

# Core CMS Event-Processing Software Re-engineering

L. Bauerdick, K. Bloom, W. Brown, P. Elmer, V. Innocente, J. Kowalkowski,  
M. Paterno, E. Sexton-Kennedy, W. Tanenbaum, L. Tuura, and A. Yagil  
Revision 1.35

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Requirements, Constraints, and Guiding Principles</b>	<b>2</b>
<b>3</b>	<b>Overview of the Design</b>	<b>4</b>
3.1	Interaction with External Systems . . . . .	5
3.2	Organization of the ROOT Output . . . . .	5
<b>4</b>	<b>Usage Patterns for the Persistent Data</b>	<b>5</b>
<b>5</b>	<b>The Core Infrastructure</b>	<b>7</b>
5.1	The <i>EventStore</i> . . . . .	7
5.2	<i>EDProducts</i> . . . . .	7
5.2.1	Constraints on an <i>EDProduct</i> and its Constituents . . . . .	8
5.2.2	Common Bookkeeping Information . . . . .	8
5.2.3	Rules for <i>EDProduct</i> -derived Classes . . . . .	9
5.3	Modules . . . . .	10
5.3.1	General Characteristics . . . . .	10
5.3.2	Types of Framework Modules . . . . .	10
5.3.3	<i>EDProducers</i> . . . . .	10
5.3.4	Mixing Modules . . . . .	11
5.3.5	Input and Output Modules . . . . .	11
5.4	<i>Selectors</i> . . . . .	11
5.5	The Scheduler System . . . . .	12
<b>6</b>	<b>Facilities Used by the Core Infrastructure</b>	<b>12</b>
6.1	Factories . . . . .	12
6.2	Services . . . . .	13

---

6.3	Threads . . . . .	13
6.4	Interfaces to Data Management . . . . .	13
6.5	The <i>ParameterSet</i> System . . . . .	14
6.5.1	<i>ParameterSets</i> . . . . .	14
6.5.2	Identifying Parameter Sets . . . . .	15
6.5.3	User Creation of Parameter Sets . . . . .	15
6.6	Non-Event Data . . . . .	15
6.7	Granularity of <i>CalibrationIDs</i> . . . . .	16
<b>A</b>	<b>Glossary of Terms</b>	<b>16</b>
<b>B</b>	<b>Mapping Between Existing and Proposed CMS Concepts</b>	<b>17</b>

---

## 1 Introduction

This document is a record of design decisions reached during a workshop held at Fermilab November 9–19, 2004. The participants were: L. Bauerdick, K. Bloom, W. Brown, P. Elmer, V. Innocente, J. Kowalkowski, M. Paterno, E. Sexton-Kennedy, W. Tanenbaum, L. Tuura, and A. Yagil.

This document is *not* a blueprint for design; it lacks many details. Instead, it captures the decisions reached at the aforementioned workshop. This document should evolve into—or be replaced by—a detailed plan to guide progress toward the goals described herein.

In §2, we recapitulate a number of the agreed-upon *desiderata* (requirements, constraints, and guiding principles) underlying the directions taken during the workshop discussions. Next we present in §3 a high-level overview of the design, followed by a discussion in §4 of the envisioned persistent data usage patterns. §5 details each layer of the core infrastructure, while §6 identifies the facilities that the core infrastructure relies on and uses. A glossary of technical nomenclature may be found in Appendix A.

## 2 Requirements, Constraints, and Guiding Principles

In this section we identify (in no special order) the underlying boundary conditions that serve as this design’s underpinnings. We refer the reader to “CMS Core Software Review” [Bloom, *et al.*, November 29, 2004] for selected motivation, use cases, experience, and other rationale underlying these *desiderata*.

