

# An Overview of the CDB API

**Igor A. Gaponenko (LBNL)**

IAGaponenko@lbl.gov

# What's in the talk

- **The talk gives a short introduction into CDB API**
- **A primary focus of the talk is on how CDB API handles persistent technology specific implementations of metadata and user defined payload.**
- **It's essential to look at a general CDB talk/paper at CHEP'2004 to understand the background:**
  - **CHEP talk:**
    - <http://www.slac.stanford.edu/BFROOT/www/Public/Computing/Databases/talks/Igor/Chep2004Talk316.pdf>
  - **CHEP paper:**
    - <http://www.slac.stanford.edu/BFROOT/www/Public/Computing/Databases/proceedings/Chep2004Paper316.pdf>

# Some Common Reasoning

- **Persistency is a “cursed” problem of C++**
  - Because of its static type system
  - Persistency is “un-natural” for C++ types
  - All solutions require (quite often, clumsy and hard to maintain) data transformation (RDBMS) or class instrumentations (OO-like)
- **The problem is especially acute for Conditions (and alike) applications because of its time evolving schema for the ser defined database payload**

- **Three major solutions we have:**

- Get stuck with a particular technology/vendor (RDBMS, Objectivity/DB, ROOT I/O, etc.)
- Get stuck with a set of predefined transient containers to be serialized into many persistent technology
- **Play OO design games to reach an “acceptable” compromise**

