



BABAR Database Workshop 8th-9th July 1997

Event API Strategies

David R. Quarrie

Lawrence Berkeley National Laboratory

DRQuarrie@LBL.Gov



Overview

- 1 Background
- 1 Exposed Persistence Strategy
- 1 Transient Wrapper Strategy
- 1 Status of Prototypes
- 1 Some Philosophical Concerns
- 1 Issues



Objectivity Restrictions

- 1 The .hh file is no longer the primary class declaration file
 - n Data Definition Language (.ddl) file replaces it - the .hh file is derived from this via the DDL compiler
- 1 All persistent-capable classes must inherit from a persistent base class (*BdbObj* or *d_Persistent_Object*)
- 1 A persistent-capable object cannot contain another persistent-capable object
 - n Can only contain a reference to it
- 1 A persistent-capable object can contain a transient one
 - n But only if the transient objects has no references to other objects



Objectivity Restrictions (2)

- 1 References to another object from within a persistent-capable object cannot take the form T^* , but the form $BdbRef(T)$ must be used instead
- 1 References from a transient object to a persistent-capable object must also take the form $BdbRef(T)$ or $BdbHandle(T)$
 - n The latter “pins” the object in memory
- 1 Persistent-capable classes can be created as transients, but with significant restrictions
 - n Cannot usefully contain references to other objects
- 1 Architecture independent basic datatypes should be used
d_Long, d_Float, d_Double, d_Boolean

