# Schema Migration for BaBar's Objectivity Federations

*S. Patton*[1,2]*, I. Gaponenko*[1]

[1] Lawrence Berkeley National Laboratory, USA.
[2] On behalf of the BaBar database group.

**Abstract**

As any High Energy Physics experiment evolves, the understanding and requirements for the data objects that need to be stored will change. This paper discusses the criteria to which these changes must conform so that physicists can continue to run their analysis programs and how these criteria are satisfied in BaBar's Objectivity Federations.

The Objectivity product has a native system for managing changes. Having reviewed this system BaBar decided that this system was not appropriate for its needs. The reasons behind this decision will be discussed. Having made this decision BaBar has had to develop their own techniques for handling changes in the requirements of persistent objects. These techniques are presented along with a discussion of their benefits and drawbacks.

Keywords: chep, objectivity, schema, schema migration

## 1 Introduction

BaBar software has been developed using Object Oriented technology and it was decided that an Object Oriented Database Management System (OODBMS) was a good fit to its data storage needs. The experiment choose Objectivity as the implementation of this system. One of the core components of this OODBMS is the specification of the characteristics of persistent classes in a "schema". During the lifetime of an experiment experience grows on what is necessary information and what is not. This naturally leads to physicists wishing to change the characteristics of those persistent classes, and thus the schema will need to be modified. These modifications to the schema are called a schema migration, or evolution.

This paper will discuss the criteria BaBar imposed on such migration bearing in mind the requirements of the physicist and data storage itself. Given these criteria the schema migration tools provided by Objectivity are reviewed. It turned out that these tools did not satisfy BaBar's criteria so it was decided that BaBar should implement its own solutions to the schema migration problem which would satisfy their criteria and function within the established framework of BaBar software. These solutions are outlined in this paper.

## 2 Schema and the Criteria for its Migration

An object oriented database (OODB) needs a description of the classes which it may contain. In the case of an Objectivity OODB this information is held in the "catalog" file in the form of type information which, among other things, includes the shape information of the class and whether it has a virtual table. The shape information basically records the type of and order of the member data for every class which may appear OODB. This includes persistent classes, i.e. ones explicitly derived for the Objectivity base class `ooObj`, and non-persistent classes , i.e. standard C++ classes which are allowed to appear as data members, either explicitly or implicitly, of persistent classes.

As understanding of the data improves, many of the classes in the schema turn out to be either cumbersome at best or insufficient at worst for their original task. This means that they need to be modified. Some modifications, e.g. changes in the implementation of the method of a class, do not effect information contained in the schema and therefore do not require explicit changes to the schema. However, there are many modifications (the Objectivity manual[1] lists 50 different possibilities) which do require a modification to the schema.

Modification to the schema, due to a change in an existing class, will effect programs which have already been built. For example, if a program uses an instance of a class which has been removed from the schema it clearly can not execute correctly. Similarly, if it is statically linked then its member functions are fixed and these expect a particular layout for a class (at least in C++). Therefore changing the structure of a class will cause such a program to become invalid. BaBar decided that this type of behavior was not acceptable as it could require the physicist to frequently rebuild their programs and production programs would also need constant rebuilding to match an constantly evolving schema. BaBar therefore stipulated that any modification to the schema must allow any program which already works with some subset of the data in the OODB to continue to work over that subset of data without any need to rebuild that program. This is a very severe restriction on the allowable changes, however, it does create a practical development environment for the physicists.

Another important consideration is that once much of the data has been placed in the OODB it will be migrated to tertiary media and effectively become read-only. This means that any schema migration which requires data to be changed to match the resulting changed schema can not be allowed.

## 3 Objectivity Schema Evolution

Objectivity provides a considerable number of tools to handle schema "evolution", as they call it (see chapter 5 of [1]). The general technique is to modify their class description files (`.ddl` files) and process them one or more times with their schema management tool, `ooddlx`. Once this has taken place then those classes whose data need modification must be "converted", i.e. made to conform to their new definition. While there are some migrations which change the schema but do not require objects to be converted, e.g. changing a data members name, the problem is that many migrations require data to be "converted", and these do not accommodate the "read-only" nature of the tertiary media storage of the data. The only sensible approach to accommodate these changes would be to update, and thus completely rewrite, all the data held on the tertiary media using a single "update application". However this would not be practical for more than one or two evolutions, if that! Moreover, as noted in the Objectivity manual[1]:

> Both conversion and non-conversion operations require you to *rebuild existing applications*.

This clearly fails the BaBar stipulation. It was therefore clear that BaBar could not use the Objectivity tools to manage its schema migration.

## 4 BaBar Schema Migration

The BaBar stipulation that required working programs not to become invalid meant that the schema could not be changed by any method that would require a conversion or non-conversion operation. Fortunately, adding a new class to a schema falls into this category. Therefore BaBar's approach to changing a persistent class is to create a new class whenever a modification to the definition of a class is required. By convention class names have a numerical suffix which specifies the

generation of a class within a "family" of classes, where each family represents the modification history of a single conceptual class.