*An original approach  
of BABAR*



*The current  
CDB approach*



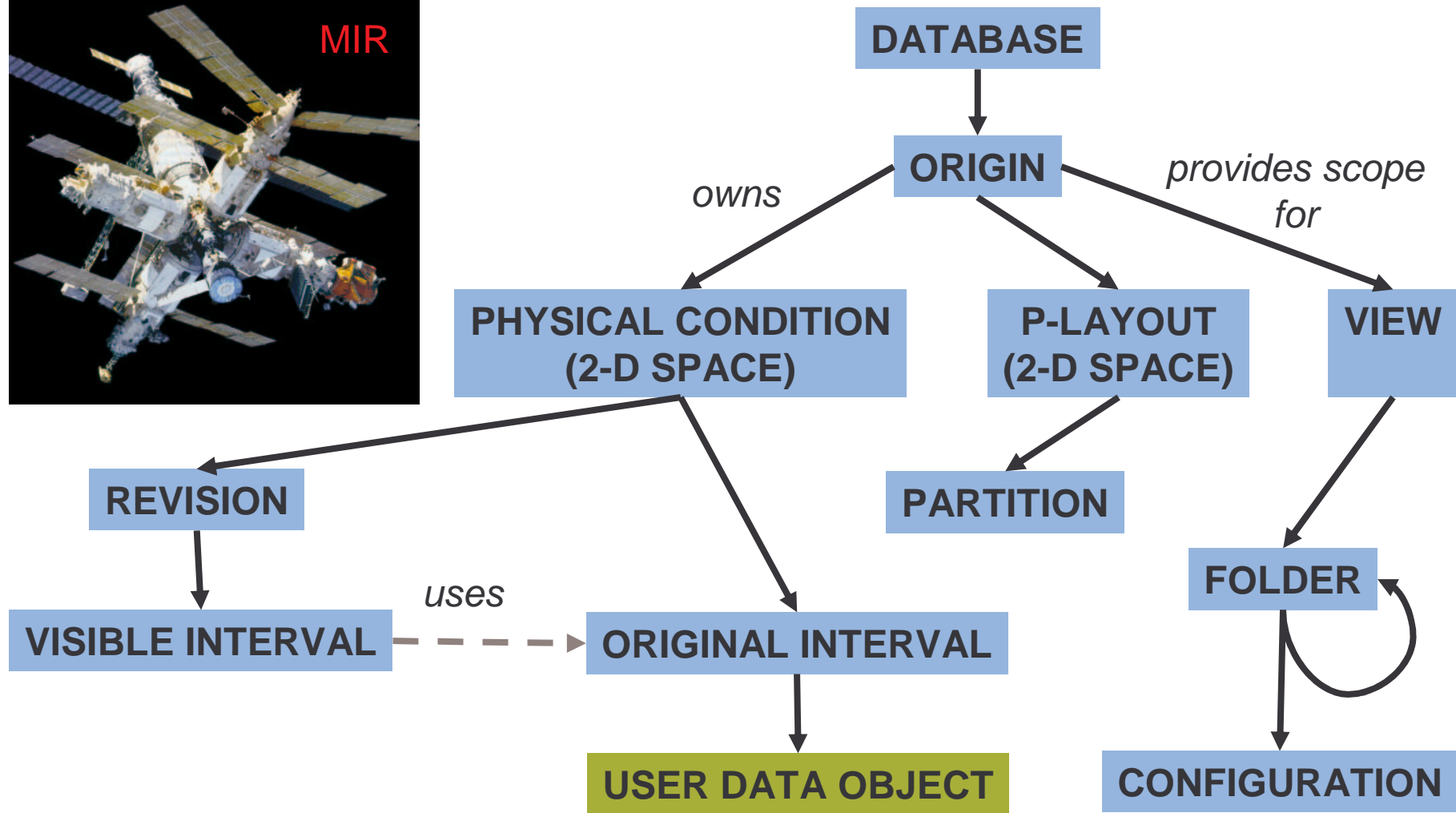
# CDB API Highlights

- **The current CDB API provides proper modeling of the fundamental concepts of CDB in a form of interfaces, abstract classes, etc.**
- **It does not exhibit any persistent technology for API components representing metadata (95% of all classes)**
- **Provides converters (“bridges”) and factories to deal with technology specific user defined objects (payload) for specific TECHNOLOGIES (Objectivity/DB OID-s, etc.). Converters are used to:**
  - Get an actual persistent object
  - To store a newly created persistent object into CDB
- **Provides support for multiple persistent TECHNOLOGIES and IMPLEMENTATIONS (for a given TECHNOLOGY) of the abstract API**
  - An **IMPLEMENTATION** is a sort of **plug-in**
  - The implementations must be statically linked with an applications (no dynamic loading yet)
  - A choice of which one to use is a subject for a job’s configuration or it (the choice) is made automatically at a presence of external stimulus (environment variables)

