Mapping Auxiliary Persistent Data to C++ Objects in the CDF Reconstruction Framework

J.Kowalkowski[1], *D.Box*[1], *P.Calafiura*[2],
J.Cranshaw[3], *R.Harris*[1], *M.Lancaster*[4], *M.Shapiro*[2]

[1] Fermi National Accelerator Laboratory, Batavia, Illinois, USA
[2] Lawrence Berkeley Laboratory, Berkeley, CA, USA
[3] Texas Tech University, Lubbock, Texas, USA
[4] University College London, London, UK

Abstract

Within the CDF event processing software, there is a need for information that is not directly stored with the event, such as calibration and alignment constants. This information is currently stored in relational databases and flat files and is presented to the event processing algorithms as objects. The objects are retrieved using a compound key that can be stored with the event data. The objects and keys are designed to be the algorithms view of the information in the database. The persistent form will be typically quite different - spanning several relational database tables or retrieved from several files. A database interface management layer exists for the purpose of managing the mapping of persistent data to transient objects that can used by the event processing algorithms. This layer sits between the algorithm code and the code that reads the data directly from permanent storage. At the persistent storage end, it allows multiple back-end mapping objects to be plugged in and identified as data sources at run time by a simple character string. At the user end, it places a get/put interface on top of a transient class for retrieval or storage of objects of this class using a key. The layer allows the user, anywhere in algorithm code, to make requests for specific objects. The mapping object creates the transient object requested by the user from information in the database using the key. The layer caches objects by key to prevent multiple database accesses from different algorithms for the same objects. The layer also provides a default key mechanism that allows different objects that share the same key type to be managed as a group or set, with a single set identifier. This grouping facility allows entire calibration sets to managed in one place within the reconstruction program and can be used to insure that algorithms retrieve the correct calibration constants for the event being processed.

Keywords: CDF,RDBMS,Object Persistence,Databases

1 Introduction

The software infrastructure at CDF contains a package called DBManager. The purpose of this package is to completely isolate the algorithms and code that run in the framework [6] from database or persistent storage technology. The DBManager package provides an API [1] that allows users to store and retrieve objects using keys in a database independent way. It provides a second back-end API used for creating transient to persistent mapping objects. These mapping objects are coupled to a specific type of database. The package manages sets of transient to persistent mapping objects that are chosen at runtime. Database dependent code and persistent object views are completely decoupled from user algorithms and transient objects. The primary user of the DBManager package is the calibration constants management system [1] at CDF and relational databases such as Oracle are the target persistent storage technology.

[1] API - Application Program Interface

2 Overview

The DBManager has two APIs. The back-end transient to persistent mapping API, or *IOPackage* [2] is an abstract base class. To create a new persistent to transient mapping class, one derives the class from *IOPackage* [3] . The front-end API that the user sees is C++ template based [8]. Transient objects are not related to the DBManager package, they are used as the template instantiation parameters for the *Manager<OBJ,KEY>* template class. The *Manager<OBJ,KEY>* class has methods such as get and put to retrieve and store transient objects.

To allow objects to be stored and retrieved with the front-end API, one defines a transient class and a key class. The API returns transient objects associated with the values in the associated key objects. In order to perform this task, the API uses the back-end *Mapper* objects. A single instance of a manager class can retrieve many transients objects.

In the calibration database, an example transient class has a list of channels that contain pedestal calibration information. The key is basically a (run,version) pair. Three *Mapper* classes exist for this transient object: one for Oracle (OCI), one for MSQL, and one for a simple file system text database. The code in an algorithm that uses this object named CalPed looks as follows.

```
// Manager<CalPed,CalibKey> ≡ CalPed_mgr
CalPed_mgr pedmgr("production","CalPed");
// Handle<CalPed> ≡ CalPed_var
CalPed_var ped;
CalibKey key(1200, latest);
// fill handle to object using the given key
pedmgr.get(key,ped);
```

The front-end API is contained in the CalPed_mgr class. This is really a template instantiation of the DBManager template class *Manager<OBJ,KEY>* . The argument ”production” identifies the back-end *Mapper* class set that should be used to perform the I/O. This is really an entire set of classes, such as the OCI set, or MSQL set. A simple text configuration file associates this string with a real database instance, normally by including a user, password, node name, and class set name. The indirection introduced by the configuration file allows for the database source to be switched by runtime (user logins, server names, database types). It also allow the same object to be retrieved from more than one database in the same job without changing any code. Using templated get/put classes allow objects to be manipulated by there correct type instead of an abstract one. It also allows the transient objects to be decoupled [4] from the I/O system. All objects returned through the API are managed by smart pointer classes. The CalPed_var class is really a typedef to smart pointer class *Handle<CalPed>*.

3 Features

The DBManager package manages a set of global structures within a running job. The global structures are hidden behind the *Manager<OBJ,KEY>* classes. This package manages all the database resources, including the objects stored and retreived, the connections, and the *Mapper* class instances.

Smart Pointers Instances of the class *Handle<OBJ>* appear as pointers to algorithm code. There is only one instance of the class associated with a given key that is shared by all code in a running application, regardless as to how many instances of *Manager<OBJ,KEY>* are created. The smart pointers handle memory management through reference counting [5]. In addition, these

[2] Concepts that have real C++ class associated with them will be shown in italics

[3] Any transient to persistent mapping class derived from *IOPackage* will be referred to as the *Mapper*

classes do deferred I/O; the objects are not retrieved from the database until the first use in the algorithm.