1. A concrete class, directly corresponding to the concept of an event, is a required part of the design.

2. The design is required to incorporate explicit framework modules that communicate with each other only via the event class.
3. Data objects of the design are required:
  - to be composed of only simple data, and
  - to be isolated from the algorithms that create them.
4. There must be a system whereby modules can be scheduled for execution. This requirement does not preclude support for alternative mechanisms (*e.g.*, implicit invocation). This system must be:
  - efficient,
  - deterministic, and
  - easy to use.
5. A job that needs to look at XXX and YYY data (*e.g.*, reconstruction, analysis, raw data) must declare, up front, that it will access XXX and YYY data.
6. Event objects must be associated with appropriate metadata so that the metadata can be used to identify interesting event data.
7. The physical design must be carried out so as to minimize library dependencies.
8. Consideration must be given to the future role of multithreading. The design must achieve the traditional benefits of multithreading in order to justify its added complexity, or else it must dispense with multithreading in its entirety.
9. A *layered architecture* is a required design element. The following major layers have been identified:
  - a data layer,
  - an algorithm layer, and
  - a module layer.
10. We must have the ability to load all libraries at program start time, before handling of the first event. An important use case supporting this requirement is the HLT. The mechanism for explicitly loading libraries must be robust and user-friendly.
11. Framework jobs will run within some work flow management system (see §6.4). It is important to design the application framework with this system in mind. Consideration of all outputs, including job status as well as all inputs is important.
12. To guarantee reproducibility, full information (*e.g.*, algorithm parameters, calibration data) is required for “officially produced” event data. Such full information is not required for development by individuals doing something other than “official production.”
13. The following distinct usage patterns (see §4) for the persistent data must be supported:
  - use from “bare” ROOT (*i.e.*, no additional code or libraries needed),
  - use from ROOT with a small set of basic general purpose libraries,

- use from ROOT with a medium set of libraries (a.k.a. “ORCA-lite”),
- use in the full reconstruction, trigger, and simulation programs.

### 3 Overview of the Design

The design expressed in this document is a prototype, and is not complete. As in any software project of significant scope, details of the design may (and are likely) to change, as better understanding of how to achieve the stated goal within the stated constraints is achieved.

This document addresses only a part of the CMS software. Specifically, it describes a re-engineering of the core software for event-processing applications. It describes:

- a *framework for event-processing applications* (hereafter called “the framework”), and
- a set of classes for implementing a *model of the event-data* (hereafter called “the EDM”).

This document addresses other software systems (*e.g.*, the data management system) insofar as to describe boundaries between the system with which the event-processing applications interact, and to describe the responsibilities of each system.

The framework exists to make the task of writing reconstruction and analysis software simpler. It is responsible for handling the ordering (*i.e.*, scheduling) of event-processing tasks to be performed. The objects that perform the event-processing tasks<sup>1</sup> are called *modules* (see §5.3). The event-related data produced by event-processing are represented by the classes of the EDM.

The application framework will support two different scheduling models:

1. *explicit scheduling*, in which the user directly specifies the ordering in which modules are to be run (see §5.5), and
2. *no scheduling*—also called *reconstruction on demand*, *implicit invocation*, and *implicit scheduling*—in which the dependencies inherent in the data products determines the order in which modules are to be run (see §5.5).

Both scheduling models use the same modules and generate the same data products. The only difference is how the execution of reconstruction steps are scheduled.

Modules never communicate directly with other modules; they communicate only through objects passed to their functions. Modules that perform reconstruction, for example, communicate only through an *EventStore* object (see §5.1), which behaves as an in-memory database carrying information regarding the reconstruction (and perhaps simulation) of a single beam crossing. Functionality that must be shared between modules will often be packaged as *services* (see §6.2).

---

<sup>1</sup>We include in “event-processing” the processing of groups of events, such as “runs,” *etc.*

Elements of the framework that are user-configurable (such as the schedule in an explicitly scheduled application, and the various modules) are all configured using a single *parameter set system* (see §6.5).

### 3.1 Interaction with External Systems

A framework application (or, perhaps, an application wrapper that invokes the event-processing application) must communicate with several external systems. Such systems include (but are not necessarily limited to):

- the *parameter set system* (see §6.5),
- various *services* (see §6.2), *e.g.*, the calibration system,
- the *data management system* (see §6.4), which is responsible for access to files, and
- the *program status reporting system*.