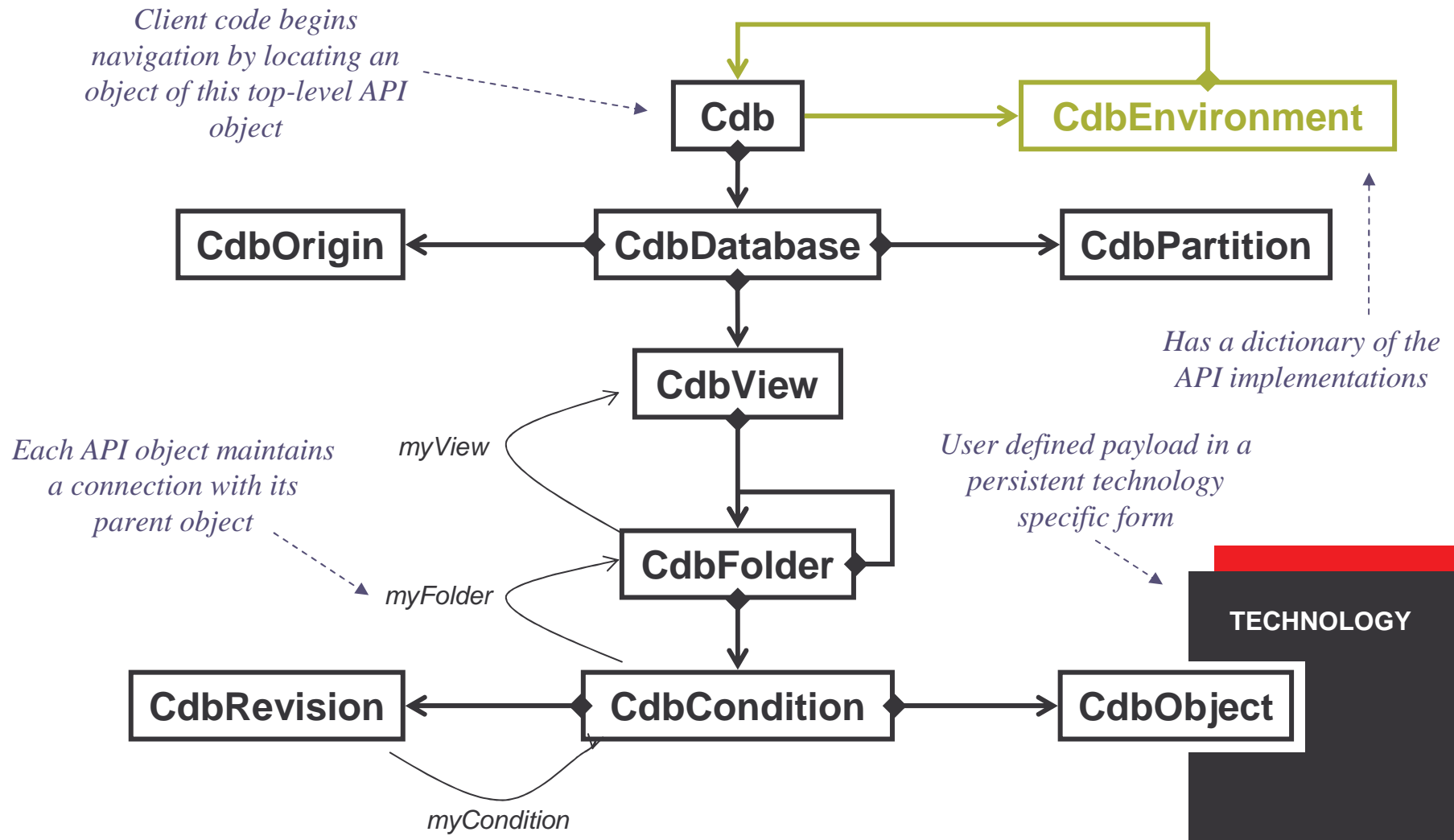
# More CDB API Highlights

- **Properly handles defaults to answer the following questions:**
  - What's the default **TECHNOLOGY**?
  - What's the default **IMPLEMENTATION** for given **TECHNOLOGY**?
  - What's the default **DATABASE** for given **TECHNOLOGY**?
  - What's the default **VIEW** for given **DATABASE**?
- **All API components are made available to clients via COUNTED SMART POINTERS**
  - We need pointers to handle multiple implementations (plug-ins)
  - We want to make life of CDB API clients a bit easier
- **Provides users with two ways to navigate across CDB API components:**
  - A “standard” one:
    - **TECHNOLOGY <> IMPLEMENTATION <> DATABASE <> VIEW <> FOLDER : CONDITION <> OBJECT**
  - A “shortcut” one:
    - **CONDITION <> OBJECT**
    - It's the most useful way to deal with the API when all defaults are known
- **Strict control over the placement and clustering of the user defined objects (payload). See next slides for more details...**