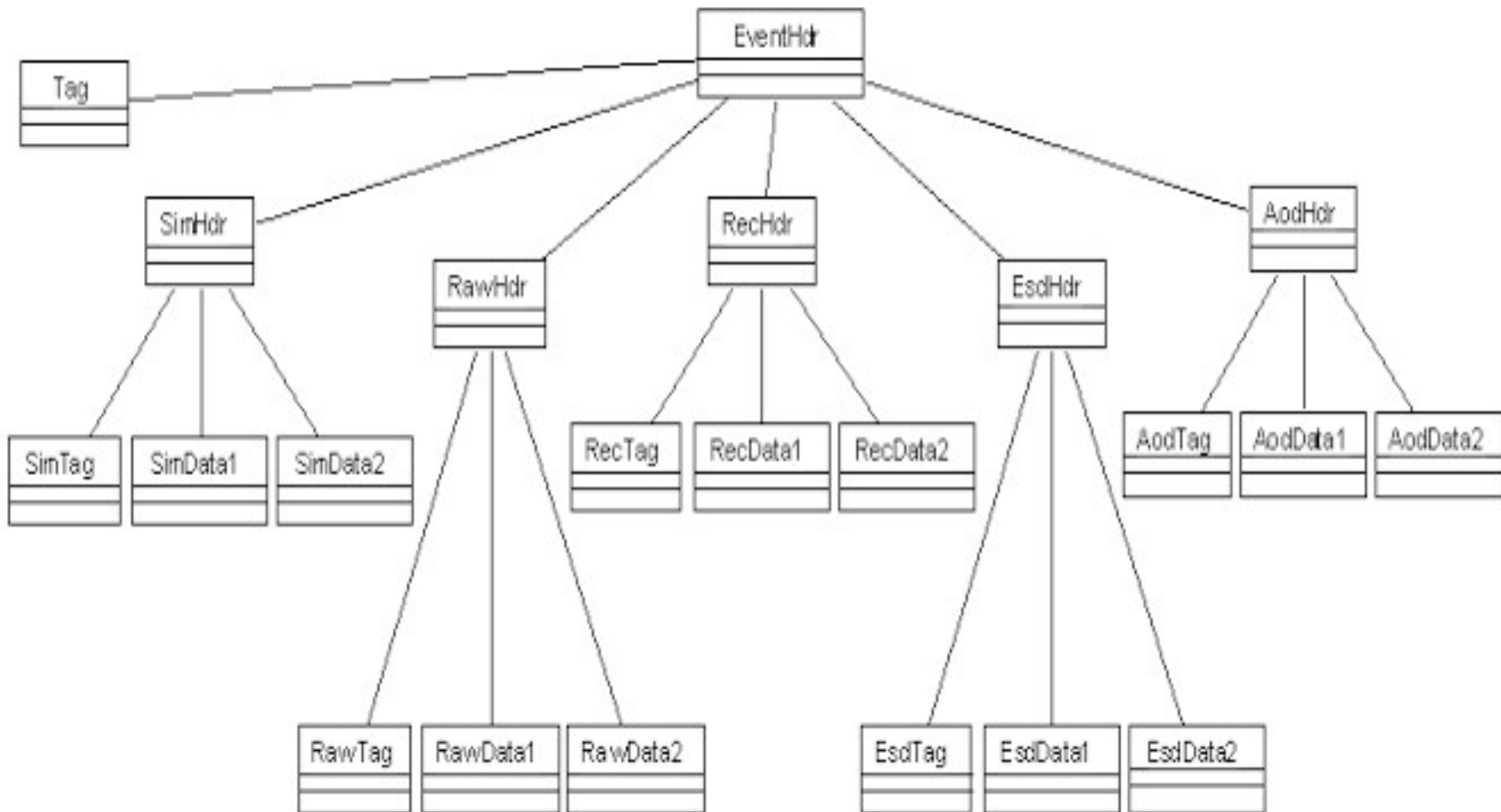


Objectivity Restrictions (3)

- 1 Persistent objects created using a clustering hint with *new*
 `BdbHandle(A) theA = new(A::clustering()) A;`
- 1 Persistent objects must be deleted using *BdbDelete(T)*
- 1 All persistent objects must be reachable by navigation from other objects or by name
- 1 Which persistent collection classes to use
 - n Objectivity intrinsic, STL, Tools.h++
- 1 In order to navigate efficiently, navigation nodes within the event should be small
 - n Leads to a hierarchical event data model rather than a flat one



Event Hierarchy





Restrictions Summary

- 1 Use of Objectivity has significant advantages, but not without some cost
- 1 Issue is whether to lessen the impact on the application programmer by shielding them in some way or to fully expose the database.
- 1 Different strategies



ProxyDict

- 1 Templated type-safe access to an extensible event
 - n Used in current transient event
- 1 Provides a dependency fire-wall
 - n Decouples clients from dependencies on components of the event that they are not interested in
- 1 Can be used to “flatten” the event
 - n Hide the hierarchy & provide access to leaf nodes
 - n Proxies perform the navigation
 - Must be written!
- 1 Current thinking is to continue to base the event interface on ProxyDict



Exposed Persistence Strategy

- 1 Database is exposed to client programmers
 - n Minimal wrapping with *BABAR* macros & classes
- 1 If a class is to be made persistent it is converted and all clients must be modified accordingly
- 1 Proposal is to use ProxyDict to flatten the event hierarchy but retrieve & store persistent objects directly
 - n Proxies are simple
 - Provide location navigation
 - Provide links to newly created objects so they can be located
 - n Note that number of proxies is probably identical for both strategies
- 1 Strategy promoted by RD45



Exposed Persistence Advantages

- 1 Single class per type
 - n A transient class can be converted to being persistent without increasing the overall number of classes
- 1 Simple to maintain
 - n No coupled class pairs to synchronize
- 1 Maximally efficient
 - n No memory to memory copies on access
 - n No smart pointer overhead (other than Objectivity)
- 1 Extensible to cover all classes
 - n Storage of adjustable parameters for reconstruction classes?
- 1 Simple Proxies



Exposed Persistence Disadvantages

- 1 Significant modifications to client code
 - n Not just to classes made persistent, but to all client classes
- 1 Use of .ddl file instead of .hh file complicates the dependency structure
 - n True for both strategies, but greater impact with this one
- 1 Use of *new* operator by mistake can create persistent leaks
- 1 *BdbDelete(T)* is irreversible
 - n Apart from transaction abort
 - n Read-only database access?
- 1 What-if analyses directly impact the persistent data
 - n *e.g.* removing hits for interactive track fitting



