SCHEMA EVOLUTION STRATEGY FOR THE CONDITIONS DATABASE

April 21, 1999

(BaBar Databases Group)

What's new in this version of the document

1. Changes through March 8, 1999

This document overrides and extends the original version of the document as of March 8, 1999.

1.1 Modifications

Section **II.2.2** has been slightly modified. In an example of simple proxy all persistent classes' castings which were previously done through:

```
myObjectH = (BdbHandle(MyPersClass)) objectH;
```

have been replaced by:

```
myObjectH = (BdbHandle(MyPersClass)&) objectH;
```

which does not require a temporary object to be created. There is some experience indicating that creating a temporary objects at OSF1 platform may lead to crashes in this piece of code.

Section **IV** has been renumbered as **V**.

1.2 Extensions

A new section **IV.1** has been added. This section considers a possible design option relocating most of the persistent-transient data conversion out of a proxy into a persistent class itself. This allows to make proxies more stable, or virtually unchangeable with the persistent schema evolution. Another benefit of this approach is that the class's version-specific data conversion is bound to the class itself, providing more robust code management.

I. Introduction

As understanding of the detector, electronics and software "evolves" in a course of an experiment it necessitates a similar evolution to the data structures and classes (both persistent and transient) used in the experimental data acquisition and processing systems.

This implies some extra requirements to the BaBar database software in general, and the Conditions database (as well as all the Conditions-like databases) software in particular.

From a practical point of view there are two most typical use cases which have to be satisfied with a schema evolution solution:

- FORWARD compatibility of the old (existing) binaries with new schema.
- BACKWARD compatibility of the new binaries with the old data.

The first means that existing (old) binaries should be run against a new federation (with evolved schema) without a re-compilation or rebuilding. In other words this means a schema compatibility.

> **NOTE**: This capability extends only to the ability to access old data, not new data.

The last one means that new binaries should be able to read and correctly interpret existing data (written with old schema).

Our recent experience with "out of box" Objectivity/DB schema evolution shows that it falls short in satisfying these criteria. This is why we've chosen to implement their own schema evolution strategy.

We call it "strategy" for this is mostly a (logically related of course) collection of rules to follow rather then an API or a special set of tools.

The main idea in the schema evolution strategy (or just the Strategy) presented here is that, in a course of the evolution process, a new schema is always a true super-set of an old one. We control the evolution process by applying some rules describing how to modify and how to use this schema.

Given the persistent OO model implemented by Objectivity/DB (TM) software, this was an easiest, if not to say the only, way to meet the above stated goals.

Another goal behind this strategy was to minimize and localize as much as possible changes in users's code, both in respect of changes to be done right now (in order to accommodate the proposed evolution strategy) and in the future (in a process of a real schema evolution itself).

The subsequent chapters give a complete description of this strategy.

II. Schema Evolution Strategy

An explanation of the Strategy presented in this chapter falls onto three domains of rules describing how:

- to create a persistent schema to comply to the Strategy;
- to store new conditions objects in the database;
- to retrieve objects back from the database.

Also we leave more discussion on the possible side effects and implications of using transient classes as base classes and/or data member classes for the persistent ones to the next chapter. Since we can't dictate the design solutions to the schema developers we give them a list of recommendations how to make a design of persistent and transient classes more evolution-safe.

1. Persistent schema

Here is a logical sequence of rules we follow in implementing persistent schema in according to our Strategy:

- We deny the use of Objectivity/DB schema evolution. We do this by explicitly disabling the corresponding option of DDL compiler. Any attempts to evolve (in terms of Objectivity/ DB) existing persistent class will be treated as compilation errors.
- We implement their own evolution strategy.

  NOTE: Here and after we substitute the default use of the term of *evolution* as it's implemented by Objectivity/DB by our interpretation of this, unless it's stated explicitly.

- The basic statement in our strategy is that a new schema is always a super-set of existing one.
- The previous statement means that every time we need to evolve a persistent class - we create a new persistent class. This new class logically inherits from the old one, but in respect of Objectivity/DB persistent schema this is completely new class.
- In order to reflect this logical dependency (or *evolution*, as we now interpret this word), we put some restrictions (requirements) to the names of these classes. Here are these conventions:
  - All the classes in a class's evolution history have the same root name. Suppose it's:

    `MyPersClass`

  - We recommend/require using an existing (un-evolved) class name as the root name.
  - Every time the persistent classes evolves - the class's version number is concatenated in the end of the root name, like this:

    `MyPersClass_001`
    `MyPersClass_002`

  - We require the version number to increment monotonically. This will be useful in the future when we have some automation tools to deal with versions.
  - We require the version number to appear exactly as it's shown in the previous example, i.e. made from underscore and 3 digits, where the least significant one comes last (This is exactly as in *printf("%3.0d")* format string.).

