HepODBMS

A higher level interface to ODBMS

- Goals of the Package
- Package Components
- Status & Plans

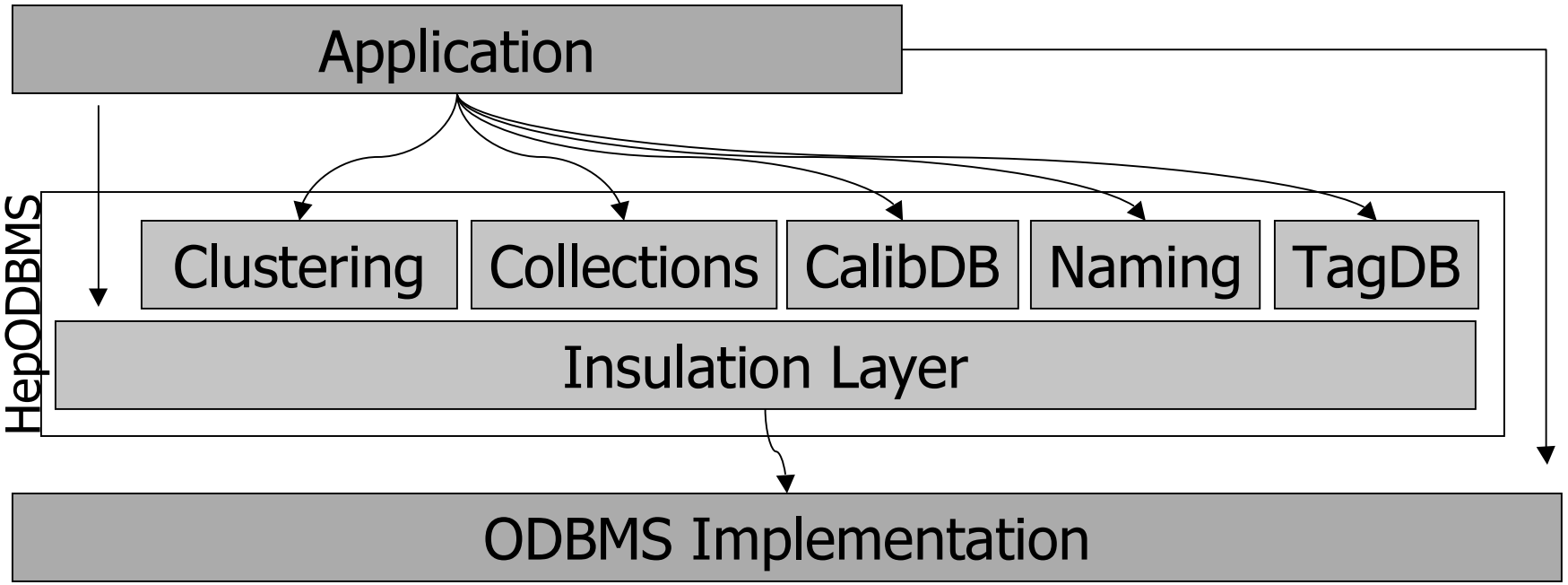
Tight Binding & Dependency

- ODBMS use a tight binding to programming languages like C++ or Java
- Why a tight binding?
  - Seamless integration of I/O on demand
    - No explicit I/O or data copies, just navigation between objects
  - Efficient down to single object granularity
    - Heavy use of inline methods, avoids virtual function calls
- Drawback
  - Tight binding means usually compile time dependency between application code and ODBMS API
  - Application code relies on details of a particular ODBMS implementation

ODBMS Binding Standard

- Need an API standard - e.g. ODMG
  - only a subset of the API is defined in the ODMG standard
  - only a subset of the ODMG standard is actually implemented by most vendors
- HepODBMS Goals
  - Provide an insulation layer for HEP applications
    - minimise database vendor and release dependencies
  - Provide a higher level interface that encapsulate
    - clustering & locking strategies, database session and transaction control
    - event collections, selection predicates, tagDB access, indexing

