An Approach for Implementing Object Persistence in C++ using Broker

T. Kulathu Sarma. GoldAvenue, Geneva, Switzerland. Email: kulsarma@goldavenue.com

Abstract— Object and relational technology have conflicting architectural and design principles. Storing objects in a relational database takes considerable amount of time and effort in an overall project development schedule. One approach to implement object-relational mapping is to use a persistence framework. Smalltalk and Java have in built runtime features like Reflection and Introspection to assist in building such a framework. C++ runtime support does not go beyond type identification and its strong type checking imposes additional complexity in implementing a persistence framework. This paper demonstrates an approach for implementing a persistence framework (O-R Link) in C++.

I. INTRODUCTION

Object Oriented technology has become a de-facto standard for constructing mission critical applications. Object Oriented applications have proved to be more robust, scalable and adaptable when compared to traditional ones. Object persistence (or simply persistence) refers to storing an object in a persistence mechanism after the lifetime of the application that created it. When the application is run again, the data is read from the persistence mechanism to re-create the object. In practice, object persistence is found more readily in relational databases rather than object databases. This is either because of legacy reasons, i.e. for accessing relational data present in legacy systems or for the perceived strengths of relational technology in transaction processing, powerful querying/reporting capabilities or its support for integrity constraints.

Marrying object-relational technology is a common problem to handle in any serious application development project because of the conflicting architectural and design principles. One approach is to use a “Persistence Framework” (also called as a Persistence Layer or Object-Relational mapping layer) to map domain objects to database tables. A persistence framework is a set of closely related classes to “broker” object-based data requests to an underlying persistence mechanism. A well-designed persistence framework allows application developers to concentrate on modeling the business requirements without having to worry about any data model changes, ranging from a simple column change or a table re-organization, to a more complex one like changing the persistence mechanism from a database to a flat file or vice versa.

Smalltalk and Java have Reflection and Introspection features to assist in building a persistence framework. C++ runtime support does not go beyond type identification and its strong type checking imposes additional complexity in implementing one. Scott. W. Ambler [2] talks about a generic design of a persistence framework without going much into the language specifics. According to Ambler, implementing a persistence framework is very difficult and it is even more difficult in a language like C++. Kyle Brown’s [6] work on Cross Chasms goes over a set of related patterns involved in Object-RDBMS integration, mainly from a Smalltalk perspective. Frank Buschmann [1] uses a persistence component example in explaining reflection architectural pattern and metaobject protocol (MOP). Buschmann talks about reflection in the context of a wider problem in architecting a dynamically changing system, both in structure and behaviour.

This paper introduces the O-R Link (Object-Relational Link), a persistence framework implemented in Visual C++ and MFC (Microsoft Foundation Classes) and talks about metadata and reflection usage in the implementation. The core framework implementation uses C++ macros and MFC’s general-purpose classes, which is easily portable to other platforms and C++ compilers. Next section goes over O-R Link application architecture and framework classes. This is followed by a discussion on the key implementation concepts, metadata objects and reflection with explanations using class diagrams and source code segments. Appendix explains few other concepts along with other O-R Link classes of interest.

II. O-R LINK APPLICATION ARCHITECTURE AND FRAMEWORK CLASSES OVERVIEW

O-R Link applications are based on the ‘Layer pattern’ as described by Frank Buschmann [1]. A Layer pattern decomposes an application into a group of subtasks in which each group belongs to a particular level of abstraction. A typical example for the Layer pattern is the OSI 7-Layer Model.
In the Layer pattern, a layer at any given level uses the services provided by the layer below it and in turn provides services to the layer above it. The top most layer provides services to the client or the end user. Applications developed using O-R Link Framework follow a typical three-layer model as shown in Figure 1.

O-R Link Framework classes form the data access layer. Domain classes present in the business logic layer inherit their behaviour from CPersistentObject, the base class for all persistent domain classes. Object to relational mapping details are stored as metadata objects. The Broker class, CPersistenceBroker and its associates handle object mapping using metadata objects.

