A Framework for Object-Relational Mapping With An Example in C++

Xiaobing Zhang

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada
April, 2004

February, 2004

Your file Votre référence
ISBN: 0-612-94760-2
Our file Notre référence
ISBN: 0-612-94760-2

ABSTRACT

A Framework for Object-Relational Mapping With An Example in C++

Xiaobing Zhang

As the Object-Oriented programming technique becomes more and more popular in contemporary software design, issues related to persistent objects must be addressed. This thesis introduces a framework for Object-Relational Mapping. The framework is intended to simplify the handling of persistent objects in a Relational Database System.

The framework's architecture consists of two layers: an object layer that contains the infrastructure for persistent objects and a storage layer that provides an interface to the Physical Storage System. As contributions, I have introduced my original work including mapping inheritance with inheritance, a particular solution for aggregation and associations mapping, a cache of object references for constructing objects and name conventions for preserving object maps.

The Framework for Object-Relational Mapping is a C++ Framework (a set of Classes). It supports most of relational database systems. Developers can use these classes to obtain abilities about object relational mapping. In this thesis, a teaching assignment planner project is used to test my framework's performance in saving coding work.

Acknowledgements

There is a person without whom this thesis would not have been at all possible and whom I need to thank:

Professor Peter Grogono, my supervisor, has endlessly and tirelessly mentored, taught and encouraged me since the summer, 2001. I appreciate his many useful comments on this work, but even more so, I appreciate his advice, comments, and willingness to discuss any questions or ideas that I have had.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Object Persistence

Object-Oriented modeling, design, and programming is becoming more and more popular in software design nowadays. Since most computer applications need to store and retrieve data, saving and recreating objects is an unavoidable problem in object-oriented applications. A definition for object persistence was given in [1]:

> “The process of storing and retrieving objects (or more accurately their attributes) is called persistence”.

This thesis is about object relational mapping design and implementation. It describes a framework to deal with object relational mapping that allows developers who are using object oriented programming to store their objects into a relational database, thereby achieving persistence.

To start, I will review the background knowledge of some possible ways to deal with object persistence

1.2 Background

When developers are facing the problem of object persistence, there are several solutions they can choose from: serialization, object-oriented database, and relational database (mapping). Developers choose these solutions based on system requirements.

1.2.1 Serialization

Object Serialization is the ability to save the states of an object to an output stream. Users can recreate the object by reading the saved states from an input stream. The stream may be a disk file, a byte array or a stream associated with a TCP/IP socket.

Serialization has two major disadvantages. First, The object can only be accessed as a whole. Serialization does not allow access to or update of a single object independently because the stream medium does not support random read and write operations. Secondly, since there is no transaction control in Serialization, it does not support concurrent access. When two or more users are serializing objects, they could overwrite another user's objects by saving objects into the same stream. This makes Serialization unsuitable for multiple user systems.

Although Serialization has many disadvantages, it is provided by most OOP languages in the form of a method or interface. Developers can use it without any further work.

1.2.2 Object oriented databases

Srinivasan and Chang in their paper [1] introduced object oriented database as follow:

> Object-oriented DBMSs (OODBMSs) are basically built on the principle that the best way to add persistence to objects is to make objects persistent that are used in an object-oriented programming language (OOPL) like C++ or Smalltalk. Because OODBMSs have

their roots in object-oriented programming languages, they are frequently referred to as persistent programming language systems. Object-oriented DBMSs, however, go much beyond simply adding persistence to any one object-oriented programming language. This is because, historically, many object-oriented DBMSs were built to serve the market for computer-aided design/computer-aided manufacturing (CAD/CAM) applications in which features like fast navigational access, versions, and long transactions are extremely important. Object-oriented DBMSs, therefore, support advanced object-oriented database applications with features like support for persistent objects from more than one programming language, distribution of data, advanced transaction models, versions, schema evolution, and dynamic generation of new types. Even though many of these features have little to do with object orientation, object-oriented DBMSs emphasize them in their systems and applications. There are several object-oriented DBMSs in the market (e.g., Gemstone**, Objectivity/DB**, ObjectStore**, Ontos**, O2**, Itasca**, Matisse**).