- All of the evolved classes must be in the same package as the original one is.

2. Dealing with evolved persistent classes

2.1 Storing new objects in the database

The primary (low level) interface to store the conditions data is the following method of *BdbDatabase* class:

```
::store( BdbHandle(BdbObject)& obj, const char* containerName,, )
```

There is nothing special here in connection with the schema evolution strategy, but the only requirement:

- All the new objects of the original class and their logical descendants must go into the same container. In case of the mentioned before original *MyPersClass* class and their versions, this would always be:

```
"MyPersClass"
```

This means however that all the objects of the evolved class will be kept within the same historical line.

EXAMPLE:

```
BdbDatabase theDb( "xxx" );
BdbCondClusteringHint theHint( "xxx" );
...
BdbHandle(MyPersClass) obj1;
BdbHandle(MyPersClass_001) obj2;
BdbHandle(MyPersClass_002) obj3;
...
obj1 = new(theHint.updatedHint()) MyPersClass(...);
obj2 = new(theHint.updatedHint()) MyPersClass_001(...);
obj3 = new(theHint.updatedHint()) MyPersClass_002(...);
...
theDb.store( obj1, "MyPersClass",,, );
theDb.store( obj2, "MyPersClass",,, );
theDb.store( obj3, "MyPersClass",,, );
```

2.2 Fetching objects from the database

Fetching persistent objects is a little bit more tricky than storing them, though the low-level interface to the Conditions database, defined through BdbDatabase class, remains the same:

```
::fetch( ,BdbHandle(BdbObject)& obj, const char* containerName,, )
```

The first rule we introduce here with the new strategy requires:

- The objects of different versions of a class are accessed by specifying the same container name (as in case of storing them. See previous section).

This rule is a symmetric one to the storing interface. It means that all the objects of the same evolution line (sequence of persistent class versions) are fetched from the same historical line.

The "assymetric" side is that in the current organization of the BaBar data processing software the conditions data are accessed by users (see a definition of a "user" below) via an intermediate "proxy" layer, which is a collection of special transient classes ("proxies") hiding the low-level interface.

> **NOTE**: In a terminology used throughout this document, a "user" is defined as person who is not responsible for developing persistent schema. He could (in theory) be even unfamiliar with the details how a persistent "world" is organized. This is an alternative to a so called "developer" - who represents a main auditorium this document is aimed at.

The functions of the existing proxies are:

- Fetching persistent data from a database (thus hiding the low-level Conditions DB interface from users' code).
- Creating an object of a transient class to be filled with the information taken from the persistent classe(s).

  > **NOTE**: Typically a proxy serve as *one-to-many* interface, what means that one proxy may create a transient object which carries the information from one or more persistent ones.

- Caching some persistent information in order to reduce a number of "real" calls to the Objectivity/DB API.

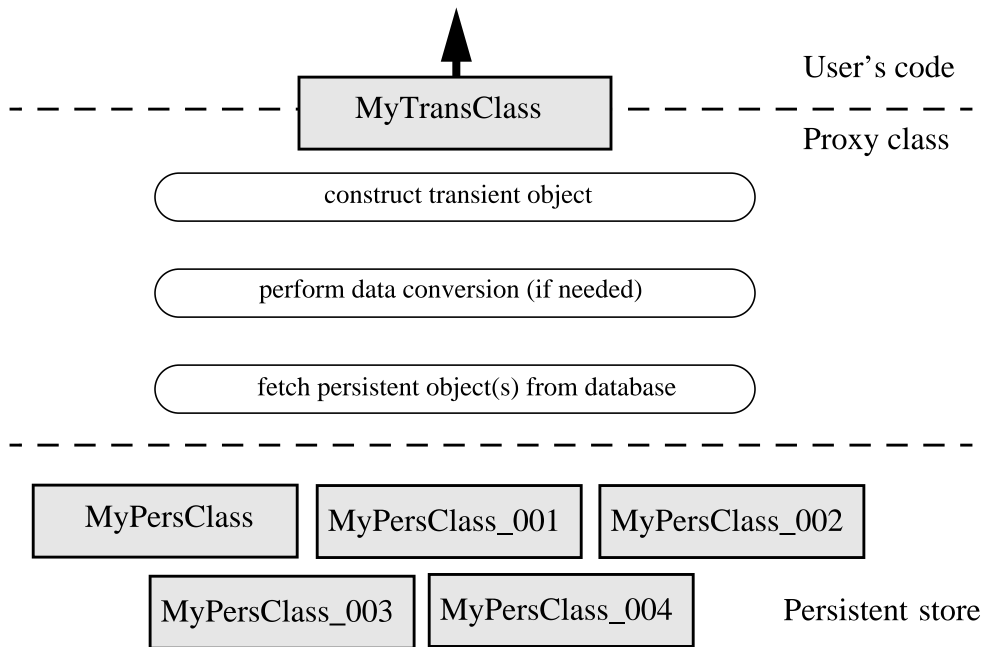
Given the schema evolution strategy presented here, we extend the functionality, but leave the semantics of proxies unchanged.