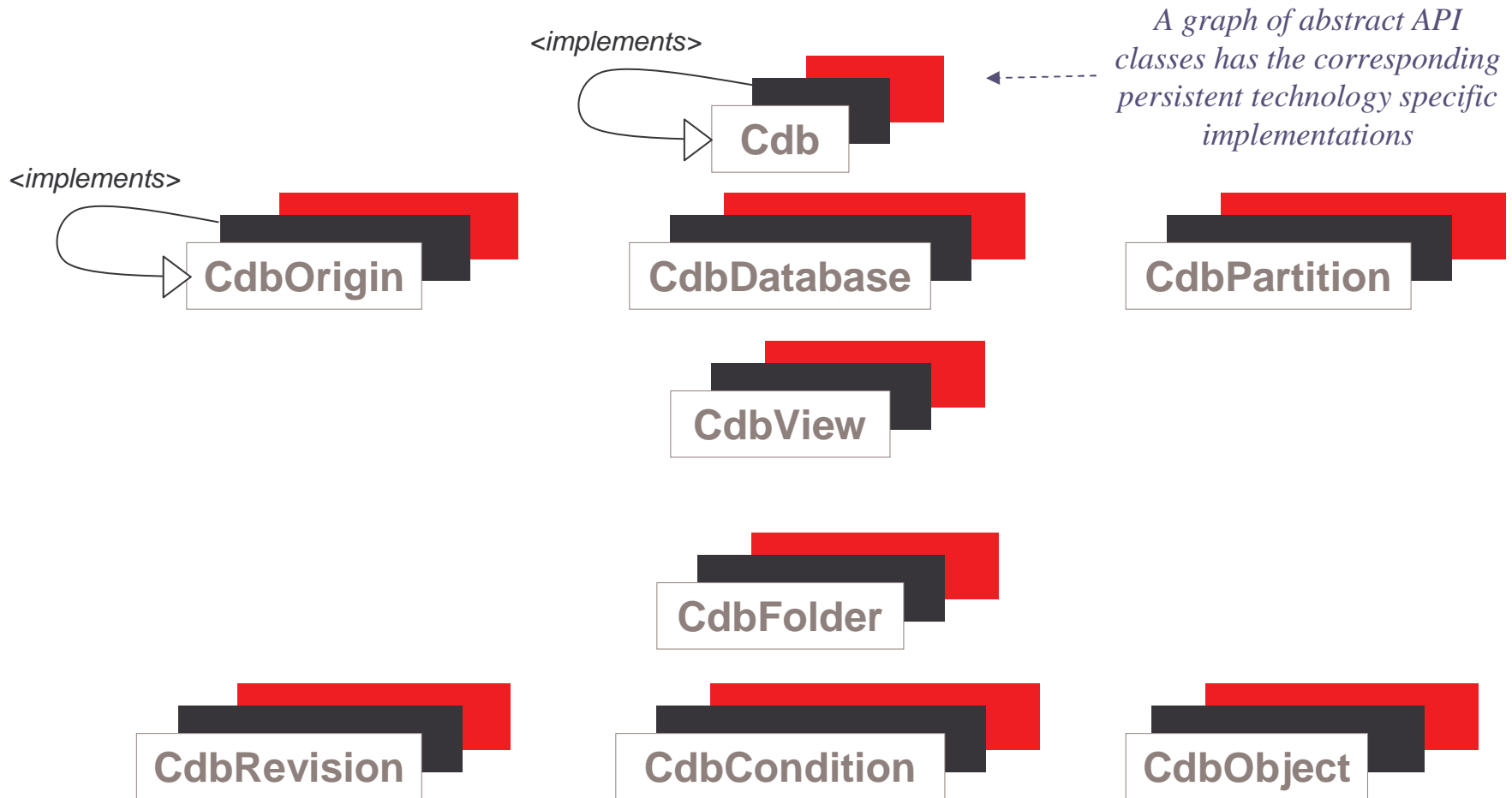
# CDB Concepts : Scope & Ownership Diagram



