Geant 4

Persistency

Author: *Youhei Morita*

Note by MGP

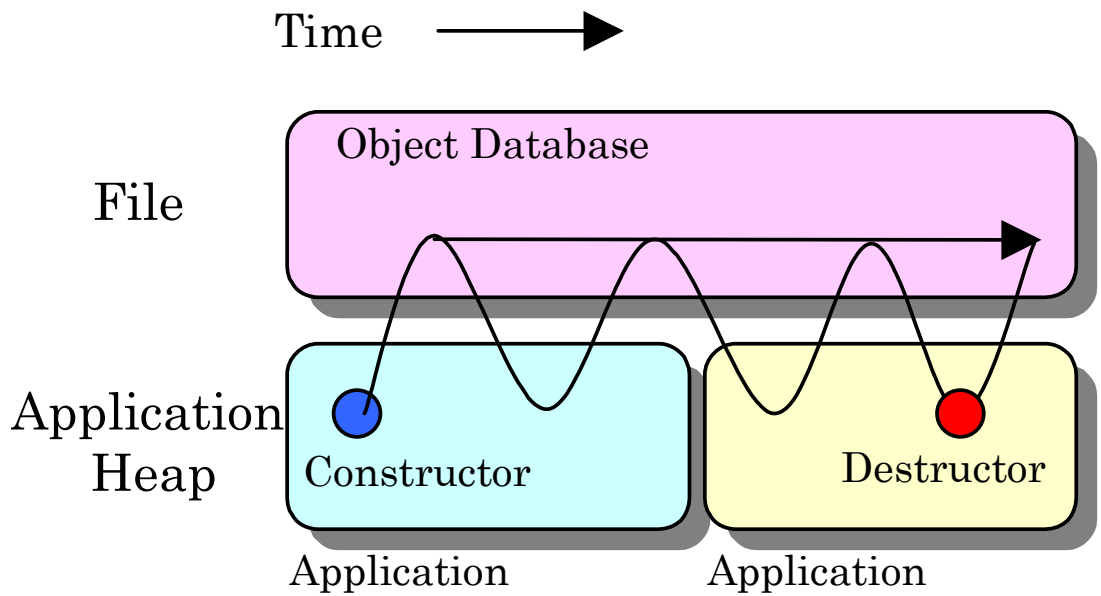
- As a novice user developing small/medium scale simulation applications, you are probably not interested in the details of persistency as described in the following
- Most probably, simple analysis objects (like tuple) would be sufficient to store the output of your simulation for further analysis

Category Requirements

- Geant4 Persistency makes run, event, hits, digits and geometry information be persistent, to be read back later by user programs
- Geant4 shall make use of industrial standard ODMG C++ binding and HepODBMS as persistency interface
- Kernel part of Geant4 should not be affected by the choice of persistency mechanism (Geant4 should be able to run with or without persistency mechanism)

What is “object persistency” ?

- Persistent object lives beyond an application process, may be accessed by other processes.
- When an object is “deactivated”, state of the object are stored into the database system. Once “activated”, the state information of the object is read back from the database.



What is ODMG ?

- Object Database Management Group

  a non-profit consortium of vendors and interested parties who collaborate to develop and promote standards for object database management systems (ODBMS). http://www.odmg.org/

- ODBMS Standard Documents ODMG 2.0 released in 1997
  - Object Model
  - Object Definition Language
  - Object Query Language
  - Language Bindings to C++, SmallTalk, Java