![Layered Architecture](image)

Figure 1 – Layered Architecture used by O-R Link applications

Table I presents an overview of O-R Link classes with their usage and application visibility. As shown in Table I, application developers only access a few classes directly, whereas the rest provide persistence support behind the scenes.

### III. O-R LINK FRAMEWORK IMPLEMENTATION CONCEPTS

Metadata management and reflection are the key concepts employed in O-R Link implementation. It uses a variant of a Reflection architectural pattern as discussed by Frank Buschmann [1].

#### A. Metadata management

“Object Type” uniquely identifies objects of a given domain class and “Attribute ID” identifies attributes of the class. For O-R Link to map objects to tables, it needs to know:

1. the relational database (persistence mechanism) connection details including the data source name, the database server name and user id/password;
2. the table(s) within the database with details including column name, type, size etc;
3. the mapping details between the object attributes and the table columns.

The challenge for O-R Link is to broker object data to table(s) without directly knowing about the object or the table. O-R Link uses metadata objects to achieve this as metadata objects capture information about the domain object types,
attribute to column mapping details and the persistent mechanism configuration details. Any change in the data representation, table structure or persistent mechanism will need relevant metadata changes without affecting the domain application.

Figure 2 shows CPersistenceBroker class with the associated metadata classes. CPersistenceBroker implementation is based on the Singleton Pattern discussed by T. Kulathu Sarma [4] and by Erich Gamma [3].

Read and Delete requests follow a similar route. O-R Link uses an ODBC data source to store metadata objects (an Access database by default). The administration module maintains metadata and uses wizards to generate DDL (Data Definition Language) scripts and source code for domain classes. These tools expedite application development process.

### B. Reflection

Metadata assists O-R Link in object mapping. However, the framework should know the class members to get/set the values. Typically, each private data member has a pair of get and set methods. For example, in a customer domain class, GetFirstname method returns the first name of the customer and O-R Link calls this method when preparing an Insert/Update SQL statement. Conversely, when reading the object from the persistent mechanism, SetFirstname method is called.

In built reflection feature in Smalltalk and Java, help to discover and invoke get or set methods at runtime. In the purest sense, Reflection is a feature that allows an object to look at itself, as in a mirror. Reflection identifies data members and methods that are associated with an object, which makes it possible for others to query and invoke an object’s method at runtime. Sometimes this feature is called Introspection but the intent is the same regardless of the name.

C++, being a strongly typed compiler language, has no direct support for Reflection but O-R Link mimics this feature using attribute maps, in a technique similar to MFC’s message maps [7]. Attribute map uses “Attribute ID” to associate an attribute to get and set methods. Classes derived from CPersistentObject automatically have the attribute map support. Attribute map uses a pair of structures and three macros. This section covers the implementation of:

- CPersistentObject forwards the Save request to the broker;
- Broker checks the map class cache (m_MapCache) for a corresponding mapper object (CDBMap object). If there is a match, the request is forwarded to the map class object or else the broker creates the mapper object using the runtime information and configures it with the CPersObjDefn, before forwarding the request. In the latter case, the mapper object is added to the cache;
- Mapper object uses CPersObjDefn and CAttributeDefn to form Insert/Update SQL statement and returns it to broker for execution;
- Broker uses CPersistenceMechanism object to process the SQL.

![Figure 2 – CPersistenceBroker and metadata classes](image-url)
An Approach for Implementing Object Persistence in C++ using Broker

• the additional macros;
• calling getters and setters from the framework code.

The first structure is PERS_ATTRIBMAP_ENTRY, which is used to represent the individual entry in the attribute map table. This structure is shown below:

```c
struct PERS_ATTRIBMAP_ENTRY {
    LPCSTR lpszAttributeID;
    UINT nSig;
    PERS_ATTRIB pfn;
};
```

The first data member (lpszAttributeID) represents the Attribute ID. The signature member (nSig) indicates the signature of the member function (parameters and return value) and the third member (pfn) is a function pointer to Get/Set method.

