDM or the physics objects that make it convenient to work with these collections. See the examples in §3.2.

[8] We also note that the `deallocate` method in *StorableObject* does nothing except print to `cerr`. See §9.7.2 for comment.

9.7.2 Destroy and Deallocate

It is unclear to us how one should properly use `destroy` and `deallocate`, and how they are related to the destructor of the class. The `Edm/doc` directory should be populated with a user guide that explains how to use methods and why they exists, and each of the *StorableObject* subclasses should be inspected to assure that they are implementing (or using) these functions correctly. Many of these classes merely call through to the base class's function.

9.7.3 Typedefs

The various handle classes should contain a standard `typedef` (we suggest `element_type`, as used in *auto_ptr*). This is to allow for generic programming constructs to manipulate handles.

Each of the EDM collection classes (*RefVector*, *ValueVector*, *etc.*) should contain a standard `typedef` (we suggest `collection_type`) that describes the kind of collection it holds. This is to allow for generic programming constructs to manipulate collections.