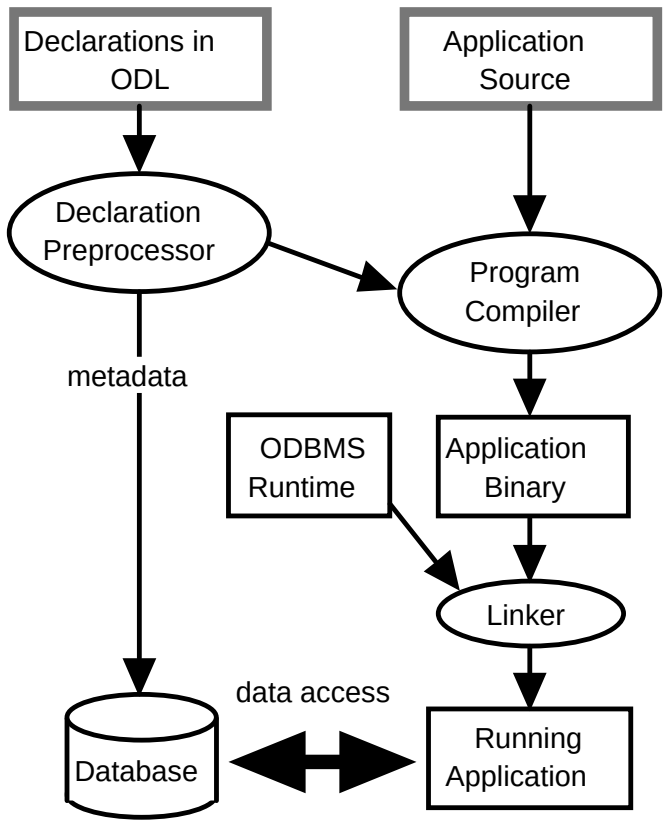
C++ Binding of ODMG

- Design persistent class using ODL (Object Definition Language)

```
class G4PEvent : public HepPersObj
{                                ↑
 public:                  persistent-capable base class
    G4PEvent();
      :
 private:
    G4Pint  eventID;  ← persistent-capable type
      :
}
```

- Compile ODL files (schema) to schema metadata, C++ header files, wrapper C++ source code.

ex. Objectivity/DB:
ooddlx preprocessor processes *.ddl files into *.hh, *_ref.hh, *_ddl.cc files, and stores schema metadata into a federated database file.



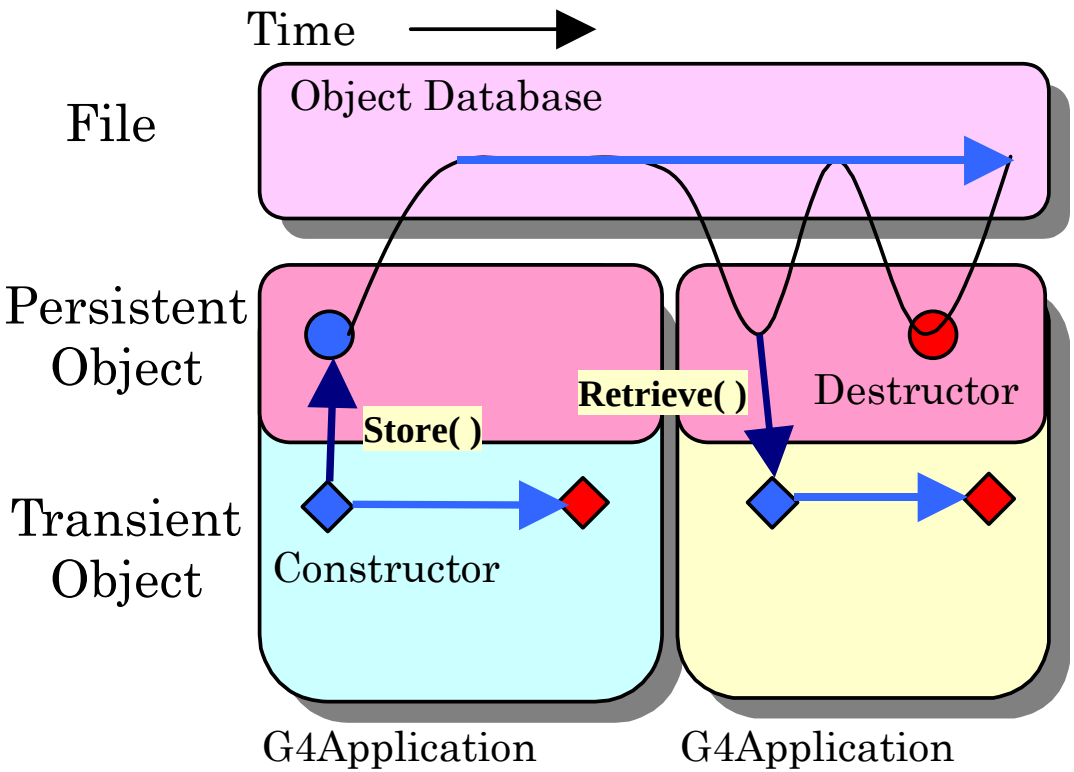
What is HepODBMS ?