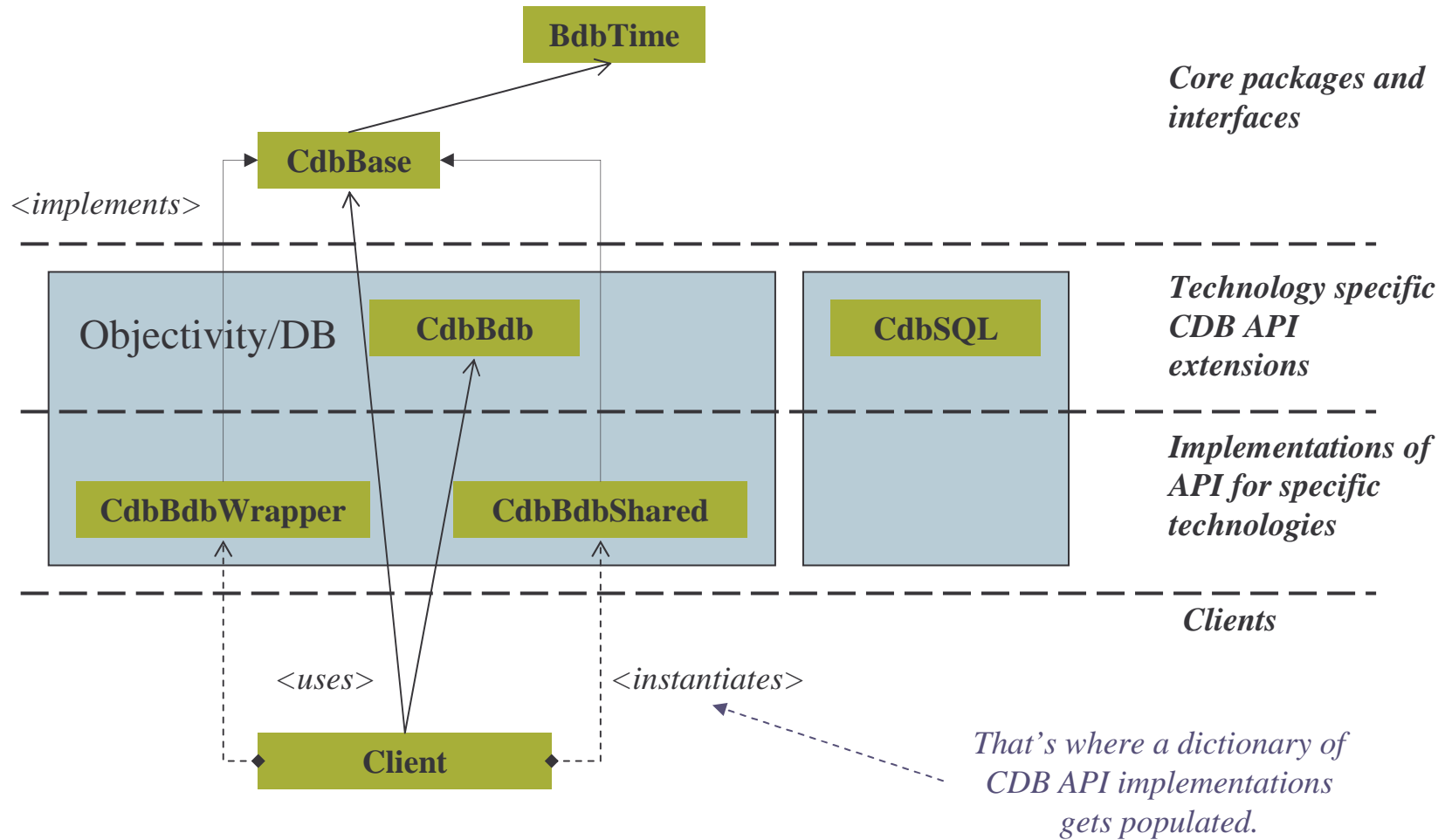
# Metadata: Technology-independent API



# Metadata: Implementations



# API : Packaging



# 3 Roles of the *Cdb* class

```
class Cdb {  
public:  
  
    // A locator of the top-level API object  
  
    static CdbPtr instance( const char* technology = 0,  
                           const char* implmenation = 0 );  
  
    // Parameters of the implementation  
  
    virtual const char* technologyName ( ) const = 0;  
    virtual const char* implementationName( ) const = 0;  
  
    // Find a database  
  
    virtual CdbStatus findDatabase( CdbDatabasePtr& ptr,  
                                   const char* name = 0 ) = 0;  
};
```

1: The locator would interact with an internal dictionary to find out a top-level API object matching requested criteria. Default values mean default object which is a subject of an application's configuration. Note, that the method returns a **counted smart pointer** onto an object of the *Cdb* class. This is a **typedef** for something like `boost::shared_ptr<Cdb>`.

2: These two methods is the only link to an underlying persistent technology specific implementation of the abstract API available to client applications .

3: Zero database name means the default database installation for the current application.

# Metadata: Navigation

- **There are two methods to navigate in the graph:**
  - “**standard**” : a full path starting from an object of *Cdb* class down to an object of *CdbObject*
  - “**shortcut**” : a quick way of getting to an object of *CdbObject* or storing a new object in the database
- **Why the “shortcut” is needed?**
  - It appears that 95% of users do 2 very simple operations with the API:
    - **CASE I:** find a *condition object* from a *condition* at a *validity time* assuming the current *configuration* of the condition in the corresponding *view*.
    - **CASE II:** create a new *condition object* and store it with the specified *validity interval* at the specified *condition*.
  - For them the API can be and must be very simple.
  - For the rest of 5% uses:
    - People prefer using “trusted-out-of-box” interactive tools to manage the database contents! No fancy operations (like using iterators of intervals) are done programmatically. All (well, most) of this is utilized by the interactive tools internally.