The choice to create a new class for each migration has the potential to make code very difficult to write or understand as there could be many parts that would have to know which generation of a class family was being used and act accordingly. Therefore is was necessary, if the physicist's code was to be manageable, to design the BaBar framework in such a way that any client code can be insulated from having to know which generation of a class was used at any particular time.

The core design which enabled support for this requirement is that the client code is written in terms of transient classes, i.e. C++ classes which **never** appear in the schema of the OODB. This allows the persistent class to be changed without the physicists needing to know. (It also allows the underlying OODBMS to be changed but that is the topic of a different talk.) The management of creating these transient classes is handled in two different ways at BaBar:

- By using transient code, e.g. condition "proxy" code.
- By the persistent object itself, e.g. event store data.

The difference between these approaches is twofold. Firstly, it is a question of where to draw the line between transient and persistent code, i.e. whether the code should be associated with a persistent class or not. Secondly, in BaBar's implementation, the "transient code" approach requires one piece of code to know about all generations in a family of classes, whereas the "persistent object" version requires only knowledge of the previous generation.

Both approaches had been used successfully and the choice of which one to use in which situation was largely a matter of the designer's personal taste.

The creation of a persistent object from a transient object which is to be stored in the OODB is much easier than the opposite conversion. BaBar simply requires that it be stored as the most recent generation of the persistent class. Therefore this process is straight forward and only needs to involve one generation of a family and so will not be dealt any further in this paper.

### 4.1 Transient Conversion by a Transient Code[2]

The BaBar framework accesses the non-event store data via proxies. These proxies are activated by a request from a client for a particular piece of non-event data. If the valid value of the data is not already available then a fault handler is called which reads in the currently valid value of the data. It is this fault handler which fetches the persistent object from the OODB and converts it into a transient object, which is then returned to the client code.

The "standard" fault handler simply makes a call to the relevant API to fetch a generic handle to the persistent object. It then executes what is effectively a `case` statement, testing the actual object type against that for each generation in a family of classes. When it matches the correct type it casts the generic handle into the specific handle for that type and then builds the transient class based on the specific type. The convention in BaBar is that the persistent class has a member function, `transient`, which contains the code to build the matching transient class.

### 4.2 Transient Conversion by a Persistent Object[3]

Access to event store data is organized differently from the non-event store data as the access patterns are expected to be different. The expectation is that event data will be accessed very frequently while a program is executing and will change for every event. On the other hand non-event data is likely to be accessed less frequently and be valid for a range of events. This led to the design where non-event data can be efficiently managed by proxies (see Section 4.1) but this was not thought to be beneficial for event data. Therefore when a physicist configures their

program they need to explicitly specify which parts of the event data they will be using. This is done by activating the necessary pieces of code (called "scribes") which manage the conversion of transient and persistent objects.

The scribes are designed slightly differently from the proxies in that they only explicitly know about the most recent generation of the persistent class. Also, the persistent class has a more defined structure, mainly so the scribes can be implemented in terms of template classes, thus limiting the number of scribes which need to be written. Each generation of a class in a family must inherit from a common interface which is used for most operations by the Scribe. (A template class, whose parameter is the transient class, is used once again to define the interface.)

Unlike the proxy system, the API to the event store can not retrieve objects using a generic handle. Instead, the explicit type of the persistent class needs to be known. This means that the Scribes can not fetch the persistent class using a generic handle. The way the Scribes retrieve the correct handle is that each generation of a family has a static member function, `pullInterface`, which attempts to fetch an instance of the class, which matches the request, from the OODBMS. If a match is found it is returned as its base class (the "interface"). If no match is found then, if there is an earlier generation of the class, the `pullInterface` function of that previous generation is called. That way the initial call from the scribe can cascade through all generations until there is a match. If no match is found then a "null" handle is returned.

Once the appropriate persistent object is returned to the scribe the interface is then used to create the transient version of the class by calling the `transient` member function of the persistent object.

## 5 Conclusions

Physicists at BaBar have been insulated from having to deal directly with schema migration by the requirement on the framework that persistent objects must be converted into transient ones, which are independent of any underlying OODBMS, and are then used by any client code. The conversion can be handled either in transient code, e.g. proxies, or by the persistent classes themselves. e.g. scribes.

The drawbacks to this approach are that there is some amount of "code bloat" as all possible persistent classes need to be present in the code. Also, for any class which needs to be made persistent there needs to be a minimum of two versions, one transient version and one version for each OODBMS. However, the benefits of this approach is that it provides a stable environment in which the physicists can work and, as implemented at BaBar, does not require constant rebuild of code as the data model evolves.

## References


1 "Using Objectivity/C++ Data Definition Language."

2 `http://www.slac.stanford.edu/BFROOT/www/Computing/Offline/Databases/Conditions/Docs/SchemaEvolution/Strategy.pdf`

3 `http://www.slac.stanford.edu/BFROOT/www/Computing/Offline/Databases/Event/Docs/ScribesUserGuide.html`