- C++ class library that provides a simplified and consistent interface to underlying ODMG-compliant Object Database Management System
- Current implementation is based on Objectivity/DB
- Goals:
  - an insulation layer to minimize dependencies on a given database vendor or release.
  - high level base classes that encapsulate features such as clustering and locking strategies, database session
  - transaction control, event collections, selection predicates, tagDB access and calibration
  - whilst not introducing any significant performance or storage overhead.
- See Also:

  http://wwwinfo.cern.ch/asd/lhc++/HepODBMS/user-guide/H1Introduction.html

Persistency in Geant4

- "Parallel World" approach
  Data members of transient and persistent objects are copied by *Store( )* and *Retrieve( )*



G4 kernel objects have corresponding "P" objects in G4Persistency

G4Run ↔ G4PRun
G4Event ↔ G4PEvent
G4Hit ↔ G4PHit
: :

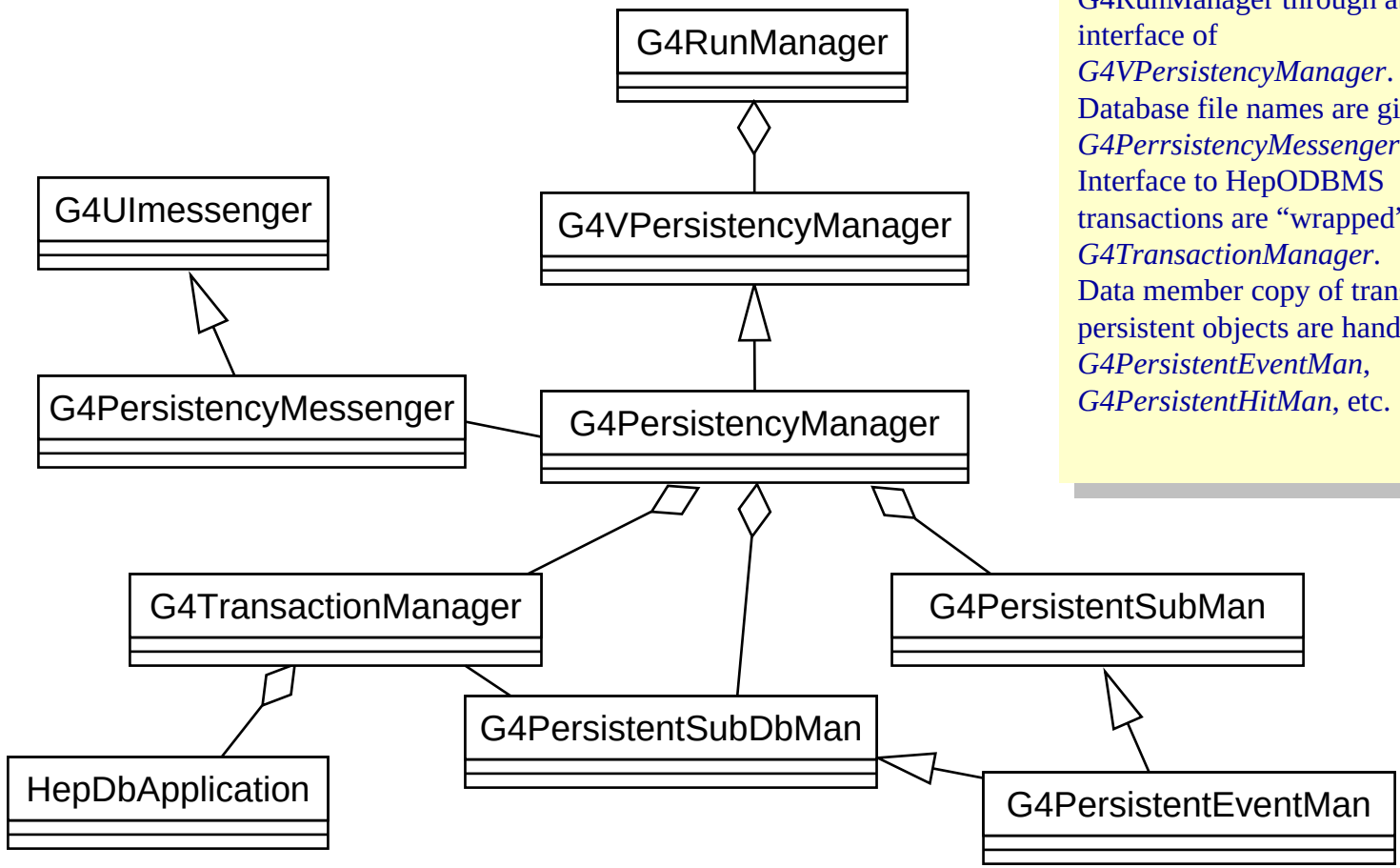
G4Persistency

Inherits from *HepPersObj* in HepODBMS

G4Kernel

Persistency in Geant4 (2)