### 3.2 Organization of the ROOT Output

In order to support the use of ROOT described in §2, and especially to support direct use of CMS data files from the ROOT prompt (see §4), the ROOT output files must follow a strict organization.

Within a single file, event data will be written to one ROOT tree. This tree will have one branch for each *EDProducer* instance (see §5.3.3), containing the output of that instance. The names of the branches will be automatically generated, to allow the system to ensure that there are no collisions. Most branch names will be in the form: “Class-NameOfEDProduct:UniqueParameterSetID,” where “UniqueParameterSetID” is the parameter set ID of §6.5.2. It is possible, in rare circumstances, for this branch naming algorithm to produce a collision. In cases where there is a branch with a duplicate name in a particular tree, the algorithm will append “:2” to the new branch name; additional digits will be used, if needed, to avoid further collisions. The branch names so generated will tend to be long, and thus not convenient for use at the ROOT prompt. We have spoken with Philippe Canal, of the ROOT core development team, who told us that ROOT can be extended to allow *aliases* for branch names. Such aliases will allow the user to associate a short name with each branch. Such names will be usable from the DRAW language, and from compiled functions called from the DRAW language.

## 4 Usage Patterns for the Persistent Data

Before discussing the details of the infrastructure, it would be useful to discuss the usage patterns that must be supported (see §2) in more detail.

Classes in the event data fall into three categories:

**Type 1** “Basic elements” (*e.g.*, 3-vectors, 4-vectors). The representation in memory, and the persistent representation, of such elements should be in terms of fundamental (C++) types. No code should be needed to interpret such basic elements.

**Type 2** “Higher level elements” (*e.g.*, tracks, jets).

**Type 3** “Packed raw data”.

Four usage patterns of the event data will be supported.

1. Use from “bare” ROOT.

To allow use of “bare” ROOT, the stored form of objects would have to be sufficiently simple to allow their use without class libraries or code to support their use.

2. Use with a small set of libraries.

Use of the data in this pattern may require the use of classes of type “basic elements” above. The set of “basic elements” must be defined by CMS.

Analysis of data in this pattern shall not require access to external databases, and thus event-data classes to be used in this pattern must be designed in a way to be useful without such access.

3. Use with a medium set of libraries (“ORCA-lite”).

Use of the data in this pattern may require the use of classes of type “basic elements,” and may also require classes of type “higher level elements”.

Analysis of data in this pattern shall not require access to external databases, and thus event-data classes to be used in this pattern must be designed in a way to be useful without such access.

4. Use in the full reconstruction, trigger, and simulation programs.

Use of the data in this pattern may require the full suite of CMS libraries, and may require connectivity to external resources.

We provide a brief example to demonstrate what is meant by use of “bare” ROOT to access CMS event data (*i.e.*, usage pattern 1).

Our example assumes a branch “ele” carrying electrons from one (electron finding) module, and a branch “trk” carrying tracks from one tracking module. Our example show how one might histogram the transverse momentum of the track associated with each electron in the electron collection:

```
1 // We have stored in the Electron a data member that is a smart
2 // pointer named 'tk', which contains an int named 'id'
3 t->Draw("trk.pt[ele.tk.id]")
```

If the electrons are sorted on  $p_T$ , and one wanted to get the  $p_T$  of the track associated with the highest  $p_T$  electron, then one would use:

```
1 // assuming electrons are sorted on pt...
2 t->Draw("trk.pt[ele[0].tk.id]")
```

## 5 The Core Infrastructure

### 5.1 The *EventStore*

There will be only one *EventStore* class.

Purpose: Responsible for managing lifetimes for each *EDProduct* it contains. Manages relationships between *EDProduct* and metadata. Provides access to event data (*EDProducts*) for any consumer of event data. Allows communication between “modules.”

A single *EventStore* instance corresponds to the detector output, reconstruction products, and/or analysis objects from a single crossing or the simulation of a single crossing.

*EventStore* is a concrete class.