There is no doubt that OODBMS is ideal to deal with object persistence save objects because:

> "OODBMS (object-oriented database management system) products are designed to work well with object programming languages such as C#, C++, and Java. When you integrate database capabilities with object programming language capabilities, the result is an OODBMS. An OODBMS makes database objects appear as programming language objects in one or more object programming languages. An OODBMS extends the language with transparently persistent data, concurrency control, data recovery, associative queries, and other capabilities." [3]
>
> "OODBMSs provide the lowest cost for development and best performance combination when using objects because they store objects on disk and have the transparent program integration with object-oriented programming languages. This is because an OODBMS stores exactly the same object model that is used at the application level, both development and maintenance costs can be reduced." [3]

However, the OODBMS does have one major disadvantage: limited market acceptance. Comparing with the relational counterpart, it has fewer users because the existence of large RDBs would slow down the acceptance of OODBs. This is a major reason why OOP developers need relational database.

1.2.3 Relational Database

Srinivasan and D. T. Chang in their paper [1] also introduced relational database as follow:

> Relational DBMSs typically provide support for storing data used in traditional business applications such as banking transactions and inventory control. The relational model is the basis of many commercial relational DBMS products (e.g., DB2*, Informix**, Oracle**, Sybase**) and the structured query language (SQL) is now a widely accepted standard for both retrieving and updating data. The basic relational model is simple and mainly views data as tables of rows and columns. The types of data that can be stored in a table are basic types such as integer, string, and decimal, and other special types such as BLOB (binary large object) and CLOB (character large object). These systems typically do not allow users to extend the type system by adding new data types. They also only support first-normal-form relations in which the type of every column must be atomic, i.e., no sets, lists, or tables are allowed inside a column. Relational DBMSs have been extremely successful in the marketplace, growing into an approximately four-billion-dollar market in a decade. These systems are extremely good for a class of applications with simple data models and extensive querying needs. The use of a standard declarative query language in SQL makes it possible for applications to transparently access relational DBMS data from different vendors [1].

Another way of obtaining object persistence is to use a relational database as the storage medium. A relational database is slower than an object-oriented database. However, systems often need to store objects in a relational database because relational databases are more popular than OODBMS now.

Since relational databases do not support object model and need transformations for Object modeling, designing software to connect an object-oriented business system with a relational database is a difficult task. Transformations between object and relational model are called Object-Relational Mapping (ORM) [1][7].

1.3 Object-Relational Mapping

Object-Relational Mapping is the process of transforming between object and relational modeling approaches and between the systems that support these approaches.

To introduce Object-Relational Mapping, I will discuss the differences between object and relational modeling.

1.3.1 Object Modeling

"Object modeling describes systems built out of objects, where objects are programming abstractions that have identity, behavior, and state" [8]. Objects are an abstraction beyond simple data type (single value data type), which are provided by most relational database systems. As such, the object modeling includes many other concepts, such as aggregation, inheritance, polymorphism, association and data types, all of which make object oriented types "smarter" than relational database data types [11].

Object modeling uses Interclass references and class hierarchy inheritance to represent the relationship between objects.

1.3.2 Relational Modeling

Relational modeling and object modeling are different paradigms of programming. Relational modeling is based on the information. It only supports simple data type (single value data type) and uses value-based (Key) Approach to build the relationship between tables. For mapping other concepts of object modeling (inheritance, association and aggregation), relational models must perform transformations.

1.3.3 Problem Statement

Although the way to avoid the impedance mismatch between objects and relations is to use an object-oriented database, systems often need to store objects in a relational database. Relational databases such as MS Access, SQl Server, Mysql are being used in many software systems from desktop to web based. On the other hand, one of the major works in upgrading is to reuse existing data in Relational database [2].

When developers use the relational database as storage for persisting objects, they could spend much effort in dealing with mapping their object data model with relational database.

This thesis introduces an object relational mapping framework for simplifying combination of object persistence and relational database.

Since framework of object relational mapping is a complete working solution, developer can use one instance of framework to deal with mapping works more than once.

Most of frameworks of object relational mapping use SQL to deal with database programming. SQL is supported by most relational database; it is good at ad-hoc query including searching and sorting operations.

A framework also saves investment. The investment is not only the expense of existing relational database, but also the investment for learning other persisting solution especially OODBMS.