- Top Level Class Diagram



Transient G4 objects are “stored” by G4RunManager through abstract interface of *G4VPersistencyManager*.
Database file names are given via *G4PerrsistencyMessenger*.
Interface to HepODBMS transactions are “wrapped” at *G4TransactionManager*.
Data member copy of transient and persistent objects are handled by *G4PersistentEventMan*, *G4PersistentHitMan*, etc.

How to design your own persistent objects in ODBMS

- Design persistent-capable classes
- Design the object clustering
- Design the access patterns
- Design the transaction scenario

How to design your own persistent objects in ODBMS:
Design persistent-capable classes

- Create ODL (DDL) files (similar to C++ header files)
- Inherit “persistency” from ***HepPersObj***
- Use ODMG persistent basic types such as d_Double, d_Float
  - In Geant4, basic types are cast into G4Pint, G4Pdoulbe *etc* in ***G4PersistentTypes.hh***
- Use ***HepRef()*** macro as smart pointers of persistent objects in run time

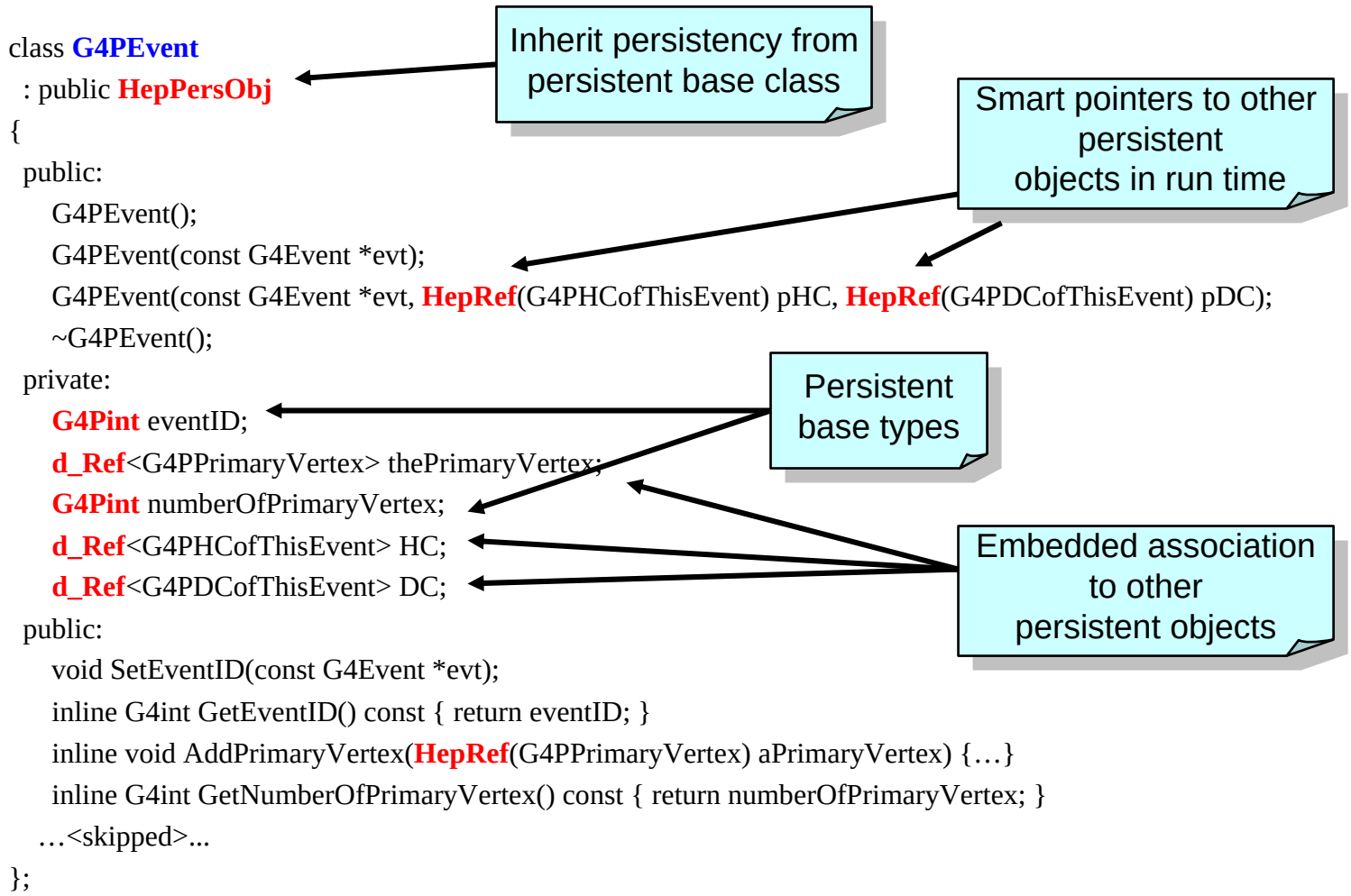
```
HepRef(G4PEvent) anEvt;
anEvt = new G4PEvent(….);
```