# Metadata: “Standard” Navigation

```
CdbPtr api = Cdb::instance( "Bdb", "Shared" ); // Tech & Impl
```

```
CdbDatabasePtr database;  
api->findDatabase( database,  
                  "OO_FD_BOOT" );  
  
CdbViewPtr view;  
database->findView( view,  
                   "<recent>" );  
  
CdbFolderPtr folder;  
view->findFolder( folder,  
                 "/DAQ/Trigger3/" );
```

```
CdbConditionPtr condition;  
folder->findCondition( condition,  
                      "Config" );
```

```
CdbObjectPtr object;  
condition->findObject( object,  
                       BdbTime( "15-NOV-2004 16:30:01" ) );
```

```
cout << "FOUND OBJECT ADDRESS : " << object->id      () << endl  
      << "                STORED : " << object->stored() << endl  
      << "          VALIDITY BEGIN : " << object->begin () << endl  
      << "          VALIDITY END   : " << object->end   () << endl;
```

*This “step-by-step” navigation process is very helpful when multiple operations over the contents of the database are to be performed at different levels of the containment hierarchy.*

*Each level is represented by a counted “smart” pointer onto the corresponding API object.*

*Each API object (but Cdb) has a parent link back to its upper level “context” API object (not shown on this example).*

*An over-simplified example: methods should return completion statuses of operations.*

# Metadata: “Shortcut” Navigation

```
CdbConditionPtr condition;
Condition::instance( condition,
                    "/DAQ/Trigger3/Config",
                    "<recent>",
                    "OO_FD_BOOT",
                    "Shared",
                    "Bdb" );
```

*That's essentially the same sequence of actions as it was illustrated for the “standard” method on the previous slide.*

...or...

```
CdbConditionPtr condition;
Condition::instance( condition,
                    "/DAQ/Trigger3/Config" );
```

*Though, things may look even simple, in case when a client would agree with default values of other parameters.*

*These defaults are put into a special configuration dictionary represented by the **CdbEnvironment** class.*

*That's actually how **MOST OF BABAR CLIENTS' CODE IS WRITTEN!!!***

```
CdbObjectPtr object;
condition->findObject( object,
                      BdbTime( "15-NOV-2004 16:30:01" );
```

...

# Metadata & Payload

- **An important question:**
  - How do we get from the *metadata* (interval) to *payload* (data)?
- **The CDB answer:**
  - *Logically* it's a one way road
  - At a technology-neutral level the payload exhibits some form of ID (“address”) to reflect this idea
- **The CDB API:**
  - Does NOT really assume that metadata and payload are stored **separately** (as separate objects, tables, etc.). It's up to a particular implementation to establish this connection in a most efficient way for the corresponding persistent technology to be used.
  - Has provisions for technology specific API implementations to **force** putting new payload into a desired location to **ensure** that metadata and the corresponding payload were stored within the same unit of the data distribution (file, table, cluster, etc.).
- **Goals:**
  - Efficient Data Distribution
  - Robustness, referential integrity, etc.

# CdbObject

- It's a “bridge” between metadata and the actual payload
- What's in CdbObject:
  - “original” validity interval
  - “visible” validity interval
  - “stored” time of the payload object
  - “address” of the payload object
- A persistent technology specific persistent data object can be *extracted* from CdbObject using *CONVERTOR* facilities supplied with the underlying implementations of the CDB API
  - It's a sort of a type-safe “cast” which would assure that the *actual* persistent technology behind CdbObject matches the one *expected* by the **CONVERTOR**. And if so then it would use information stored in the underlying implementation of CdbObject into a technology specific object of the requested type.

# Implementations of CdbObject: Objectivity

```
// FILE: CdbBdb/CdbBdbObject.hh

// TECHNOLOGY: "Bdb" (Objectivity/DB)

#include "CdbBase/CdbObject.hh"

class CdbBdbObject : public CdbObject {
public:
    ...
    // An object address
    virtual std::string id( ) const { return _handle.sprintf( ); }

    // Get to the payload object
    ooRef(ooObj) ref( ) const { return _ref; }

private:
    ooRef(ooObj) _ref; // a persistent object reference
};
```

# Using CONVERTOR-s: Objectivity

```
// FILE: MyPackage/MyClient.cc

// TECHNOLOGY: "Bdb" (Objectivity/DB)

#include "CdbBase/CdbObject.hh"
#include "CdbBdb/CdbBdbObjectConvertor.hh"

// Get to the metadata object first

CdbObjectPtr object;
...

// Get to the payload

ooHandle(ooObj) ojectH;
if( CdbStatus::Success != CdbBdbObjectConvertor::narrow( objectH,
                                                         object )) {
...