HepOBMS Overview



HepODBMS Packaging

- Low Level
  - odbms - Insulation Layer (header file only)
- Higher Level
  - goodies - Objectivity Helper Classes (transient)
  - rd45 - Miscellaneous Utilities
  - naming - Logical Data Organisation
  - collections - Persistent Collections
  - clustering - Physical Data Organisation
  - tagdb - Data Selection & Extraction Interface
  - calib - Calibration DB
  - etc - Schema & Build Support
- Each sub-package is organised as a separate include directory and library
  - higher level packages depend on insulation layer
  - transient user interface to persistent lower level implementation

“Neutral” Implementation Strategy

- No performance penalties
  - A *thin* insulation layer
    - no virtual functions for classes that don’t have them already
    - no function call overhead for inline methods
    - no increase of storage size for persistent objects
- No loss of ODBMS functionality
  - I/O on demand with transaction control
  - navigation in arbitrary object graphs
- No additional portability constrains
  - Runs on all platforms supported by Objectivity/DB
  - Persistent data should be usable in a heterogeneous environment

Insulation Layer

- ODMG Base Types
  - d_Long, d_Short, d_Boolean, d_Float, d_Double
- Persistent Object Base Class
  - d_Object, HepPersObject
- Object References
  - HepRef(T)
- Simple Collections and Associations
  - HepVector(T), HepRefVector(T)
- Trivial Implementation
  - mostly just a compile time type name indirection,
    some "all inline" wrapper classes
  - insulation is certainly not complete but the number of source lines
    containing ooXXXX is much lower!
- More complete insulation could be done at higher level
  - typically trading insulation against performance and functionality

Database Session Control

- HepDbApplication - End-user access to database session control, naming and clustering
  - Heavily based on the ooSession class from Objectivity
    - minor local bug fixes and extensions
  - Start/commit/abort transactions
  - Set lock handling options, lock wait time, number of retries
  - High level interface that allows to open/create FDBs, DBs and containers
  - Provide job or transaction level performance statistics
    - cache efficiency
    - disk I/Os
    - object accesses and updates
    - container and variable length object extension operations
  - Configuration using a method interface and/or environment variables

Setting up a DB session using the HepDbApplication class

```
main()
   {
   HepDbApplication dbApp; // create an appl. object
   dbApp.init("MyFD");     // init FD connection
   dbApp.startUpdate();    // update mode transaction
   dbApp.db("analysis");   // switch to db "analysis"

   // create a new container
   ContRef histCont = dbApp.container("histos");
   // create a histogram in this container
   HepRef(Histo1D) h = new(histCont) Histo1D(10,0,5);

   dbApp.commit();      // Commit all changes
   }
```

Physical Data Clustering

- ODMG-like bindings use the new operator to specify the object clustering
  - e.g. which db file, which container, close to which old object should be used to store a new object

- Encapsulate the clustering strategy in "Clustering Hint" objects
  - HepAbstractClusteringHint
    - abstract base class
  - HepContainerHint
    - clustering into single physical containers (< .5 GB for 8kB pages)
  - HepClusteringHint
    - clustering into logical containers (infinite size, spread over several db files)
    - parallel writing without lock contention
    - parallel load balanced reading
    - persistent definition of clustering

Clustering by Class

```
// class definition in Track.ddl
class Track : public d_Object {
    d_Double phi;
    d_Double theta;
    d_ULong  noOfHits;
// more stuff
public:
    static HepContainerHint clustering;
};
[…]
// define clustering at startup
Track::clustering = dbApp.container(“tracks”);
[…]
// use the clustering defined for tracks
HepRef(Track) aTrack =
      new (Track::clustering()) Track;
```