The second structure is PERS_ATTRIBMAP. The first member is a pointer to the base class’s attribute map (pBaseMap). The second member (lpEntries) is a list of attribute map entries. If there are no matching get/set methods for an attribute in a given class, the framework checks the base class attribute map (pBaseMap). This check continues all the way up, until a matching entry is found or CPersistentObject is reached.

```c
struct PERS_ATTRIBMAP {
    const PERS_ATTRIBMAP * pBaseMap;
    const PERS_ATTRIBMAP_ENTRY * lpEntries;
};
```

DECLARE_ATTRIB_MAP, BEGIN_ATTRIB_MAP and END_ATTRIB_MAP are the three macros used in the implementation of attribute maps. The first macro DECLARE_ATTRIB_MAP is included in the header file (.h file) of CPersistentObject’s descendent whereas the second and the third macro is part of the implementation file (.cpp).

DECLARE_ATTRIB_MAP is shown below:

```c
#define DECLARE_ATTRIB_MAP() \
private:
    static const PERS_ATTRIBMAP_ENTRY _attribEntries[];
    static const PERS_ATTRIBMAP attribMap;
public:
    virtual const PERS_ATTRIBMAP* GetAttribMap() const;
```

This macro declares a private static array of attribute map entry structures (_attribEntries) and a private static member of attribute map (attribMap). Also, it declares a public method (GetAttribMap()).

The second macro BEGIN_ATTRIB_MAP expands to:

```c
#define BEGIN_ATTRIB_MAP(theClass, baseClass) \
const PERS_ATTRIBMAP* theClass::GetAttribMap() const \
{ return &theClass::attribMap; }\
const PERS_ATTRIBMAP theClass::attribMap = \
{ &baseclass::attribMap, &theClass::_attribEntries[0] ;}\
const PERS_ATTRIBMAP_ENTRY theClass::_attribEntries[] = \
{ \
}
```

and the third macro END_ATTRIB_MAP expands to:

```c
#define END_ATTRIB_MAP() \
{0, PERSSig_end, (PERS_ATTRIB)0 } \
};
```

BEGIN_ATTRIB_MAP macro takes the current class and the base class as parameters. It implements GetAttribMap() method to return a pointer to the attribute map static member (attribMap). The attribute map structure (attribMap) is then initialized with the base class map and the pointer (lpEntries) is set to the first entry in the attribute array (_attribEntries[0]). Finally, the macro adds the necessary code for initializing attribute entries. Given below is the sample code (CCustomer class) using the macros:

```c
// Class declaration for CCustomer in Customer.h
class CCustomer : public CPersistentObject
{ public:
    CCustomer();
    CCustomer( CCustomer & );
    virtual ~CCustomer();

    DECLARE_ATTRIB_MAP()
};
```

The preprocessor creates the following code:

```c
// Class declaration for CCustomer in Customer.h
// (expanded)
class CCustomer : public CPersistentObject
{ public:
    CCustomer();
    CCustomer( CCustomer & );
    virtual ~CCustomer();

    DECLARE_ATTRIB_MAP()
};
```

The preprocessor creates the following code:
An Approach for Implementing Object Persistence in C++ using Broker

public:
    virtual const PERS_ATTRIBMAP* GetAttribMap() const;
}

// Implementation in Customer.cpp
BEGIN_ATTRIB_MAP(CCustomer, CPersistentObject)
    //\{PERS_ATTRIB_MAP(CCustomer)
    ON_SETATTRIB_LONG(CUSTOMERID_ATTRIB, 
        OnSetCustomerID)
    ...
    ON_GETATTRIB_CSTRING(FIRSTNAME_ATTRIB, 
        OnGetFirstName)
END_ATTRIB_MAP()

At the preprocessing stage this expands to:

// Implementation in Customer.cpp (expanded)
const PERS_ATTRIBMAP* CCustomer::GetAttribMap() const
{ return &CCustomer::attribMap; }
const PERS_ATTRIBMAP CCustomer::attribMap = 
{ & CPersistentObject::attribMap, & CCustomer::_attribEntries[0] }; 
const PERS_ATTRIBMAP_ENTRY CCustomer::_attribEntries[] =
{\{PERS_ATTRIB_MAP(CCustomer)
    ON_SETATTRIB_LONG(CUSTOMERID_ATTRIB, 
        OnSetCustomerID)
    ...
    ON_GETATTRIB_CSTRING(FIRSTNAME_ATTRIB, 
        OnGetFirstName)
    {0, PERSSig_end, (PERS_ATTRIB)0} 
};

Between BEGIN_ATTRIB_MAP and END_ATTRIB_MAP, there are additional macros that expand to fill in attribute map entries. In the above example, ON_SETATTRIB_LONG macro expands to associate a set method (OnSetCustomerID) for customer id attribute (CUSTOMERID_ATTRIB). Similarly, ON_GETATTRIB_CSTRING macro expands to associate a get method (OnGetFirstName) for customer first name attribute (FIRSTNAME_ATTRIB). Given below is the definition for these macros:

#define ON_SETATTRIB_LONG(lpszAttribute, memberFxn) 
{ (LPCSTR)lpszAttribute, PERSSig_SetAttrib_LONG, 
    (PERS_ATTRIB)(void(PERS_ATTRIB_CALL CPersistentObject::*)(const 
        LONG))&memberFxn },
#define ON_GETATTRIB_CSTRING(lpszAttribute, memberFxn) 
{ (LPCSTR)lpszAttribute, PERSSig_GetAttrib_CSTRING, 
    (PERS_ATTRIB)(const CString(PERS_ATTRIB_CALL 
        CPersistentObject::*)())&memberFxn },

These macros expand to fill in the Attribute ID (lpszAttributeID), signature (nSig) and the member function (pfn) of the attribute map entry structure. O-R Link Framework uses the signature to determine the member function (pfn) parameters and the return type. In the above example, macros expand to set and get attributes of type long and CString respectively. Similar macros are available for other data types including char, int, float, double, CTime and CPersistentObject. Signatures are declared as enumerated data type (PERSSig), a portion of which is shown below:

// List of predefined signatures
enum PERSSig
{ 
    PERSSig_end = 0, // marks end of attribute map 
    PERSSig_SetAttrib_LONG, // BOOL (const LONG) 
    PERSSig_SetAttrib_CSTRING, // BOOL (const CString &)
    ...
    PERSSig_GetAttrib_LONG, // const LONG 
    PERSSig_GetAttrib_CSTRING, // const CString &
    ...
};

When an object is saved, getter methods are called to form an Insert/Update statement. Given below is the GetAttributeValue method called from FormInsertStatement or FormUpdateStatement methods (not shown). GetAttributeValue method takes Attribute ID (lpszAttribute) and object being saved (pPersistentObject) as parameter and returns attribute value as a CString object.

CString CDBMap::GetAttributeValue( LPCSTR lpszAttribute, 
    CPersistentObject * pPersistentObject, 
    INT & nStatus ) 
{ 
    union AttribMapFunctions amf;
    const PERS_ATTRIBMAP * pAttribMap =NULL; 
    const PERS_ATTRIBMAP_ENTRY * lpEntry = NULL;
    CHAR szValue[ 255 ];
    nStatus = 0;
    for(pAttribMap = pPersistentObject ->GetAttribMap(); 
        pAttribMap != NULL; 
        pAttribMap = pAttribMap ->pBaseMap )
    { 
        lpEntry = pAttribMap->lpEntries;
        while (lpEntry ->nSig != PERSSig_end)
        { 
            if( stricmp( lpszAttribute, 
                lpEntry->lpszAttribute ) == 0 )
            { 
                switch(lpEntry->nSig)
                { 
                    case PERSSig_GetAttrib_LONG:
                        amf.pfn = lpEntry->pfn;
                        sprintf( szValue, "%ld", 
                            (pPersistentObject - >*amf.pfn_GETATTRIB_LONG)( );
                        return CString( szValue );
                    case PERSSig_GetAttrib_CSTRING:
                        amf.pfn = lpEntry->pfn;
                        return CString( "\"" ) + 
                            (pPersistentObject - >*amf.pfn_GETATTRIBCSTRING()) + 
                            CString( "\"" );
                }
            }
            lpEntry++;
            amf.pfn = lpEntry->pfn;
        }
    }
GetAttributeValue starts with AttribMapFunctions (amf) union declaration. A part of the union declaration is shown below. It has a pointer to a generic member function (pfn) and a list of possible prototypes for getters/setters can be called. The union assigns proper getter/setter prototype based on the signature.