```

*Use this (Objectivity/DB) technology specific conversion facility to extract a persistent reference onto a payload object.*

## Storing new payload (objects)

- **Two problems here:**
  - How to separate a persistent technology specific payload creation from the technology-neutral CDB API?
  - How to control the placement of newly created objects?
    - That's a very important aspect of a distributed database.
- **CDB API answer is:**
  - Confine an actual object creation code into so called “**FACTORIES**” of persistent objects
- **What's the **FACTORY**?**
  - It's a user transient class prepared by a user. The class derives (directly or indirectly) from a special base class supplied by CDB API in order to implement a persistent class and persistent technology specific object creation sequence at a suggested by CDB API location.
  - See the next slide for more details...

# Base FACTORY

```
// FILE: CdbBase/CdbObjectFactoryBase.hh
```

```
class CdbObjectFactoryBase {  
protected:  
  
    // Create a new object for specified validity interval  
  
    virtual CdbStatus create( CdbObjectPtr&          object,  
                             const CdbConditionPtr& condition,  
                             const BdbTime&         begin,  
                             const BdbTime&         end ) = 0;  
};
```

```
// FILE: CdbBase/CdbCondition.hh
```

```
class CdbCondition : ... {  
public:  
  
    virtual CdbStatus storeObject( CdbObjectFactoryBase& factory,  
                                   const BdbTime&         begin,  
                                   const BdbTime&         end ) = 0;  
};
```

An underlying persistent technology implementation of the *CdbCondition* class would invoke the above defined “*::create*” method of the factory.

# Generic FACTORY

```
// FILE: CdbBase/CdbObjectFactory.hh

#include "CdbBase/CdbObjectFactoryBase.hh"

template< class PRODUCT,      // a type of a (persistent) product
          class HINT,        // a placement for the product
          ... >
class CdbBObjectFactory : public CdbObjectFactoryBase {
protected:

    // Create a new object for specified validity interval (IMPLEMENT)

    virtual CdbStatus create( CdbObjectPtr&          object,
                             const CdbConditionPtr& condition,
                             const BdbTime&         begin,
                             const BdbTime&         end );

    // Invoke a user defined creation sequence

    virtual CdbStatus doCreate( PRODUCT&    product,
                               const HINT& hint ) = 0;
};
```

# Implementing FACTORY: Objectivity

```
// FILE: CdbBdb/CdbBdbObjectFactory.hh
```

```
#include "CdbBase/CdbObjectFactory.hh"
```

```
typedef CdbObjectFactory < ooHandle(ooObj), ooRefAny > CdbBdbObjectFactory;
```

```
// File: CdbBdb/CdbBdbTObjectFactory.hh
```

```
template< class TRANSIENT,  
          class PERSISTENT >  
class CdbBdbTObjectFactory : public CdbBdbObjectFactory {  
public:  
    CdbBdbTObjectFactory( TRANSIENT t ) : _t(t) {}  
protected:  
    virtual CdbStatus doCreate( PERSISTENT& product,  
                                ooRefAny& hint )  
    {  
        product = new( hint ) PERSISTENT( _t );  
        return CdbStatus::Success;  
    }  
private:  
    TRANSIENT _t;  
};
```

*This is an “out-of-box” solution for those cases when a trivial TRANSIENT-to-PERSISTENT conversion exists.*

# Using the FACTORY: Objectivity

```
// FILE: MyPackage/MyClient.cc

// TECHNOLOGY: "Bdb" (Objectivity/DB)

#include "CdbBase/CdbCondition.hh"
#include "CdbBdb/CdbBdbTObjectConvertor.hh"

#include "MyPackage/MyTransientClass.hh"
#include "MyPackage/MyPersistentClass.hh"

// Get to the CdbCondition API object first

CdbConditionPtr condition ...;

// Instantiate a factory for a pair of classes

MyTransientClass transient( ... );
CdbBdbTObjectFactory< MyTransientClass,
                    MyPersistentClass > factory( transient );

// Store a new object (the factory will be used)

if( CdbStatus::Success != condition->storeObject( factory, ... ) ) ...
```

# Need more information?

- **A successful attempt of using predefined transient containers (tables) for storing user data**
  - <http://www.slac.stanford.edu/BFROOT/www/Public/Computing/Databases/experts/docGenericNTuplesAPI.html>
- **DOXYGEN generated documentation on Cdb\* packages:**
  - <http://www.slac.stanford.edu/BFROOT/www/Public/Computing/Databases/srcDocs/index.shtml>
- **BaBar CVS repository:**
  - <http://babar-hn.slac.stanford.edu:5090/cgi-bin/internal/cvsweb.cgi/>