Caching The transient class instances can be cached by key value to a configurable depth. This feature is useful when algorithms continually retrieve the same objects. In the CDF Calibration system, the cache depth is two, meaning that the DBManager package will hold up to two transient class instances per class type. The cache is simply a map from key to instance and uses a write through policy. It is most useful in applications where objects are not modified in place.

Factory A pluggable factory [2] [3] is used for creating the transient *Mapper* class instances at run time by name. There is a simple procedure that developers must follow when a new *Mapper* is introduced into the system, the procedure causes the *IOPackage* factory to be populated automatically at run time. The factory has no coupling to the *Mappers* that it creates. Information in a configuration file describes which concrete *Mapper* instances will be created and used at run time.

Notification System Each *Mapper* instance is registered by key type name in a subsystem that follows the observer pattern [3]. The example code earlier in this paper shows a get with key; one can also perform a get without a key. From a DBManager controller object, the user can inform the system that *Mappers* registered with a particular key type name should be notified. The purpose of this notification procedure is to inform all *Mappers* that use the same key type that they should set up a default object. The default object is what is returned by the get without key.

This feature exists to automatically support run number changes. All calibration object types use the same key type. There exists a single, unique number for each run that identifies a complete set of calibration objects across all detectors that can be used for a that run. When the run changes, all *Mappers* are notified. They go to the database and translate the unique number into a specific key and automatically load the associated object and put it into the cache as the default object. The algorithm can always get the correct calibration constants for the current run by using the get without key.

The smart pointers take part in this feature. If a user has a smart pointer to an object retrieved using the get without key call, the data in the object will be automatically replaced with fresh or current data each time the run changes - without the algorithm reissuing the get.

Code Generation A code generator sits on top of this package. A simple Java class describing the transient class structure is read in by the generator. The generator uses this information to create the transient classes and *Mapper* classes for each of the three supported databases. This facility makes it easy for users to add or change calibration objects. This facility makes several ad-hoc assumptions as to the structure of the calibration objects: they are containers of simple data-only classes (channel information) and they are all retrieved using a common key type (run/version).

4 Conclusion

This package has been in place and tested for quite a while. It is just recently that algorithms are requiring the use of calibration constants is a coherent fashion. It is yet to be determined if the package is sufficient to do all that is needed. The package is also being used by the CDF Data Handling Data File Catalog project. A package that maps relational database table rows to objects is hard to design [7]. This one assumes simple class structures, ones with mostly data or collections of data. Separating the loading of data into the transient objects into different classes

(*Mappers*) allows for a great deal of flexibility, but also forces exposure of the implementation details of the transient classes. For simple objects, this does not appear to be a problem, although it is not very object-oriented. The alternatives to this in C++, such as streamers in the objects themselves, are better in some ways and worse in others. A language with a meta-object facility such as Java is better suited for this type of problem. Strong points of this package are:

- Strong typing of transient objects at back and front ends through the use of templates
- Objects and access code at the algorithm or user level look nice
- The package is almost completely hidden by C++ typedefs
- Adding new code-generated objects that fit the model is very easy
- Choosing a source database is configurable at run time
- Coupling to a database system occurs only at link time or run time
- *IOPackage* interface is easy to use (*Mappers* are easy to create)
- Database connections are managed for the developer
- Objects and data are shared everywhere in the program
- Package will maintains data consistency provided rules are followed
- Allows database technology to be swapped out without code changes
- Can make for an easy transition from a two to three tier database architecture in the future.

Weak points of this package include:

- Problem keeping *Mapper* objects in sync for all databases
- Persistent objects are generally simple data objects
- Code generation is still primitive and needs expansion
- Statically bound executables can include unused code, making them large
- Fitting classes into the code generator object model can be difficult
- Difficult to do user friendly error checking without C++ exceptions
- Users can bypass the rules and therefore bypass consistency checks
- Provides a flat view of data, hierarchical useful many times

Future In future releases, the code generation will be enhanced and lighter weight handle classes will be provided. Dynamically loadable *Mapper* classes will also be made available, this feature would allow code for the requested database type to be pulled in on demand. It would be nice to use UML tools to generate the *Mapper* and transient classes directly off diagrams. Many application do not require deferred I/O or the advanced notification mechanism; a light weight handle class could be provided that does not include these facilities.

References

1 Cranshaw J. "The CDF Run II Calibration Database," `http://www-cdf.fnal.gov/upgrades/computing/calibration_db/orac_calibdb.html`, 1999.

2 Culp T. "Industrial Strength Pluggable Factories," *C++ Report*, 11(9): 21-26, October 1999.

3 Gamma E., et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

4 Lakos J. *Large Scale C++ Software Design*, Addison-Wesley, Reading, MA,1996.

5 Meyers S. *More Effective C++*, Addison-Wesley, Reading, MA,1996.

6 Sexton-Kennedy E. "A Users Guide to the AC++ Framework," `http://www-cdf.fnal.gov/upgrades/computing/projects/framework/frame_over/frame_over.html`, 1997.

7 Vadaprty K. "Bridging the Gap Between Objects and Tables: Object and Data Models," *JOOP*, 12(2): 60-64, May 1999.

8 Weihe K. "Using Templates to Improve C++ Designs," *C++ Report*, 10(2): 14-21, February 1998.