Exposed Persistence

Disadvantages (2)

- 1 Cannot easily use local caching within objects
 - n For performance or hashing *etc.*
- 1 Implications for OEP environment?
 - n Currently OEP foreseen to use raw data in “sequential” form
 - Use of reconstruction code requiring persistent objects?
 - Need to access conditions database
- 1 What persistent collection classes to use?
 - n Earlier talk
- 1 Some migration errors will be hard to detect
 - n e.g. using T^* instead of $BdbRef(T)$ will work but might fail
 - n Pinning using $BdbHandle(T)$ might help



Transient Wrapper Strategy

- 1 Attempt to hide database from client programmers
 - n Minimize impact of database
- 1 Retain existing interfaces
 - n Recompilation only
- 1 Coupled Transient/Persistent “sibling” classes
 - n Transient sibling has essentially same interface as current class
 - n Persistent sibling has simple interface declaring data members
- 1 Transient class created dynamically when requested
 - n *Ifd<T>::get()* request to ProxyDict
- 1 Persistent class created if necessary when event “stored”



Transient Wrapper Advantages

- 1 No (or little) modifications to existing classes
 - n Some modifications to “Data” classes, none to “Reco” classes
- 1 Standard C++ class definitions & event API
 - n Retains existing event API
 - n Conventional use of C++ pointers in client classes
- 1 Allows the possibility of compression of data in database
 - n Non-simple mapping between transient & persistent objects
 - n Minimize database overhead through embedded objects?
- 1 Allows for caching of information by transient objects
 - n Possible performance & space advantages



Transient Wrapper Advantages (2)

- 1 Use of existing Tools.h++ & CLHEP collection classes
 - n Mapped to smaller set of persistent collection classes
 - e.g.* Doubly-linked sorted list mapped to simple persistent vector
- 1 Allows easier “what-if” analyses without affecting the underlying persistent information
 - n Modifications made only to transient objects
- 1 Minimizes the impact to existing code base
 - n No modifications to non “data” classes



Transient Wrapper Disadvantages

- 1 Twice as many “data” classes
 - n What is a “data” class?
 - n Could all classes eventually be considered as “data” classes?
- 1 Proxies are more complex
 - n Must not just locate persistent information, but create the transient information on input
 - n Must create the persistent information on output.
- 1 Memory to memory copying
 - n Performance & space impact
- 1 Must synchronize transient/persistent siblings
 - n At run-time, and also at source code level



Transient Wrapper

Disadvantages (2)

- 1 Must use smart pointers to defer access to secondary information or create transitive closure
 - n If conventional C++ pointers are used to reference an object from the “primary” one, then that object also has to be created when the primary one is created.
 - n Transitive closure is complete set of all connected objects
 - Could be significant fractions of the event
- 1 Smart pointers are classes that have semantics of pointers
 - n \rightarrow & * operators etc.
 - n Used to replace data members within “data” classes
 - n Invisible to client classes
- 1 How many smart pointers & performance implications?



Polymorphism

- 1 Requirement - ability to handle polymorphic lists of objects
 - n e.g. EmcBump, EmClBump, EmcGVBump
- 1 Objectivity can only do this by dealing with references to each object
 - n Database overhead of 14 bytes
- 1 Both strategies must handle this requirement correctly



Prototypes

- 1 Understand code & design changes
- 1 How complex are proxies?
- 1 How many smart pointers?
 - n How complex are they?
- 1 Can both strategies meet the requirements?
 - n Any “drop-dead” limitations?
- 1 Understand migration issues
 - n How easily can we get from where we are now to full implementation?
- 1 What persistent collection classes?