Persistent Clustering for Parallel Writers

```
// class definition in Track.ddl
class Track : public d_Object {
    d_Double phi;
    d_Double theta;
    d_ULong  noOfHits;
// more stuff
public:
    static HepClusteringHint clustering;
};

// find the clustering for tracks
if ( !Track::clustering.find("tracks"))
    Track::clustering.create("tracks"));
Track::clustering.setParallelWriterMode(noOfProcs,myID);

// clustering use spread all over the source code
HepRef(Track) aTrack = new (Track::clustering()) Track;
```

Logical Data Organisation

- Need a way to organise/lookup objects which are entry points into disconnected domains of our object model
  - e.g. Event Collections or Histograms
  - e.g. "well known" containers, databases
- Each user might need to reference thousands of those objects
  - Flat name space would become difficult to manage
  - Tree like approach (as used in file systems) is familiar to most users
- At the RD45 Workshop in February/April '98
  - Hierarchical naming service for (any) persistent object
  - Agreement on the main requirements

Naming Requirements

- External Naming
  - any persistent class may be named
  - no change to object schema
- Logical Naming
  - Naming hierarchy is independent of physical location
- Multiple Names for the same Object
- Scalable Lookup
  - E.g. One hash table per directory

- Not meant to replace associations with names!

HepNamingTree

- Abstract Naming Interface
  - HepNamingTree (transient)
  - Provides "file system"-like methods to navigate within the logical tree structure
    - nameObject(objRef,path), findObject(path), removeName(path), removeObject(path)
    - makeDirectory(path), changeDirectory(path), removeDirectory(path)
    - startItr(), nextItr()
- Concrete Implementation
  - HepMapTree - based on Objectivity's persistent hash tables (ooMap)
  - Internally uses persistent node objects

Limited Support for Meta Data

- HepMapNodes allow to keep some Meta Data
  - always
    - time of creation
    - object type
  - optional
    - extendible list of property value pairs (strings)
    - e.g. comment = “my higgs candidates”;
- Basic support for finding objects by property
  - iteration over directory or complete subtree
  - application of search predicate object
- Browser Example Programs
  - Text based simple shell, Java/Swing based GUI

HepODBMS Collections

- Why yet another set of collections?
  - Our requirements are different
    - very large collections
      - efficient set operations
      - efficient iteration order
    - problems with exposing the underlying implementation of many different collection types
    - need some integration of queries
  - Collections and Iterators are another MAJOR part of the visible interface of an ODBMS
    - E.g. Using Objectivity's physical containers directly is a major source of source code coupling
    - Extension of the HepODBMS insulation layer

Collection Implementation

- Templated collection of any type of persistent object
  - typedef h_seq<Event> EventCollection;
- Single class interface
  - STL interface independent of implementation
    - Single User visible collection class : h_seq<T>
    - Single STL like iterator: h_seq<T>::iterator
  - Uses hybrid of templated classes and delegation
  - User extensible through strategy objects
- Currently Implemented Strategies
  - vector of references (based on STL)
  - paged vector of references (based on raArray)
  - single container
  - group of containers ooVarray(ooRef(ooContObj))

Writer Example

```
 h_seq<Event> seq(”collections/myEvents", asSingleContainer );

HepRef(Event) evt;

for (int i=0; i<500000; i++)
    {
    // create a new event using the clustering hint provided by the event sequence
    evt = new(seq.clustering()) Event;

    // store the new object ref in the sequence (only needed for ref collections)
    seq.push_back(evt);

    // fill the event
    evt->setEventNo(i);
  }
```