It is possible to allow different *EventStore* interfaces, or merely different member functions, some of which perform ROD, and some of which do not.

- Any *EDProduct* should be immutable after insertion into the *EventStore* (see §5.2.3).
- The *ParameterSet* provenance of input objects to a particular *EDProduct* should survive the dropping (*dropping* means not writing to the output file) of the original input object.

The *EventStore* will use methods of the *Selector* class (see §5.4) to search for *EDProducts* matching a given criterion.

An ancillary class of the *EventStore* will keep track of the full invocation sequence

1. *EDProducer::produce*,
2. *EventStore::make*,
3. *EventStore::get*.

This information will be used to build a provenance “record” to be associated with the *EDProduct*.

### 5.2 *EDProducts*

Purpose: The basic unit of event data managed by the *EventStore*.

*EDProduct* is an abstract base class. Derived classes are also referred to as *EDProducts*. Each instance of such a class represents a component of an event, and is capable of persistence.

Each *EDProduct* instance has an ID that is unique within the event.

A “map” of the *EDProduct* instances for an event is kept in the event store.

An *EDProduct* that needs to be readable by bare ROOT may contain only built-in data types (*e.g.*, float, double, int), and must have the same shape in its transient and persistent forms. The data members of such a class should have meaningful names and allow simple use. Those *EDProducts* that need not be readable by bare ROOT (*e.g.*, raw data) may be packed and may or may not require additional software in order to be unpacked for browsing.

Each class that represents an *EDProduct* should be as simple as is feasible (with respect to the four usage patterns we have documented). In particular, usage pattern 4 objects (*i.e.*, objects that need external data to be usable) should be used only when necessary (for functionality or performance).

*EDProducts* are often collections, but they are not required to be. They should not be small.

### 5.2.1 Constraints on an *EDProduct* and its Constituents

There is a category of classes (for example, raw data) that do not have to be browsable from ROOT. A packed format, which needs some software for interpretation, is acceptable for such data. We do not imply that a packed format is *required* for such data.

How do we define which classes fall into this category? Some believe it is trivial to decide for each class.

In order to obtain usage pattern 1 above, we place the following constraints upon the implementation of the components of the EDM.

- *Change of Shape* is not allowed. The type of each component written to the file must exactly match the type seen in memory. However, this does not prohibit “puffing,” *i.e.*, the component may have transient (*i.e.*, not written to the file) members, provided that the values of such transient members are reproducibly determinable upon reading of the component from the file. Also, this does not prohibit conversion of a persistent item of a built-in type to a longer type (*e.g.*, float to double) in memory. However, any other form of packing is prohibited.
- A bit-packed representation is not allowed for any item that needs to be usable from the ROOT prompt.

### 5.2.2 Common Bookkeeping Information

There are several purposes for saving bookkeeping information:

1. To allow users to identify the *EDProduct* they want by identifying
  - a) the type of the *EDProduct*
  - b) the name of the “module” *instance* that created it—this is not merely the name of the *class* of that module; it is a name, unique within that executable, that identifies a particular “module” instance

- c) the configuration of the “module” that created it
- d) the calibration data used by the “module” that created it
- e) the processing step that created it.
- f) the release of the software that created it.

This may not be an exhaustive list.

2. To provide summary information that the user can take elsewhere to look at the actual parameter sets/calibrations/*etc.*

Sufficient bookkeeping information should be stored to allow re-production of the same *EDProduct* instance. This is not yet resolved for *simulation products*; it may be sufficient to reconstruct the entire event. There may also be a problem involving *regional reconstruction*; this seems resolvable by identifying as part of the algorithm the description of the region on which it acted.

This bookkeeping information will be used by the *Selector* class (see §5.4). Some selectors will use *all* the information to make “perfect matches.” Other selectors can use *some* of the information, and then possibly match more than one *EDProduct*.

Each *EDProduct* instance must be associated with its bookkeeping information.

### 5.2.3 Rules for *EDProduct*-derived Classes

- An *EDProduct* instance should not depend upon the classes that create it.
- An *EDProduct* instance should be immutable once it is made persistent.