Transient Wrapper Prototype

- 1 25 classes given persistent siblings (*e.g. EmcGHitP*)
 - n DchGHit
 - n DrcGHit, DrcDigi
 - n EmcGHit, EmcDigi, EmcCluster, EmcBump, EmcClBump, EmcGVBump, EmcPidInfo, EmcCand
 - n GTrack, Gvertex
 - n IfrGHit
 - n SvtGHit, SvtDigi, SvtCluster, SvtHit
 - n TrkFundHit, TrkHitOnTrk, TrkHotList, TrkRecoTrk, TrkRep, TrkSimpleRep, TrkCircleRep, TrkHelixRep
- 1 Corresponding proxies
- 1 Event Input & Output Modules



Persistent Classes

- 1 Based on Event data model shown earlier
- 1 Want to replace stage headers by templated ProxyDict-style classes to break dependency tree & simplify interface
- 1 Wanted to use Objectivity intrinsic collection classes
 - n ooVArray, ooMap
 - n Problem, ooVArray is not persistent-capable
 - n Temporary solution is to use the persistent-capable Tools.h++ simple vector class

Based on ooVArray but violates my self-imposed guidelines

Wrapped by BdbRWVector macro in an attempt to allow a migration path to a future implementation



Proxies

- 1 One proxy per class accessible from the event
 - n Returned as *HepAList*<T>* via *Ifd*<T>::*get*()
- 1 All inherit from base class
 - n Only specialization is to locate & create information within event hierarchy
 - n Probably collapse to one proxy per Stage Header if templated stage headers implemented as planned
- 1 Don't need a proxy for information not accessed from event header directly
 - n e.g. *TrkHitOnTrk* information accessed via track list



Smart Pointers

- 1 *BdbPtr* $\langle T, P \rangle$ deals with references to single objects

$T^* \Rightarrow \text{BdbPtr}\langle T, P \rangle$

- 1 T is transient class & P the persistent sibling

- 1 *BdbListPtr* $\langle L, T, P \rangle$ is abstract list pointer

- n *BdbHepAListPtr* $\langle T, P \rangle$ deals with *HepALists*

$\text{HepAList}\langle T \rangle^* \Rightarrow \text{BdbHepAListPtr}\langle T, P \rangle$

- n *BdbRWVectorPtr* $\langle L, T, P \rangle$ deals with Tools.h++ vectors

$\text{RWTPtrOrderedVector}\langle T \rangle^* \Rightarrow$

$\text{BdbRWListPtr}\langle \text{RWTPtrOrderedVector}\langle T \rangle, T, P \rangle$

- n *BdbRWListPtr* $\langle L, T, P \rangle$ deals with Tools.h++ lists



Smart Pointers (2)

- 1 *BdbHashDictionaryPtr*<*K*, *T*, *P*> deals with *RWTPtrHashDictionary*< *K*, *T* > *
 - n Had to cheat a bit for this - added a *key*() function member to *T* to specify the corresponding key
 - n e.g. *EmcCluster*

```
BdbHashDictionaryPtr< TwoCoordIndex, EmcDigi, EmcDigiP >
_memberDigiMap;
```
 - n *EmcDigi*

```
TwoCoordIndex* key( );
```



Smart Pointer Ownership

- 1 Who has ownership of objects referenced by smart pointer?
- 1 If it's directly accessible via the event then the event owns it
 - n Smart pointer does not have ownership & should not delete referenced object when it is deleted
- 1 What about *TrkRecoTrk* → *TrkHitList* → *TrkHitOnTrk* etc.
 - n *TrkRecoTrk* has ownership
 - n Smart pointer should delete referenced object when it is deleted
- 1 Need to be able to customize smart pointers to deal with both ownership cases

setOwnership(bool isOwner)



Caching, Polymorphism & Simplifications

- 1 *EmcCluster* demonstrates use of transient caching
 - n HashDictionary information doesn't need to be stored persistently since it can be derived from *EmcDigi* list instead
- 1 *EmcBump*, *EmcClBump* & *EmcGVBump* demonstrate use of polymorphism in lists
 - n *EmcBump* is parent to both *EmcClBump* & *EmcGVBump*
 - n Saved as *HepAList*<*EmcBump*> within the transient event
- 1 *TrkRecoTrk* has a *RWTPtrOrderedVector*<*TrkRep*>
 - n Mapped to simple vector in *TrkRecoTrkP*