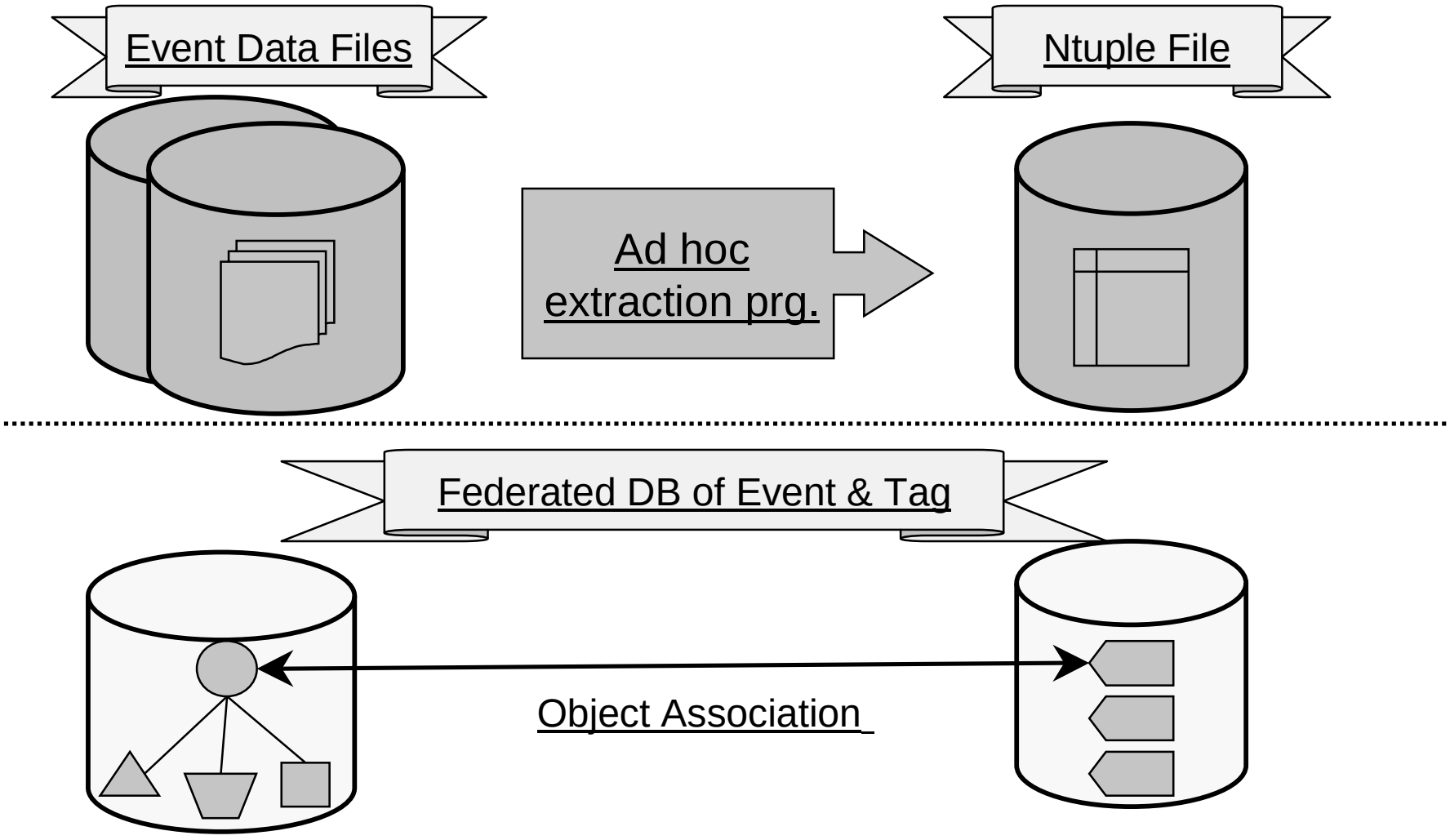
Reader Example

```
// find a collection using the naming service
h_seq<Event> seq("/usr/dirkd/collections/myEvents");

// STL like iterator
h_seq<Event>::const_iterator it = seq.begin();

while( it !=  seq.end() ) {
     cout << "Event: " << (*it)->getEventNo() << endl;
      ++ it;
}
// support for (some) STL algorithms
 int cnt=0;
 count(seq.begin(),seq.end(),1,cnt);
```