Despite the immutability of an *EDProduct*, there are two ways in which an *EDProduct* in the *EventStore* may be augmented:

- **extensible collections:** in which new objects may be added to collections already in the *EventStore*.
- **decoratable objects in collections:** in which a new *EDProduct* is added to the *EventStore* and is associated with with an *EDProduct* already in the *EventStore*.

In addition, both “puffing” and “refitting” will be supported.

*Puffing* means expanding existing data in an *EDProduct*, using no event information from outside that *EDProduct*. Outside *non-event* information (*e.g.*, detector geometry) used in creating the original *EDProduct* may be reused.

*Refitting* means generating a *new EDProduct* from an older one, using new and different information from *outside* the original *EDProduct*.

## 5.3 Modules

The purpose of a module is to encapsulate a unit of clearly defined event-processing functionality, in an independently testable and reusable package.

### 5.3.1 General Characteristics

Here are some characteristics of *Modules*:

*Modules* is the generic term for all “workers” in the framework. Not all modules have the same interface.

*Modules* are scheduled by the *ScheduleBuilder*, and invoked by the *ScheduleExecutor*. Each *Module* instance is configured with a *ParameterSet*.

*Modules* must not interact directly with (*i.e.*, call) other modules.

Only *Modules* are “configurable.” An internal algorithm is configured by “percolating” *ParameterSets* to the algorithm, by the *Module* that contains the algorithm.

### 5.3.2 Types of Framework Modules

Here is a (possibly non-exhaustive) list of framework module types:

- event data producers
- mixing
- input
- output
- filter
- analyzers (readonly)

### 5.3.3 *EDProducers*

The only service of an *EDProducer* is to produce *EDProduct* instances and placing them in an *EventStore*. This service is performed by its `produce(EventStore& ev)` method.

On invocation a transaction is started.

The *EDProducer* will create empty *EDProducts* by asking the *EventStore* to make them

```
Handle<EDP> it = ev.make<EDP>();
```

At this point, the *EDProducer* is ready to populate this *EDProduct* with the real re-constructed objects.

If its algorithm requires information from the event, it will get it from the event-store using its `get(vector<Handle<EDP2>>& edps, const Selector& s)`.

### 5.3.4 Mixing Modules

A *MixingModule* takes in a sequence of `const EventStores` and merges corresponding data objects from each into a single output merged *EventStore* which is passed back to the framework. This is its only purpose.

### 5.3.5 Input and Output Modules

*InputModule* is an abstract base class.

The *InputModule* class provides the “interface” to read objects from the “I/O system.” A “Database” model will be used, that is, specific *EDProduct* instances will be explicitly retrieved.

We discussed how the *InputModule* uses the data management system to deliver requested events to the “user,” who specifies things like a “process step,” “code version,” *etc.* The data management system resolves this to a set of files, but that isn’t enough—because the user wants only some of the events in those files. The data management system could also deliver an “event catalog” that says what events are to be included. We have agreed that an event catalog is important.

CDF notes that a system that requires strict file delivery order causes trouble. Such an ordering can avoid thrashing on “conditions data.” But the cost has been large for CDF. Creation of an event directory reduces the need for strict file delivery ordering.

Event directories can live either in the data files (such as an AOD) or in their own files. Different event directories can refer to the same data files. It seems critical that a given process use whatever event directory the user “points at.”

## 5.4 Selectors

Selectors provide the mechanism by which one specifies what pieces (*EDProducts*) of an event are of interest. They are the “query mechanism” of the EDM.

The *EventStore* uses *get* methods of the *Selector* class to search for *EDProducts* matching a given criterion. Internally, the *get* methods use the bookkeeping information to determine which *EDProducts* are a match.

In its main *get* method, `match(const Handle<EDP>& edp)`, the *Selector* will search in the event store for *all* *EDProduct* instances matching *Selector*.

*EventStore* also supports a `get(Handle<EDP>& edp, const Selector& s)` method that will produce an error unless there is one and only one *EDProduct* instance matching *s*.