1.3.4 Framework for Object-Relational Mapping

In this thesis, I introduce a framework to map the object-oriented model to the relational model; the framework is called *Framework for Object-Relational Mapping*. It can be used by developers who are using OOP and want to store/read objects into/from a relational database.

The *Framework for Object-Relational Mapping* is a C++ Framework (a set of Classes) intended to help developers in dealing with object relational mapping. It supports most of relational database systems including MS SQL Server, Access, Oracle and MySQL, since I used OLE DB to deal with database related programming. This framework implements some new ways in dealing with object relational mapping including mapping inheritance with inheritance, a solution for aggregation and associations mapping, a cache of object references for constructing objects, and name convention for preserving object maps for object creation.

This framework is written in C++. Developers could rewrite it in other OOP language.

Developers can inherit from a base class provided by my framework to obtain abilities of object Identity management and simple data type mapping. To map complex data types, developers can use similar data structure (classes) provided by my framework. This will save developers' learning time and reduce the work of coding, they have to do.

1.4 Related Work

In this section, I will discuss and review related works about object relational mapping. It includes some useful patterns such as mapping aggregation, inheritance and association. It also lists several frameworks for Java and C++.

1.4.1 Related Pattern

We first consider way of mapping objects to tables.

1.4.1.1 Mapping objects to tables

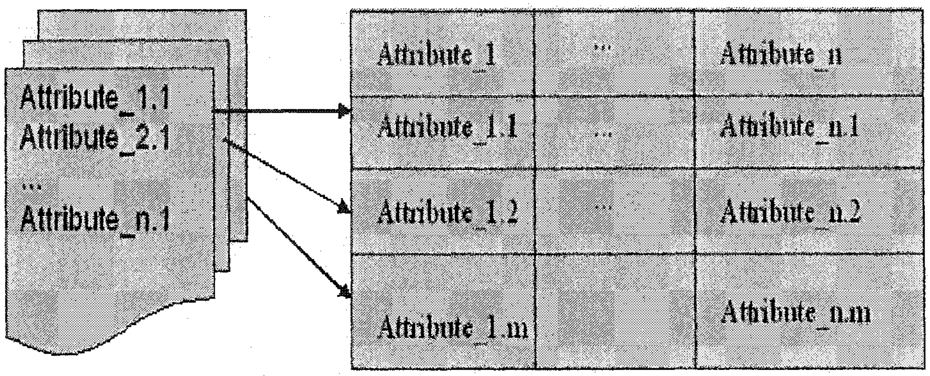


Figure 1 Mapping objects to table

Each attribute of an object becomes a column in a table, and each object becomes a row in a table [16]. The mapped attribute is the kind of ADT supported by the relational database system. Other complex data types need more transformations, as described in the following sections.

This mapping is suitable for objects that have only simple types as attributes.

1.4.1.2 Mapping Complex Data Type

The object oriented data model supports some complex data types such as aggregation, inheritance, and associations. Comparing with simple data type, complex data types need transform work to be applied for object relational mapping. This section will review related works for mapping aggregation, inheritance, and associations.

1.4.1.2.1 MAPPING AGGREGATION

There are two patterns to map aggregation to relational tables: Single Table Aggregation and Foreign Key Aggregation [10][17].

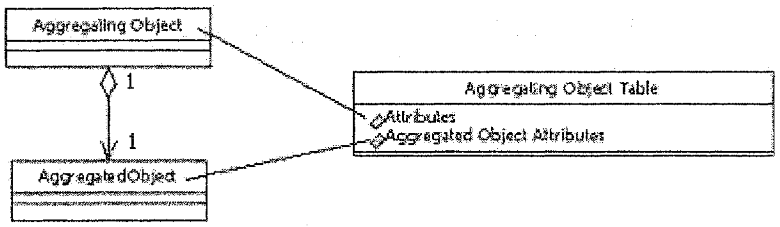


Figure 2 Single Table Aggregation

Single table aggregation consists of putting the aggregated object's attributes into the same table as the aggregating object's.

If the aggregated object type is aggregated in more than one object type, this design results in poor maintainability because each change of the aggregated type requires an adaptation of all of the aggregating object type's database tables.