Scaling Studies

- 1 Maximum of 30,000 events input
 - n Limited by 2GB database size & lack of RD45 clustering classes
 - n Not true stress test - 2000 events replicated 15 times
- 1 6000 events passed through part of the Emc chain
 - n True events, not smaller sample replicated
 - n EmcMakeDigi, EmcMake1dCluster, EmcMakeBump
 - n Currently limited by disk space
- 1 Not done any performance studies
 - n Throughput or space
- 1 Need to expand breadth across other subsystems



Transient/Persistent Limitations

- 1 Not all classes are fully implemented
 - n Some relationships “stubbed” for now
 - Not an intrinsic limitation, just simplification tradeoff
- 1 Several existing transient classes embed lists rather than contain references to them
 - n e.g. *SvtDigi* contains a *HepAList<SvtGHit>*
 - n Cannot defer creation of such lists
- 1 No inherent limitations found so far
- 1 Probably will have to add more smart pointers
 - n e.g. *Tools.h++* lists rather than vectors
 - And change the name of the existing smart proxy



Other issues

- 1 Need to ensure one-to-one relationship between transient object & its persistent sibling
 - n Consider case when objects A & B both reference object C
 - On input must ensure that only a single copy of the transient version of C is created even if both references are followed
 - 1 Object cache associated with transient event
 - On output must ensure that only a single copy of the persistent version of C is created even if both references are followed
 - 1 Transient object contain a reference to their persistent sibling
 - n Introduces dependency coupling between them
 - n But necessary for dealing with polymorphism anyway (transient object creates its persistent sibling via a *persistent()* member function)



Migration Recipe - Transient Class

- 1 Modify existing transient class
 - n Add *persistent()* member function
creates persistent sibling
 - n Add reference to persistent sibling
Prevents multiple creation of sibling
 - n Replace pointers to other objects by smart pointers
 - n Add #include statements for persistent sibling etc.
Complicates the dependencies
I've not tried to find the minimal set yet
 - n Make persistent sibling a friend class
Not absolutely necessary - simplifies interface



Migration Example - EmcDigi

```
#ifdef OBJYBASE
#include "EmcData/EmcDigiP.hh"
#include "EmcData/EmcGHitP.hh"
#include "BdbUtil/BdbPtr.hh"
#include "BdbUtil/BdbHepAListPtr.hh"
#endif
[...]
#ifdef OBJYBASE
    BdbRef(EmcDigiP) persistent( );
    TwoCoordIndex* key( );
    friend class EmcDigiP;
#endif
[...]
#ifdef OBJYBASE
    BdbPtr<EmcGHit, EmcGHitP> _mcTrue;
    BdbHepAListPtr<EmcGHit, EmcGHitP> _memberAList;
    BdbRef(EmcDigiP) _archive;
#else
    EmcGHit* _mcTrue;
    HepAList< EmcGHit > *_memberAList;
#endif
```



Migration Recipe - Persistent Class

1 Create Persistent sibling class

n Data members

ODMG basic datatypes, persistent collections, references to other objects (BdbRef(P))

n Clustering Hint Macros

BdbHintDeclare, BdbHintInit(Class)

n Static Creation function

BdbRef(P) create(T* aTransient)

n Constructors

P::P(T* aTransient)