- Use ***d_Ref***<> template for embedded persistent association in ODL

```
class G4PEvent   : public HepPersObj
{  …
   d_Ref<G4PPrimaryVertex> thePrimaryVertex;
   …
}
```

- Use ***d_Varray***<> template for variable length array

How to design your own persistent objects in ODBMS: Design persistent-capable classes - *G4PEvent.ddl*



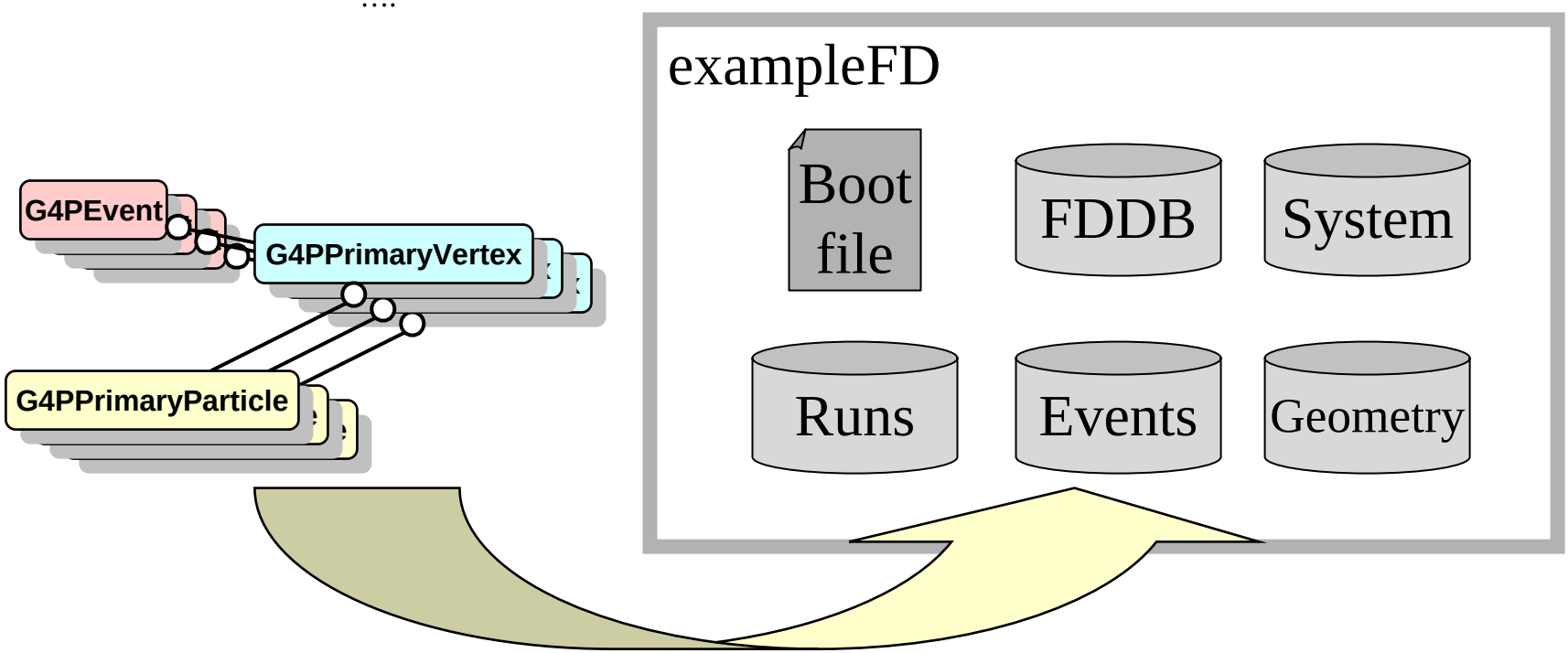

```
class G4PEvent
 : public HepPersObj
{
 public:
    G4PEvent();
    G4PEvent(const G4Event *evt);
    G4PEvent(const G4Event *evt, HepRef(G4PHCofThisEvent) pHC, HepRef(G4PDCofThisEvent) pDC);
    ~G4PEvent();
 private:
    G4Pint eventID;
    d_Ref<G4PPrimaryVertex> thePrimaryVertex;
    G4Pint numberOfPrimaryVertex;
    d_Ref<G4PHCofThisEvent> HC;
    d_Ref<G4PDCofThisEvent> DC;
 public:
    void SetEventID(const G4Event *evt);
    inline G4int GetEventID() const { return eventID; }
    inline void AddPrimaryVertex(HepRef(G4PPrimaryVertex) aPrimaryVertex) {…}
    inline G4int GetNumberOfPrimaryVertex() const { return numberOfPrimaryVertex; }
   …<skipped>...
};
```