## 5.5 The Scheduler System

The scheduler system is the subsystem in the framework that is responsible for executing the sequence of reconstruction steps in the appropriate order.

We will use a system that supports two mutually exclusive types of scheduling.

- explicit scheduling
- no scheduling

Which form of scheduling is used is at the option of the user running the program.

We have agreed to put off calculated scheduling; it may be reconsidered at a later date.

The *ScheduleBuilder* is responsible for organizing the network of modules to be invoked, and assuring that they are invoked in the correct order. It builds the schedule used by the *ScheduleExecuter*.

Both *ScheduleBuilder* and *ScheduleExecuter* are concrete classes.

The *ScheduleBuilder* is configured by the same system as the *EDProducers*.

The *ScheduleBuilder* must know the sequence of *EDProducers* for each “path,” and how each *EDProducer* is configured.

The *ScheduleExecuter* must assume that each *EDProducer* may request stopping of execution of that “path.”

The *ScheduleExecuter* deals with “framework tasks,” which may include checking memory usage between *EDProducer* invocations.

The *ScheduleExecuter* should be able to decide what action should be taken upon each return status of a *Filter*.

## 6 Facilities Used by the Core Infrastructure

### 6.1 Factories

Several parts of the core infrastructure (most importantly instances of framework *modules*, described in §5.3) are created in response to the user’s run-time configuration of the application. The SEAL plug-in mechanism<sup>2</sup>, which implements the Abstract Factory pattern, will be used for all classes which require support of the Abstract Factory pattern. The core infrastructure will ensure that all use of dynamically loadable libraries (.sos) follows the requirements described in §2.

---

<sup>2</sup>See <http://seal.web.cern.ch/seal/snapshot/work-packages/pluginmanager/index.html>.

## 6.2 Services

Several parts of the core infrastructure are implemented as *services*. A service is a software component which provides, to other components (including *modules*, §5.3), shared use of some facility. Most often, the configuration and updating of a service is controlled by the framework itself, and not by the other components that use the service. One example of a service is the *calibration service* which is responsible for providing access to calibration data for the various detector elements. Reconstruction modules (*EDProducers*, §5.3.3) make use of the calibration service but are *not* involved with making sure that the correct calibration data are loaded; this is done “behind the scenes” by the framework itself interacting with the calibration service.

Note that there is *not* an *event service*. We do not want users to have global access to event data; in order to enhance the maintainability of the code, access to event data is available (in a framework application) only through an *EventStore* instance, passed as an argument into the code in question.

The SEAL component model<sup>3</sup> will be used to implement services.

## 6.3 Threads

Non-framework code should not spawn threads. Most importantly, code in *EDProducers* should not spawn threads, nor should it need to handling locks, mutexes, semaphores, or other artifacts of multi-threaded programming. Such code is required to follow a few simple rules (*e.g.*, “never use static member data or function-local static data that is non-const”).

## 6.4 Interfaces to Data Management

This topic was only started to be discussed, and the data management task will have to work with the framework designers to firm up this section.

The data management system deals with all information the user needs to specify the collection of events to be processed by the framework application. This “data set discovery” information includes specifying

1. the offline stream/data set (defined as the class of events given by a physics selection, like all triggers with a given trigger path, or a given event generator with a given set of parameters, *etc.*)
2. data tiers
3. the kind of processing done on the related event data (production pass ID)
4. constraints on event ranges (*e.g.*,  $1 \text{ pb}^{-1}$  of luminosity, or run range  $x$ , *etc.*)

---

<sup>3</sup>See <http://seal.web.cern.ch/seal/snapshot/workbook/seal-component-model.html>.

Event data files contain the actual event data, and do not contain all the relevant non-event data. However, event data file may contain copies of subsets of this non-event data.

Framework applications query the data management system to discover which data sets meet the user's specification. The input module(s) (see §5.3.5) interacts with the data management system to map the data set to a collection of files to be processed. The details of this interaction need to be designed in conjunction with the data management task. A framework application knows only about a set of files, and perhaps a catalog file, and processes those files.