Here is what we are adding to proxy functionality:

- A proxy is responsible for the determining the actual class (version) of each persistent object fetched through *BdbDatabase::fetch()* interface.
- A proxy reports errors and returns a zero reference to a transient object upstream to users' code if it sees an object of unknown type. This also includes further evolutions of persistent objects.
- A proxy is responsible for down casting of a handle to *BdbObject* (which is expected by *BdbDatabase::fetch()* interface) to a handle of a specific class fetched.
- A proxy is responsible for an interpretation of data from both the most recent version of a class known to the proxy and all the previous (including the original one) versions.

  > **NOTE**: In our terminology we call this procedure *data conversion*.

The graphical view of this three-layered model is presented on FIGURE_1 below.



Here are additional rules to be added to the Strategy in connection with proxies:

- Each time a new version of a persistent class is created - a corresponding implementation of a proxy, which is aware of this class, must be done.
- The Strategy does not implies any special requirements or naming conventions to the names of proxies.
- The proxy class must be in the same package where the corresponding evolved persistent classes are.

Please note that in the current definition of the Strategy we don't require a persistent class and proxy developers to pass any information on an actual version(s) of persistent class(es) to be fetched. This is left up to developers. A reason why we do it is that such a requirement could affect existing users' code. However we do recommend that developers introduce a more sophisticated mechanism to pass the information about the "quality" of the data conversion from a proxy upstream to user's code.

> **NOTE**: "*More sophisticated*" means more then just a boolean one (a zero or non-zero pointer to a transient object).

Given all the above it may be seen that proxies are the only (main?) place where the knowledge about a particular class's (or classes) schema evolution is incorporated. This is, in particular, why we require to keep the proxy class's code along with corresponding persistent classes in the same package.

In the following example a fragment of a very trivial proxy is shown. This piece of pseudo-code illustrates the data conversion logic. A "fault handler" of the proxy constructs and returns back to a user an instance of *MyTransClass* which (ideally) should be independent of a class of the fetched persistent object.

```
MyTransClass*
MyProxy::faultHandler(...)
{
    MyTransClass* objectT = (MyTransClass*) 0;
  // ** Fetch an object satisfying specified criteria(s)
    BdbDatabase theDb( "xxx" );
    BdbHandle(BdbObject) objectH;
    theDb.fetch( objectH, "MyPersClass",, );
  // ** Perform "data conversion":
  //   1. Determine object class
  //   2. Perform down casting
  //   3. Construct a transient object
  //
    if( ! strcmp( "MyPersClass_002", objectH.typeName())) {
        BdbHandle(MyPersClass_002) myObjectH;
        myObjectH = (BdbHandle(MyPersClass_002)&) objectH;
        objectT = new MyTransClass(...);
    } else if( ! strcmp( "MyPersClass_001", objectH.typeName())) {
        BdbHandle(MyPersClass_001) myObjectH;
        myObjectH = (BdbHandle(MyPersClass_001)&) objectH;
        objectT = new MyTransClass(...);
    } else if( ! strcmp( "MyPersClass", objectH.typeName())) {
        BdbHandle(MyPersClass) myObjectH;
        myObjectH = (BdbHandle(MyPersClass)&) objectH;
        objectT = new MyTransClass(...);
    } else {
        cout << "Unknown class: " << objectH.typeName() << endl;
    }
    return objectT;
}
```

The technique used in this code to determine an actual class name of a persistent object is provided by Objectivity/DB API. See [OBJYC++] for the details.