How to design your own persistent objects in ODBMS:
Design the object clustering

- Organize a group of classes which will be accessed simultaneously
  - Use "new" operator with clustering directive: e.g.. HepClusteringHint
  - Use "new" operator with neighboring object
    - e.g.. In the constructor of **G4PEvent**::**G4PEvent**(...)

```
….
aVertex = new (ooThis()) G4PPrimaryVertex(…);
….
```

aVertex will be stored near this G4PEvent object



How to design your own persistent objects in ODBMS:
Design the access patterns

- Decide the primary object(s) to be picked up from the database
- Make a loop of iteration for the primary object

```
ooItr(G4PEvent) pevent_iterator;
pevent_iterator.scan(container);
while (pevent_iterator.next())
{
   // loop for all G4PEvent's in this container…
    int evt_id = pevent_iterator->GetEventID();
    ...
}
```

Iterator for G4PEvent

Using the Iterator

Using the (1st) G4PEvent

- Follow the association for the related objects

```
for ( int i = 0; i < n_pvertex; i++ )  // Loop for all primary vertex in this event
{
  HepRef(G4PPrimaryVertex) pvertex = pevent_iterator->GetPrimaryVertex(i);
  cout << "    No. of particle in the primary vertex: "
       << pvertex->GetNumberOfParticle() << G4endl;
}
```

Returns a smart pointer of the G4PPrimaryVertex

How to design your own persistent objects in ODBMS:
Design the transaction scenario