The framework application, when it is running as a production job, is producing (a set of) files containing event data. These will become part of CMS data sets through an "import" operation into the CMS data management system. Thus, when event data files are being written, descriptive information regarding the production and the physics contents of these files must be made available to the data management and data set book keeping system. Because event data are intended to be immutable, the framework application must obtain all relevant information that would identify the event data output as becoming part of a CMS data set before writing the event data files.

There are many open questions, only some of which have been addressed yet.

## 6.5 The *ParameterSet* System

### 6.5.1 *ParameterSets*

Some of the elements in a framework application can be configured at run-time by the user. All such elements will be configured by a common *parameter set system*.

A *ParameterSet* contains a collection of name/value pairs, and provides type-safe access to them. The contents of a *ParameterSet* are uniquely identified by a *PS.id*. The contained values can be anything from the following list:

- `bool`
- `long`
- `std::vector<long>`
- `double`
- `std::vector<double>`
- `std::string`
- `std::vector<std::string>`
- `ParameterSet`
- `std::vector<ParameterSet>`

It is important to note that parameter sets can be nested.

*ParameterSets* used for *official production* must be registered in a central database. IDs for such parameter sets must be distinguishable from IDs associated with parameter sets not registered in the central database.

*ParameterSets* can also be *local*; they then are associated with an ID unique *within the data file*. Local *ParameterSets* are stored in the same file as the *EventStores* with which they are associated.

An entire executable should be configured using a single *ParameterSet*, which contains the many *ParameterSets* used to configure the *Modules* within that executable. Each module should be configured with a single *ParameterSet*.

There should also be a related system of untracked parameter sets. These are similar to *ParameterSets* in how they are presented to the user, but they do *not* have associated IDs, and are *not* tracked in any repository. They are to be used to carry information which does *not* need to be tracked in the bookkeeping system. One example of such information is the verbosity of the logging level used when running a program. Untracked parameter sets should *not* be used to provide any configuration information that affects the physics of reconstruction results.

### 6.5.2 Identifying Parameter Sets

There will be a central authority to assign unique IDs to *ParameterSets* and to store those *ParameterSets* used in official processing. There will be, in addition, a local repository of *ParameterSets*, in the event data files themselves. This is needed, in part, to allow use of reconstruction code without contacting the global authority—for purposes *other* than official event processing.

*PS\_ids* are calculated from the contents of the *ParameterSet* by the MD5 algorithm, giving a 16-byte identifier. This means if two IDs are different, the parameter sets to which they refer are surely different. If two *PS\_ids* are the same, then it is very likely, but not 100% certain, that the *ParameterSets* to which they refer are the same.

### 6.5.3 User Creation of Parameter Sets

A set of tools (such as a GUI parameter set editor) will be provided. Such tools are needed to make creation and manipulation of *ParameterSets* simple.

## 6.6 Non-Event Data

We identify two different kinds of tracked metadata:

- Algorithm configuration data – we’ve been calling these *ParameterSets*. These have a central authority to provide a unique ID for each instance of a parameter set.
- Calibration, survey, *etc.*, data – these have a different central authority to issue unique IDs. A single ID should suffice to specify all such data. We believe this can be done by having a hierarchical organization of such calibration (*etc.*) data.

The system that handles “program configuration” parameters is not the same as the system that deals with “calibration constant” data.

We began to discuss how to handle various sorts of metadata:

- Algorithm configuration data
- Calibration data
- Survey data

This may not be a complete list of all types of metadata.

## 6.7 Granularity of *CalibrationIDs*

It is possible to have a single *CalibrationID* refer to the whole set of calibrations used in a program. This makes saving them simpler. It has a drawback that some piece of reconstruction (for example, calorimeter towers) might not depend on part of the calibration (for example, the muon calibration). If two calibrations differ only in a part that does not matter, then saving a single ID that refers to the whole set makes things that are not *essentially* different *appear* different.