An alternative way of doing the type comparison is based on the Objectivity's *type number*. See [OBJYC++].

We recommend:

- using the type number approach rather then the strings comparison approach. It should be faster.

Similar example of code may be found in the recent version of BdbCondTests package in the BaBar software releases area.

III. Persistent and transient classes

There is another design issue we want to discuss in this chapter which could potentially break down the evolution of a particular persistent class. It's about special dependency between transient and persistent classes in the Objectivity/DB schema.

This kind of dependency appears in the following situations:

- when a transient class is one of the base classes of a persistent one

or

- when a persistent class has data members of transient classes.

In both cases the shape of the resulting persistent class directly depends on the shape of the corresponding transient one(s).

Technically, as it's implemented in Objectivity/DB, each transient class involved into such a dependency relations is assigned a separate type number and the information about its shape is placed into the federation's schema.

As a result any changes done to the shape of the transient class lead to the change of the shape of the persistent one.

Let's consider this on the following example. In this example we use a transient ClassT as a base class of a persistent class ClassP and all their subsequent versions.

```
// file: ClassT.hh
class ClassT {
    d_ULong data;
};

// file: ClassP.ddl
#include "ClassT.hh"
class ClassP : public ClassT, public ooObj {
    ...
};
```

This code will be successfully compiled by DDL compiler. Then we will slightly modify a shape of ClassT class and create a new persistent class (according to the Strategy).

```
// file: ClassT.hh
class ClassT {
    d_ULong data;
    d_Float moreData;
};
// file: ClassP_001.ddl
#include "ClassT.hh"
class ClassP_001 : public ClassT, public ooObj {
    ...
};
```

The resulting code of both persistent classes will not compile by DDL (if Objectivity/DB

schema evolution is disabled) because the shape of ClassT is already known to the schema, even if we've changed the name of the persistent class.

Here is the right time to consider possible solutions to these problems, which again are mostly recommendations and design limitations. Here are the most trivial recommendations:

- Ideal solution would be to avoid using transient classes in persistent class definition wherever it's possible, even if it's allowed by a syntax of DDL.
- Use those transient classes which will never (for sure) evolve. For instance a class representing a complex value, for there is nothing new to add there.
- Each time the shape of a transient class is to be evolved - create a new transient class with another name. In fact this is exactly the same way the evolution of persistent classes should go according to the Strategy.

Unfortunately in the reality none of the mentioned above methods alone (or all together) could directly solve the dependency problem without putting additional limitation on the use of such transient classes in the context of BaBar software organization. The most problematic one is to follow the last recommendation despite it seeming to eliminate any dependency. This is because there is another kind of a dependency we could take into account - a dependency between the transient class and the rest of the BaBar software. In fact we've managed to get rid of this dependency by using proxies, which serving as a sort of transient interface to s persistent database.

Given this idea of performing actual data conversion between the persistent and transient "worlds" inside proxies, the desired limitation allowing avoidance of the second kind of dependency is:

- Not to expose such transient classes to the rest of user's code. It could be done by administratively disabling modification of the shape of these classes (a question to the package coordinators). In fact this rule could be also translated as - disable using these classes by the rest of the BaBar code.
- Keep these transient classes in the same package where persistent ones are. If possible.
- Do not use transient classes from other packages in a definition of the persistent classes if they (transient) classes could be a subject of modification by somebody else. In other words - we should keep the full control over this.

A compromise design solution we are recommending for those cases when the same transient class is already involved in both kinds of dependences is:

- To have two different transient classes - one for the users's code and another one for the persistent "world".
- Evolve the second transient class in a similar way the persistent classes do. The only difference here is that we do not put here any special naming requirements and leave this up to the developers.
- Perform desired *transient-to-transient* conversion inside proxies.

The price of this is that it unfortunately requires some extra work to be done to existing proxies, as well as some unavoidable code duplication. Here we sacrifice the time of persistent schema developers to the global and unpredictable changes in the rest of BaBar software above.

IV. Specific design patterns

This part of the document studies, but does not put as a requirement, various design patterns which might be found useful by developers. Each pattern is a self-consistent setup of transient and persistent classes.

1. Data conversion in a persistent class

In this section we would like to consider an option which is based on an idea of relocating most of the persistent-transient data conversion out of a proxy into a persistent class itself. This allows to make proxies more stable, or virtually unchangeable, with the persistent schema evolution. The proxy is left a role of a "dispatcher" of a proper convertor.