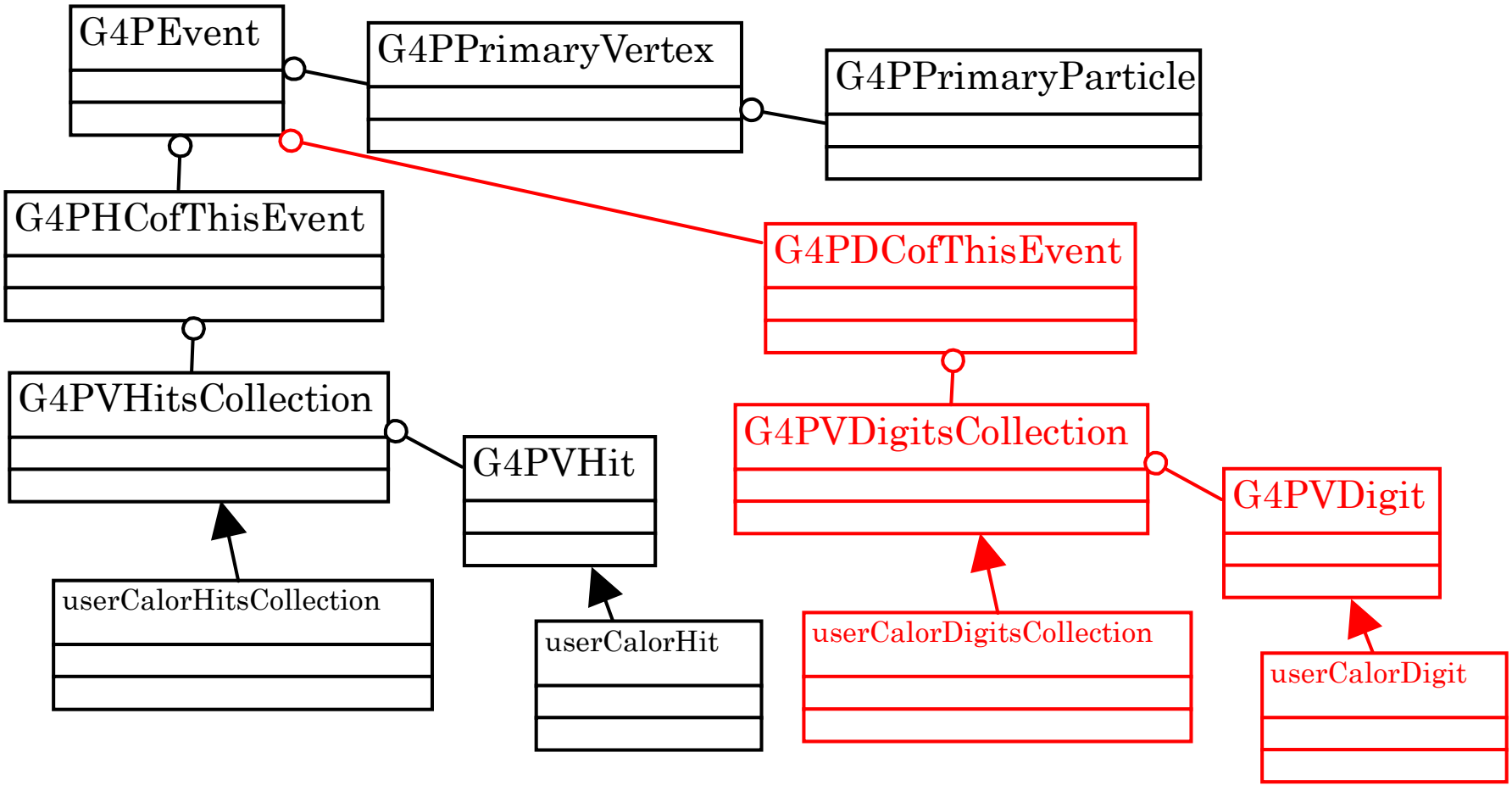
- Access to any persistent objects should be a part of “transaction”
  - HepDbApplication::Init()
  - HepDbApplication::startRead()
  - HepDbApplication::startUpdate()
  - HepDbApplication::commit()
  - HepDbApplication::abort()

- HepODBMS with Objectivity/DB has a choice of selecting “database” and “container”
  - HepDbApplication::db(dbName)
  - HepDbApplication::container(containerName)