## A Glossary of Terms

It seems useful to agree up a set of terms to use for the various ideas we have been discussing. Here is a working list of the terms we have used. This list is an uneven mixture of items, some of which are very general and some of which are very specific.

***EDProduct*** Abstract base class of “things” stored in the *EventStore*.

Sometimes we use the term *EDProduct* to mean an instance of a concrete class which derives from *EDProduct*.

***EventStore*** A concrete class. *EventStore* provides the interface used by *Module* code (among other clients) to obtain *EDProducts* used for input, and also the interface to which *EDProducts* are published.

***Module*** Abstract base class of all the “worker units” manipulated *directly* by the framework.

***EDProducer*** A *Module* which puts *EDProducts* into the *EventStore*. Often, it will put only one; it is allowed to put more.

***ModuleFactory*** A *ModuleFactory* creates *Module* instances.

***Subsystem*** A *subsystem* is a loose collection of objects which act together to perform some clearly identifiable task.

## B Mapping Between Existing and Proposed CMS Concepts

Many of the concepts described in this note find direct equivalent in most of the event processing framework in use in High Energy Physics at least in terms of provided functionalities. In most of the cases the mapping is not one to one among classes as architectures differ. The following description tries to map the functionality and interfaces of the proposed system to the one present in COBRA. It mainly addresses classes present in the user API. Due to the difference in architecture in some cases it details functionalities that are hidden to the user but provided by one of the classes described. The new system is supposed to provide enhanced functionality, we address here only the functionality that can be found in the current CMS software.

**EventStore** *RawEvent* and *TRecEventWP* classes are the closest concept in COBRA to the proposed *EventStore* (at least from a structure point of view). Unlike *EventStore*, these classes are not directly visible to the user. The services provided by *EventStore* are currently dispersed in various proxy-layers such as *G3EventProxy*, *RecCollection*, *PRecDet*.

**EventDataProduct** COBRA (CARF) *RawData* class is the closest concept to *EventDataProduct* one can find in the current CMS software. From a structure point of view also *TReconstructor* class maps well *EventDataProduct* (it is what *TRecEvent* and *PRecEvent* contain and return as result of a query. It acts as abstract container of reconstructed data) but it encapsulate additional responsibilities (management of reco on demand and of provenance tracking) that in the proposed model will be left to the *EventStore* itself. In the proposed model an *EventDataProduct* will be directly exposed as such to the user. In the current CMS software a set of proxy-layers (*RecCollection* for instance) is interposed between the data objects managed by the framework and the user her(him)self.

**Module** This concept does not exist as a unique abstract base class in COBRA. There are many base super-types of “Worker units” that either use or populate (directly or indirectly) the event: *Observers*, *RecUnits*, *RecDets*, *RawSources*, *DBPopulator*. Some of the functionalities that in the proposed model are candidate to be responsibility of a *Module* (in particular in the area of I/O) are currently implemented directly in methods of the “Event” classes.

**EventDataProducer** *RecUnit* and its derived class *RecAlgorithm* are the closest concepts in COBRA of the proposed *EventDataProducer*. Some functionalities related to create and populate event data are currently provided by a combination of *RawSource*, *ReadOutUnit* and *BasePRecDet* and their derived classes. These classes will be replaced (for what these specific functionalities are concerned) with *EventDataProducers*.

**Framework** Such a concept is not encapsulated in a single class in COBRA. The functionalities *Framework* will provide are currently dispersed in several CARF packages such as *SimApplication*, *RecApplication*, *SimReader*, *RecReader*.

**ParameterSet** The proposed *ParameterSet* package will provide the functionalities currently available in COBRA through *SimpleConfiguration* and *ParameterSet*. It will also provide configuration management functionalities currently absent in COBRA.

**Selector** Selector objects will be used in place of the current *RecQuery*. The optimization of the query mechanism itself will most probably involve some collaboration among *Selector*, *EventStore* and *ParameterSet* that will replace and enhance what currently is implemented in *RecConfig*, *RecCollection* and *RecoRegistry*.

DRAFT 1.35