A benefit of this approach is that the class’s version-specific data conversion is bound to the class itself, providing more robust code management.

The idea is illustrated with a simple code setup. The setup is made from the following components:

- A transient class *MyTransClass* which is meant to be a carrier in a data exchange process between persistent classes and the proxy. Since this class does not contribute into shapes of persistent classes it may be freely changed throughout the evolution process.
- An existing persistent class *MyPersClass* which is meant to be evolved. The shape of this class is left intact. The conversion method *::transient()* is added. This makes the class compartible with the their logical evolutions.
- A base persistent class *MyPersClassBase*. The class defines a pure virtual interface for the persistent-transient conversion method of *::transient()* as well as the persistency to derived class. Each evolved class is a direct sub-class of this base class. This is an abstract base class which will never be instantiated.
- Two evolved persistent classes: *MyPersClass_001* and *MyPersClass_002*. One of the important things about these classes is that each of them defines a trivial virtual function of *::size()* which is meant to be called by proxy to ensure the corresponding virtual table is loaded.
- A proxy class *MyProxy*. This class is also responsible for loading of the proper virtual tables for the all persistent classes. This action is done just once in the proxy’s constructor.

The code of the transient "carrier".

```
class MyTransClass {
public:
    MyTransClass( d_ULong data, d_ULong data_001=0, d_ULong data_002=0 ) {
        _data     = data;
        _data_001 = data_001;
        _data_002 = data_002;
    }
private:
    d_ULong _data, _data_001, _data_002;
};
```

The code of the old persistent class:

```
class MyPersClass : public BdbObject {
public:
    MyTransClass* transient( ) {
        MyTransClass* t = new MyTransClass( data );
        return t;
    }
private:
    d_ULong data;
};
```

The code of the base class for evolved classes. This is an abstract base class.

```
class MyPersClassBase : public BdbObject {
public:
    virtual MyTransClass* transient( )=0;
};
```

The code of the first evolution of the persistent classes:

```
class MyPersClass_001 : public MyPersClassBase {
public:
    MyTransClass* transient( ) {
        MyTransClass* t = new MyTransClass( data, data_001 );
        return t;
    }
    virtual d_ULong size { return 0; }
private:
    d_ULong _data, _data_001;
};
```

The code of the most recent evolution of the persistent classes:

```
class MyPersClass_002 : public MyPersClassBase {
public:
    MyTransClass* transient( ) {
        MyTransClass* t = new MyTransClass( data, data_001, data_002 );
        return t;
    }
    virtual d_ULong size { return 0; }
private:
    d_ULong _data, _data_001, _fata_002;
};
```

The next piece of code illustrates a proxy which will combine all things described above together.

```
MyProxy::MyProxy(...)
{
  // Load virtual tables for persistent classes.
    MyPersClass_001 t001;
    MyPersClass_002 t002;
    d_ULong size;
    size = t001.size( );
    size = t002.size( );
}
...
MyTransClass*
MyProxy::faultHandler(...)
{
    MyTransClass* objectT = (MyTransClass*) 0;
    BdbDatabase theDb( "xxx" );
    BdbHandle(BdbObject) objectH;
    theDb.fetch( objectH, "MyPersClass",, );
    if( objectH->ooIsKindOf( ooTypeN(MyPersClassBase))) {
        BdbHandle(MyPersClassBase) myObjectH;
        myObjectH = (BdbHandle(MyPersClassBase)&) objectH;
        objectT = objectH->transient( );
    } else if( objectH.typeN() == ooTypeN(MyPersClass)) {
        BdbHandle(MyPersClass) myObjectH;
        myObjectH = (BdbHandle(MyPersClass)&) objectH;
        objectT = objectH->transient( );
    } else {
        cout << "Unknown class: " << objectH.typeName() << endl;
    }
    return objectT;
}
```

A working example of the code based on this approach is spread between the following packages of the BaBar Software Repository [BSOFT]:

- **EmcEnv** V00-04-01
- **EmcGeom** V00-03-00
- **EmcGeomP** V00-01-00

V. References

[OBJYC++] "Using Objectivity/C++". 1996 Objectivity Inc., pages: 20-2, 20-4.

[BSOFT] www.slac.stanford.edu/BFROOT/www/Computing/Offline/SoftwareAdmin/ PackagesInfo.html