Scott W. Ambler has analyzed Single Table Aggregation as follows [10]:

1. Performance: The solution is optimal in terms of performance as only one table needs to be accessed to retrieve an aggregating object with all its aggregated objects. On the other hand, the fields for aggregated objects' attributes are likely to increase the number of pages retrieved with each database access, resulting in a possible waste of I/O bandwidth.

2. Maintenance and flexibility: If the aggregated object type is aggregated in more than one object type, the design results in poor maintainability as each change of the aggregated type causes an adaptation all of the aggregating object types' database tables.

3. Consistency of the database: Aggregated objects are automatically deleted on deletion of the aggregating objects. No application kernel code or database triggers are needed.

4. Ad-hoc queries: If you want to form a query that scans all AddressType objects in the database, this is very hard to formulate.

When this situation occurs, the alternative solution, Foreign Key Aggregation [10] [17], is better but is not the best.

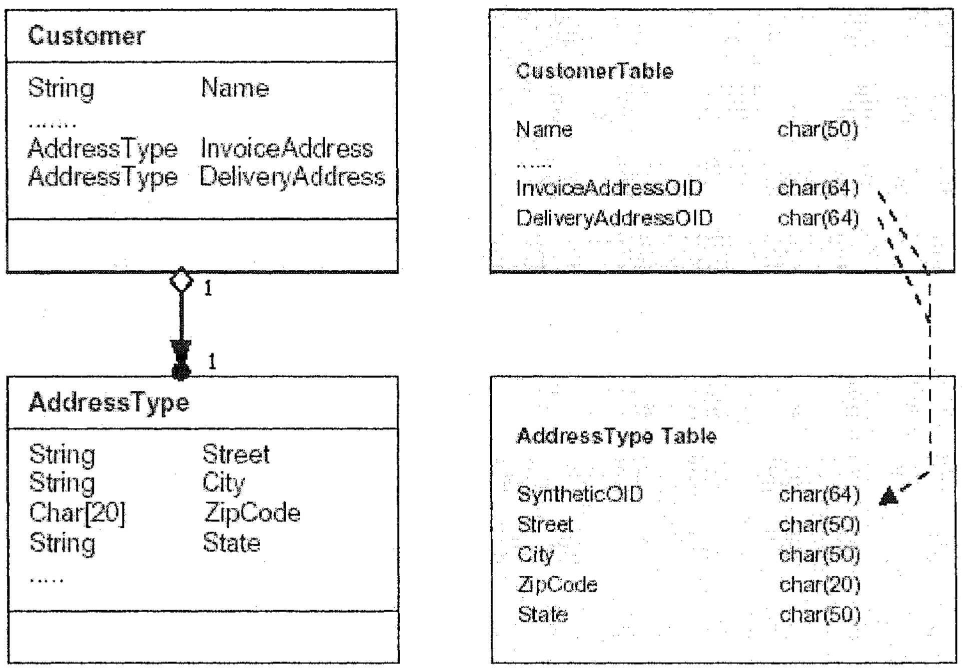


Figure 3 Foreign Key Aggregation

Foreign Key Aggregation uses a separate table for the aggregated object. The Object Identifier is inserted into the table and this object identity is used in the table of the aggregating object to make a foreign key link to the aggregated object.

Scott W. Ambler discussed the consequences in [10]:

1. Performance: Foreign Key Aggregation needs a join operation or at least two database accesses where Single Table Aggregation needs a single database operation. If accessing aggregated objects is a statistical rare case this is acceptable. If the aggregated objects are always retrieved together with the aggregating object, you have to have a second look at performance here.

2. Maintenance: Factoring out objects like the AddressTypes into tables of their own makes them easier to maintain and hence makes the mapping more flexible.

3. Consistency of the database: Aggregated objects are not automatically deleted on deletion of the aggregating objects. To perform this task you have to provide and maintain application kernel code or database triggers. This is also an implementation issue. You have to choose one of these two options.

4. Ad-hoc queries: Factoring out aggregated objects into separate tables allows easy querying these tables with ad-hoc queries.

1.4.1.2.2 MAPPING INHERITANCE

This second type of mapping provides support for implementation inheritance. The following discussion does not cover multiple inheritance.