n Transient creation member function

virtual AbsEvtObj* transient(BdbObjectCache* theCache) const



Migration Example - EmcDigiP

```
BdbHintInit(EmcDigiP);

BdbRef(EmcDigiP)EmcDigiP::create( EmcDigi* aTransient ){
    BdbRefAny theHint;
    BdbHandle(EmcDigiP) result( 0 );
    if ( 0 != aTransient ) {
        theHint = clustering( );
        result = new( theHint ) EmcDigiP( aTransient );
    }
    return result;
}

EmcDigiP::EmcDigiP( EmcDigi* aTransient )
: _energy      ( aTransient->_energy      ),
  _flags       ( aTransient->_flags       ),
  _weightSum   ( aTransient->_weightSum   ),
  _theta       ( aTransient->_theta       ),
  _phi         ( aTransient->_phi         ),
  _timeEnergy  ( aTransient->_timeEnergy  ),
  _eMax       ( aTransient->_eMax       )
{
    EmcGHit*      theTHit;
    BdbRef(EmcGHitP) thePHit;
    BdbHandle(BdbRWVector<EmcGHitP>) thePListH;
```



EmcDigiP (2)

```
BdbRefAny theHint;
// Create _mcTrue
theTHit = (EmcGHit*)aTransient->mcTrue( );
_mcTrue = theTHit->persistent( );
// Create _memberAList
const HepAList< EmcGHit >* theTHitList = aTransient->getMemberList( );
theHint = clustering( );
if ( 0 == theHint ) {
    theHint = BdbRawHdr::clustering( );
}
size_t length = theTHitList->length( );
thePListH = new( theHint ) BdbRWVector<EmcGHitP>(length);
for (size_t i=0; i < length; i++){
    theTHit = (*theTHitList)[i];
    thePHit = theTHit->persistent( );
    thePListH->replace( thePHit, i );
}
_memberAList = thePListH;
setTime( aTransient->time( ) );
}
```



EmcDigi (3)

```
AbsEvtObj*EmcDigiP::transient( BdbObjectCache* theCache ) const
{
    EmcDigi* theTransient = new EmcDigi;
    theTransient->_energy      = _energy;
theTransient->_flags          = _flags;
    theTransient->_weightSum   = _weightSum;
    theTransient->_theta       = _theta;
    theTransient->_phi         = _phi;
    theTransient->_timeEnergy  = _timeEnergy;
    theTransient->_eMax        = _eMax;
    if ( ! _mcTrue.isNull( ) ) {
        theTransient->_mcTrue.set( _mcTrue, theCache );
    }
    if ( ! _memberAList.isNull( ) ) {
        theTransient->_memberAList.set( _memberAList, theCache );
    }
    theTransient->_archive = ooThis( );
    return theTransient;
}
```



Conclusions from Transient Prototype

- 1 Viable strategy
- 1 No intrinsic limitations found so far
- 1 Relatively simple “recipe” for modifications to transient “data” classes & creation of persistent sibling classes
- 1 Some adverse additional dependencies are created
- 1 Fairly restricted set of smart pointers
- 1 Templated stage headers will reduce the number of proxy classes
- 1 Draft documentation available



Persistent Prototype

- 1 Very little done to date
- 1 Input Module, Output Module & Proxies written to deal with either strategy but code for persistent strategy protected by a preprocessor symbol
- 1 Steve Gowdy has made some modifications to some of the EmcReco classes as a first stage
- 1 Also a couple of EmcData classes have persistent versions (EmcGHit.ddl & EmcDigi.ddl)



Some Philosophy

- 1 I have reservations about exposing database to everyone
 - n Major impact on existing code & in some cases designs
- 1 Much of the data that is created is not going to be made persistent
 - n *e.g.* track finding
- 1 Most other environments shield the developers from the database
 - n Feedback from Objectivity Conference
 - n Advice from SLAC Computing Advisory Panel
 - n HEP appears to be the exception (RD45)



Summary

- 1 Two strategies under discussion
- 1 Small amount of effort on exposed persistence strategy
- 1 25 classes implemented with the transient wrapper strategy
 - n Some scaling studies done
 - n Much more testing to be done
 - n No show stoppers found
- 1 Valuable lessons learned
- 1 Draft documentation available for migration
- 1 We need to come of out here with a strong recommendation if not a decision