union AttribMapFunctions
{
    PERS_ATTRIB pfn; // generic member function pointer
    void (PERS_ATTRIB_CALL
        CPersistentObject::*pfn_SETATTRIBCHAR)( const CHAR );
    void (PERS_ATTRIB_CALL
        CPersistentObject::*pfn_SETATTRIBLONG)( const LONG );
    void (PERS_ATTRIB_CALL
        CPersistentObject::*pfn_SETATTRIBCSTRING)( const CString & );
    ... const CHAR (PERS_ATTRIB_CALL
        CPersistentObject::*pfn_GETATTRIBCHAR)();
    const LONG (PERS_ATTRIB_CALL
        CPersistentObject::*pfn_GETATTRIBLONG)();
    const CString (PERS_ATTRIB_CALL
        CPersistentObject::*pfn_GETATTRIBCSTRING)();
    ...);
}

Following the union declaration, GetAttributeValue initializes pAttribMap with the current object’s attribute map and sets lpEntry to the first element. The for loop iterates through the attribute map starting from the current class map pointer all the way up to the root class map (CPersistentObject). The while loop iterates through the current map until the signature end (PERSSig_end) is reached or a matching attribute is found.

Upon finding a matching attribute, the appropriate signature is looked for calling the member function. The member function pointer (amf.pfn) in the union (AttribMapFunctions) is set and called. If there is no match in a given map, the base class map is searched, which continues until a match is found or when the whole class tree is searched without any success. In the latter case, an error flag is set (nStatus = 1).

When an object is read from the database, set methods are invoked similarly. Note, GetAttributeValue is part of CDBMap, the class responsible for mapping objects to database tables.

IV. CONCLUSION

This paper focuses on primary concepts employed in O-R Link implementation. In order to respect the paper size, secondary concepts were not discussed. The actual O-R Link implementation has security features including Authorization, Access Control List (ACL), Role Based Access and Audit Logging. Other features include transaction support, object caching, reporting, exporting/importing objects to/from CSV, XML and other file formats. O-R Link Framework can represent complex objects as composites and supports automatic read, write, delete and object duplication. The core framework implementation uses C++ macros and MFC’s general-purpose classes, which is easily portable to other platforms and C++ compilers. Administration and metadata management tools provide O-R Link applications, the luxury of working without having to worry about persistent mechanism changes.

APPENDIX

A. Object Creation

When an object is read, O-R Link uses metadata to identify the “Object Type”. An instance of the domain class corresponding to the “Object Type” should be created before invoking set methods. MFC’s RTCI (runtime class information) feature is used to create an object at runtime without having to know the domain class directly (i.e. to include the header file in the framework code). Given below is the code segment from CPersistenceBroker class to create domain objects. Runtime class information and dynamic object creation are implemented using CRuntimeClass (a C structure) and a pair of macros DECLARE_DYNCREATE/IMPLEMENT_DYNCREATE [7].