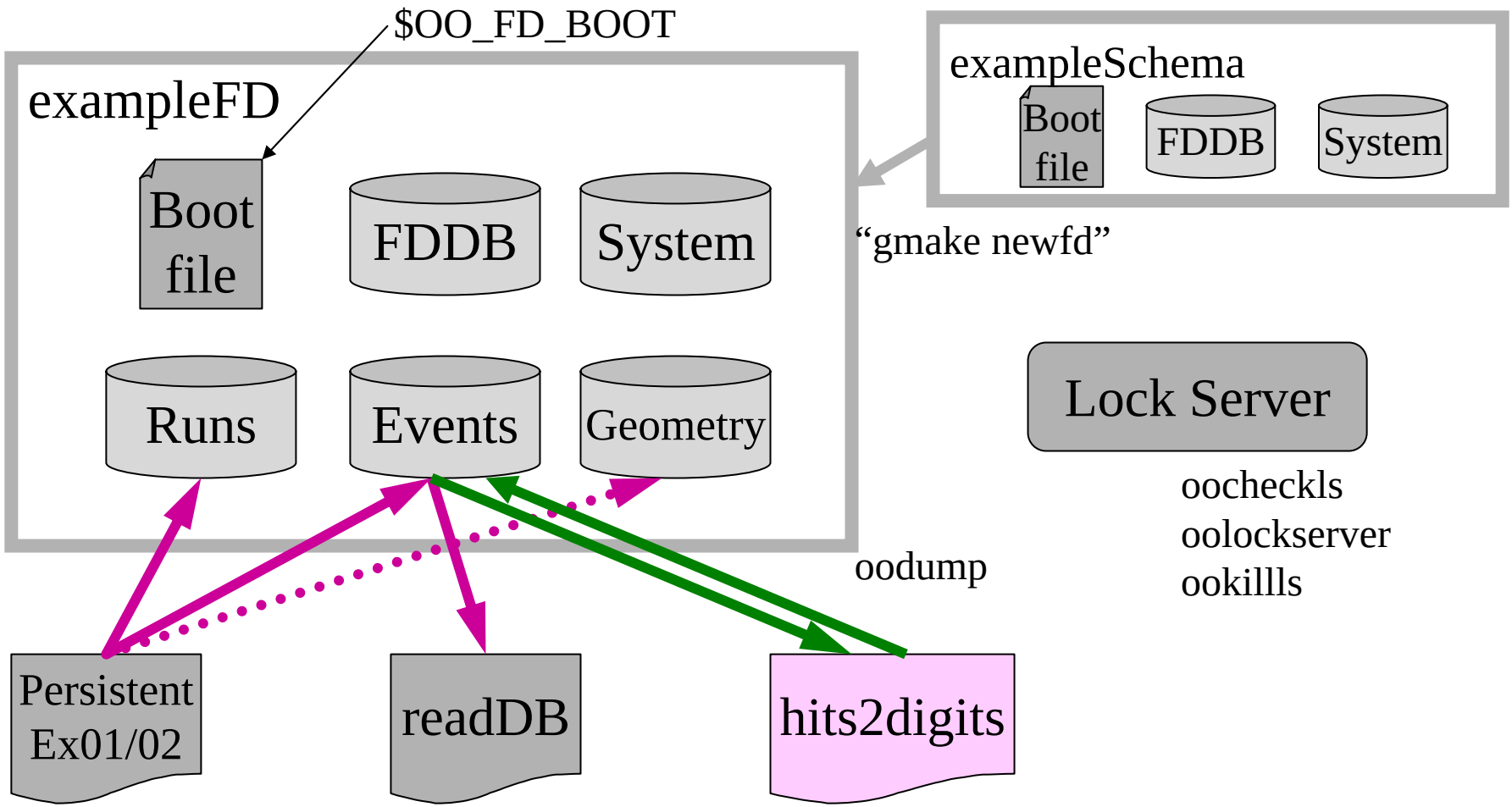
How to design your own persistent objects in ODBMS:
Design the transaction scenario - *readDB.cpp*

```
HepDbApplication* dbApp = new HepDbApplication(name);
  ……..
dbApp->Init();                                                  // initialise the db session
dbApp->startRead();                                             // start a read transaction
HepDatabaseRef  myDb = dbApp->db("Events");                     // select “Events” database
HepContainerRef cont = dbApp->container("EventContainer");      // select “EventContainer” container

ooItr(G4PEvent) pevent_iterator;                                // initialize iterator for G4PEvent
pevent_iterator.scan(cont);
while (pevent_iterator.next())                                  // Loop for all G4PEvent
{
    int evt_id = pevent_iterator->GetEventID();                  // access this G4PEvent
    int n_pvertex = pevent_iterator->GetNumberOfPrimaryVertex();
    ……
}                                                               // End of loop over events

dbApp->commit();                                                // finish this read transaction
```

Persistent Objects in “Events” Database



Example Database Configuration



How to build G4 Persistent Libraries

- Define variables
  - $G4USE_HEPODBMS = 1
  - $G4EXAMPLE_FDID
- Define HepODBMS variables
  - $HEP_ODBMS_DIR
  - $HEP_ODBMS_INCLUDES
- Include HepODBMS and Objectivity library path into $LD_LIBRARY_PATH
- Setup Objectivity variables (e.g.. on CERN AFS)
  - source /afs/cern.ch/rd45/objectivity/objyenv.csh (csh)
  - . /afs/cern.ch/rd45/objectivity/objyenv.sh (bsh)
- Check and start “Lock Server”
- Type “gmake” in $G4INSTALL/source
- See $G4INSTALL/examples/extended/persistency/PersistentEx01/README for more detail (see also the release note for version information)

Geant4 examples illustrating persistency features

Extended examples

- **PersistentEx01**: Make persistent Run/Event/Geometry objects
  - readDB: standalone HepODBMS example to read objects
  - createTag: standalone example to create HepODBMS tag
  - readTag: standalone example to read HepODBMS tag
- **PersistentEx02**: Make user defined persistent Hits objects
  - readDB: standalone HepODBMS example to read objects
  - createTag: standalone example to create HepODBMS tag
  - readTag: standalone example to read HepODBMS tag