Typically relational databases provide no support for inheritance. It is therefore necessary to define a strategy for mapping inheritance hierarchies to tables.

There are three approaches to this problem in [10], [17]. As an example, the following object model can be implemented in any of the following three ways.

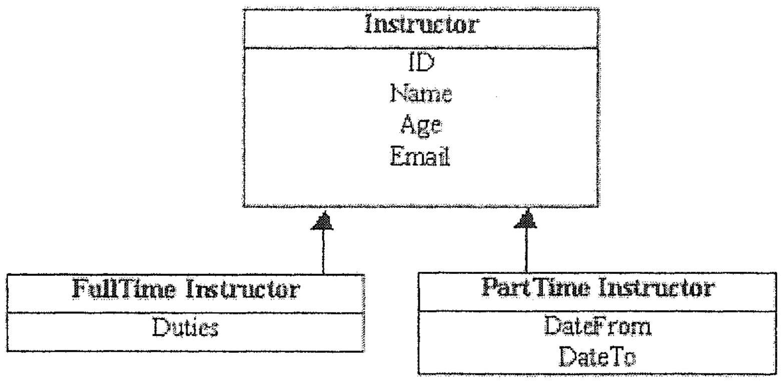


Figure 4 Inheritance

1. A common parent table plus separate subclass tables

   This approach, shown in Figure 5, wastes no space, but requires joins to load objects into memory

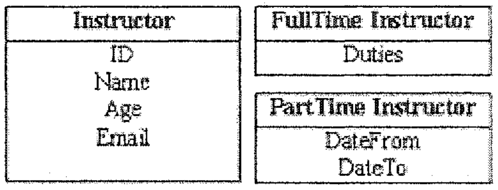


Figure 5 ***Common parent table***

2. A separate self-contained table per class.

   See Figure 6: This approach requires a different table to be accessed depending on the type of the object being loaded

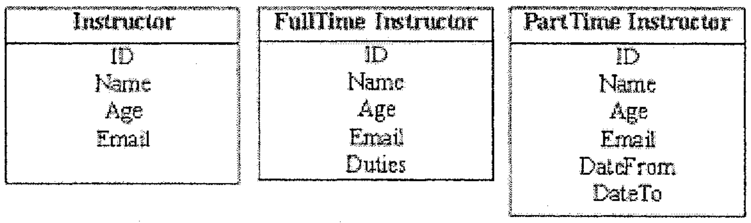


Figure 6 ***Separate tables***

3. One table per single inheritance hierarchy.

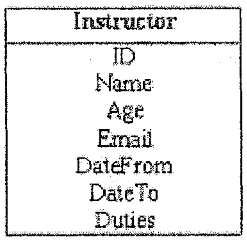


Figure 7 One table

See Figure 7: This approach is wasteful of space. The table will contain lots of nulls but requires only a single table to be accessed to load all objects in the hierarchy.

1.4.1.2.3 MAPPING ASSOCIATIONS

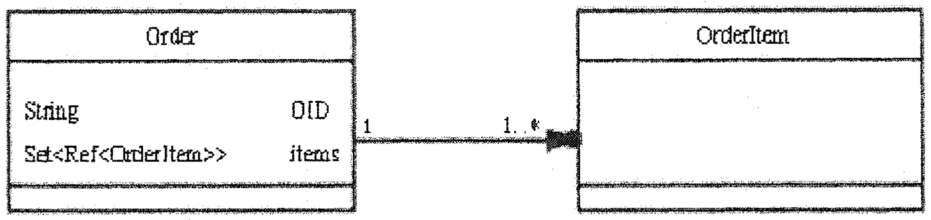
This section presents two patterns used to map associations between objects: Foreign Key Association and Association Table [10] [17].

Consider the classic Order / OrderItem example. A valid Order may have from zero to many OrderItems.

1. Foreign Key Association

The pattern shows how to map 1:n associations between objects to relational tables.

Insert the owner object's OID into the dependent objects table. The OID may be represented by a database key or an Object Identity.