Ntuple versus TagDB Model



Purpose of Using Tags

- Tags are mainly used to speedup selections
  - Tag data is better clustered than the original data
- A collection of Tags defines an Event Collection
  - Tag collections are only a special case of an event collection
- Tag attributes may be visualised interactively
  - without the need to write any code
  - abstract interface class HepExplorableCollection
- Association to the Event may be used to navigate to any other part of the Event
  - even from an interactive visualisation program

Collections of Tags

- Generic Tags
  - concrete implementation of ExplorableCollection interface
  - Generic content: No need to define a new persistent class
  - May use predefined types: float, double, short, long, char
  - Additional attributes may be added later
  - Interactive display using IRIS Explorer

```
// create a new tag collection
GenericTag highPt(“high pt events”);

// define all attributes of my tags
TagAttribute<long>  evtNo(highPt,"event number");
TagAttribute<float> pt1(highPt,”p_t track1");
TagAttribute<float> pt2(highPt,”p_t track2");
TagAttribute<long>  nTracks(highPt,”number of tracks”);
```

Filling a Tag Collection

- Tag Attributes are used **just like other C++ variables**

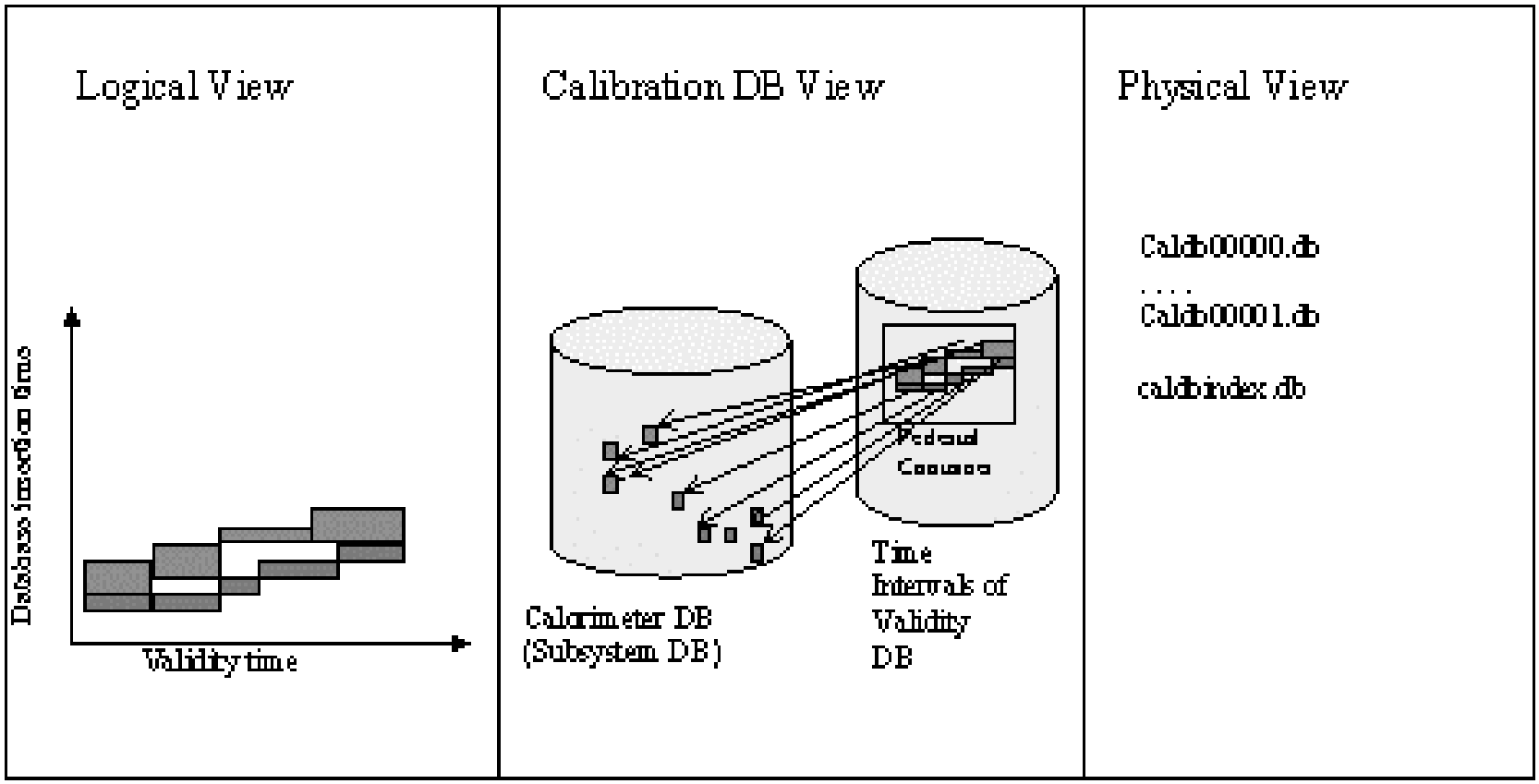
```
TagAttribute<long>  evtNo(highPt,"event number");
TagAttribute<float> pt1(highPt,”p_t track1");
TagAttribute<long>  nTracks(highPt,”number of
   tracks”);

if (highPtTracks > 2)
{
   // create a new tag for this event
   highPt.newTag(evt);
   evtNo     = evt->eventNo;
   pt1       = evt->Tracker.trackList[highPt1].pt;
   nTracks   = evt->Tracker.trackList.size();
}
```