// Persistent Broker code
CPersistentObject * CPersistenceBroker::CreateObject
( LPCSTR lpzObjectType)
{
    CPersObjDefn   * pPersObjDefn = NULL;
    CPersistentObject * pObject = NULL;
    // NOTE: m_ObjDictionary is an CMapStringToPtr object, // a
dictionary collection class that maps unique CString
// objects to void pointers. Query persistent object definition for the
given // object type.
m_ObjDictionary.Lookup(lpzObjectType, ( LPVOID & )
pPersObjDefn );
// Create an object if a valid definition is returned
if(pPersObjDefn != NULL )
    {
        pObject = pPersObjDefn->m_pRuntimeClass->CreateObject();
    }
// Return pObject which can contain a valid object or
// NULL indicating error
return pObject;
}

B. Class Registration

O-R Link Framework uses “Class Registration” to avoid direct references to the domain classes. During initialization, the framework calls an application “hook” function with the broker object as a function parameter. The application implements the
hook function to register the domain classes and their associated runtime class information with the broker. As discussed, this information is used to create domain objects dynamically. A sample hook function and a registration function are shown below:

```
//Application code
#include "customer.h"

INT GfnRegisterClasses( CPersistenceBroker * pPersistenceBroker )
{
    // Step 1 - Register classes with persistence broker
    pPersistenceBroker->Register( CUSTOMEROID_OBJ_TYPE,
        RUNTIME_CLASS( CCustomer ));
    pPersistenceBroker->Register( PRODUCTTYPE_OBJ_TYPE,
        RUNTIME_CLASS( CProductType ));
    return 0;
}
```

```
// Persistent Broker code
INT CPersistenceBroker::Register( LPCSTR lpszObjectType,
    CRuntimeClass * pRuntimeClass )
{
    // Validate pRuntimeClass and use Object Dictionary
    // to Register the object information
    ASSERT( pRuntimeClass != NULL );
    // Create CPersObjDefn for the object type
    CPersObjDefn * pPersObjDefn = new CPersObjDefn(lpszObjectType, pRuntimeClass);
    // Read attribute definitions
    pPersObjDefn ->Initilize();
    m_ObjDictionary.Add(pPersObjDefn, lpszObjectType);
    ... return 0;
}
```

C. CPersistentObject and associated classes

CPersistentObject is the base class for all persistent domain classes. It has three attributes an OID (m_pOID), an ObjectType (m_csObjectType) and a proxy indicator (m_bProxy). Proxy objects contain the minimal information needed to identify the object. When needed, Load method reads the complete object. Proxy implementation follows the Proxy design pattern as discussed by Erich Gamma [3]. Save and Delete methods forward the respective application request to the broker.

COID class (The Object Identity Class) has a data member to uniquely identify an object (m_pOID). OIDs can be generated either by using relational database sequence generator or within the application by incrementing and storing the next OID in a separate table. In the latter case, GenerateOID method in CPersistentObject is called, which uses a factory class COIDFactory for OID generation. This implementation is based on the Factory Method Pattern as described by Erich Gamma [3].
E. CPersistentMap class hierarchy

CPersistentMap is an abstract class for mapping domain objects to the persistent mechanism (see Figure 5). The broker configures the CPersistentMap object with a CPersObjDefn (m_pPersObjDefn). Interfaces provided by this class are implemented by its concrete descendents.

CDBMap is a concrete implementation of CPersistentMap. The broker calls the corresponding method to form Insert, Update or Delete statements. CDBMap uses CPersObjDefn and CAttributeDefn objects to construct an SQL statement and pass it back to the broker for execution. Query method retrieves a set of objects based on the given criteria (CRetrieveCriteria) using a dynamic CRecordset class [5] and the result is stored in a CCursor object. Read method is implemented to read all the attributes of a Proxy object.

CXMLMap provides a concrete implementation to persist objects in a XML database using XML schema.

F. Mapping Schemes

There are several schemes to map an object class to a table,

- One class can be mapped to one table
- One class can be mapped to multiple tables
- Multiple object classes can be mapped to one table
- Multiple object classes can be mapped to multiple tables

This paper will not go over these mapping schemes. Crossing Chasms [6] by Kyle Brown and Bruce G. Whitenack covers object-table mapping and related patterns from a Smalltalk perspective. O-R Link implements most of the patterns discussed by Kyle Brown in C++.

ACKNOWLEDGMENT

Thanks to Professor B. Srinivasan and Simon Davis for their valuable suggestions in improving the quality of the paper. Special thanks to my wife R. Subhashini for all her assistance in writing this paper.

REFERENCES