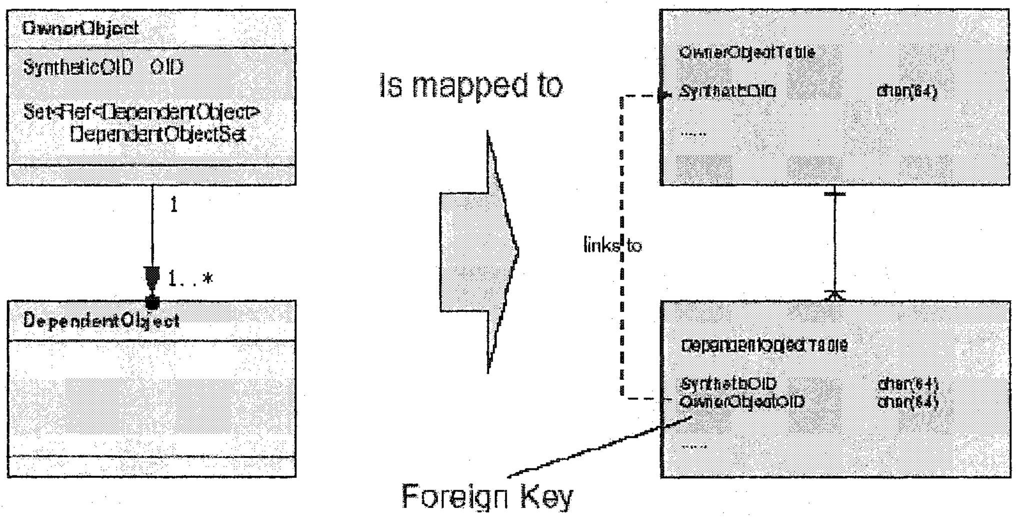
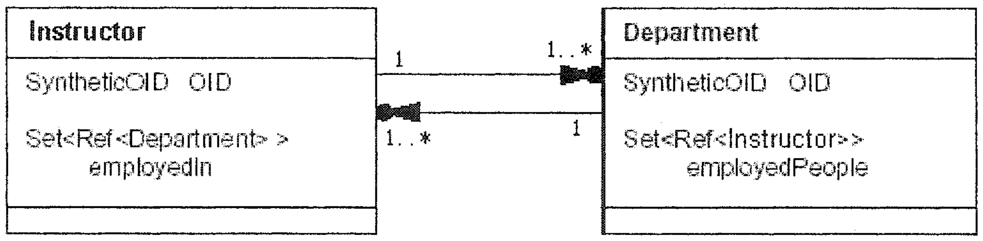


Figure 8 Foreign Key Mapping Association

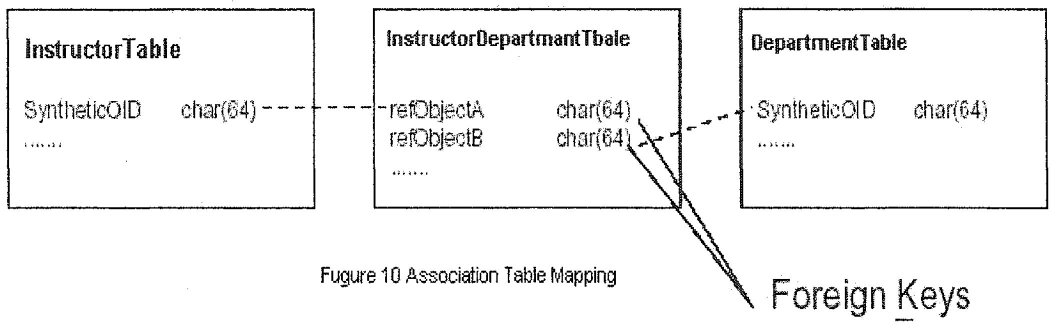
2. Association Table



Instructor
SyntheticOID OID
Set<Ref<Department> > employedIn
1
1..*
1..*
1
Department
SyntheticOID OID
Set<Ref<Instructor>> employedPeople

Figure 9 n:m Associations

As an example we use the n:m association between an Instructor object type and a Department object type. An Instructor can work for more than one department. A department usually comprises more than one Instructor.



Fugure 10 Association Table Mapping

This technique involves creating a separate table containing the Object Identifiers (or Foreign Keys) of the two object types participating in the association and then mapping the rest of the two object types to tables using any other suitable mapping patterns presented in this thesis.