Calibration Database

- Experiment independent toolkit for calibration data
  - based on the BaBar conditions package
  - integrated as a new package by Eva Arderiu-Ribera
- Calibration values
  - are user defined objects like any other persistent object
  - each re-calibration is stored as a new version
    - Old data is not deleted or updated
  - may be accessed via time of validity
    - Indexing: each new calibration value is stored in a B-tree for fast random access
    - Users may access any version of a calibration value
    - one particular version can be declared to be default
- Enhancements requested
  - **concept of global tags**

Database Structure



Schema Decoupling & Build Support

- HepODBMS defines named schemata to de-couple the type number allocation
  - two areas are used by HepODBMS itself
  - experiments are supposed to add additional named schemata
  - perl scripts are provided to exchange contents of single named schemata
- HepODBMS comes with platform independent makefiles
  - Abstract user makefiles
  - Platform dependent includes define global compiler and runtime system settings
  - Allows to build library and examples without changes on all supported compiler/platform combinations
  - Currently used by most of LHC++
  - Intention to move this service into a separate package

Documentation & Examples

- Reference Manual using DOC++
  - Class public and private interfaces
  - Inheritance graphs and alphabetic index
  - Generated from source code
  - Either as HTML and Postscript documents
  - http://wwwinfo.cern.ch/asd/lhc++/HepODBMS/reference-manual/index.html
- User Guide prepared by Eva Arderiu-Ribera
  - http://wwwinfo.cern.ch/asd/lhc++/HepODBMS/user-guide/ho.html
- Complete Example Programs (part of LHC++ examples)
  - /afs/cern.ch/sw/lhcxx/share/HepODBMS/99a-april/examples
  - populate a database with event objects
  - create a tag collection from events
  - batch analysis of tag collections
  - naming shell
  - creation and use of new collection classes

Status & Plans

- Version 0.3.0.0 has been released as part of LHC++ 99a
  - all LHC++ platforms now including Linux
  - in use by NA45, CMS, Atlas, LHC++ and Geant4
- Main new features
  - new compilers, Objectivity 5.1 {-beta for Linux}
  - packages CalibDB, Naming, Collections
  - completed shared lib support (Windows/NT)
- Plans for the next release
  - move to Objectivity 5.1.2
    - and alternatively Espresso 0.0
  - use naming to replace "hard-coded" database and container names
  - distributed registry of collections
    - support end-user collections
    - reduce lock contention on collection registry
  - additional clustering options for generic tags
    - e.g. clustering by attribute