1.4.2 Related Solutions

I have studied several existing solutions about object relational mapping for designing my framework. I have also implemented some of them in my framework design such as layer architecture [7] and introduced my work based on them such as shared object [2].

1.4.2.1 Shared object

The Ratio Group provided an approach using "smart pointers" [18] to handle this problem:

"Fortunately, a common approach to sharing objects and the associated memory management problems in C++ comes to the rescue. Smart pointers are objects that appear externally to behave like exactly like ordinary pointers, but which internally use a reference counting mechanism to note that the object being pointed is shared. Thus, if two smart pointers point to the same object the internal reference count of the pointer will be set to two."

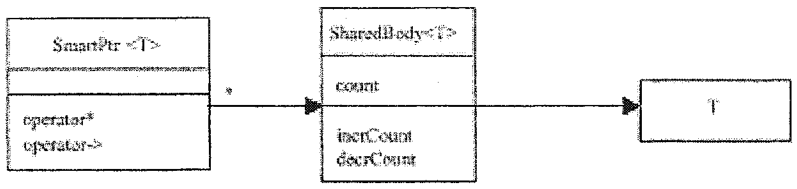


Figure 11 Reference Counting Smarter Pointer

"The smart pointer itself takes on responsibility for memory management, deleting memory only when the reference count to the shared object drops to zero. " [2]

1.4.2.2 Object/Relational Access Layers.

In [7], Wolfgang Keller described how to structure an object/relational persistence subsystem in the global context of a layered architecture for business systems. The architecture consists of two layers: an object layer that contains the infrastructure to persist business objects and a tuple layer that encapsulates a relational database.

This pattern is widely used by most mapping developers such as the Ratio Group [2], SourcePro DB [14] and DataObjects.NET [15].

1.4.2.3 Client-Server Objects.

When object relational mapping framework is using a server side database as storage of objects, it must concern the problem of client server objects. Mark L. Fussel explains the client-server objects issues and gives three approaches as follow:

Object relational mapping intrinsically brings in client-server issue because a relational server is separated from the client application. The client-server issues are nonetheless independent of relational mapping. This conveniently divides the problems into more manageable pieces.

The major issues when dealing with client-server objects is to be able to manage the identity and state of objects on each of the client and server, and then handle the relationships between the two systems' objects. This is different from the relational approach where everything is just a value. For that approach, the client is only getting a simple snapshot of the server state and then must explicitly how the server state should change. The object model tries to provide a more transparent interface for the client, but this actually causes a more complex model and a more sophisticated framework.

Because relational mapping intrinsically involves a client-server system we need to be able to handle the issues with that system. Most of the issues have nothing to do with relational mapping but are instead involved with having multiple ObjectSets between a Server and its Client applications. We need to recognize that the client objects are Replicates of the server objects, that they must keep track of the Identity of their server object, and that there are many issues and approaches for handling concurrency between the multiple clients.

Approach-1

“A client can "check-out" a collection of objects from the server and no other client can see these objects until they are checked back in. This automatically causes each client to have non-intersecting subsets.”

Approach-2

“A client can either “read check-out” or “write check-out” objects from the server. Two clients can check-out the same object as long as they are not both trying to write to it. Alternatively we can prevent dirty-reads as well: a client can only check out an object if there are no write-locks on it and can only write-lock an object if no other client has checked it out. “

Approach-3

“A client can replicate any object from the server but will only be able to write changes back to the server if the server object has not changed since the client produced the replicate. This is the standard optimistic locking” [8].

Although the problem of client-server objects does not belong to object relational mapping, it should be solved in framework design because most developers are using database on server side.

1.4.2.4 Connecting Business Objects to Relational Databases.

Joseph W. Yoder and Ralph E. Johnson introduced several patterns for mapping objects into relational database in [4].

> "These patterns describe how to implement business objects so that they can be mapped to non object oriented databases. There is an impedance mismatch between these technologies since objects consist of both data and behavior while a relational database consists of tables and relations between them. Although it is impossible to completely eliminate this impedance mismatch, you can minimize it by following the proper patterns. The proper patterns hide persistence from the developer so that effort can be spent on understanding the domain rather than in making objects persistent" [4].

These patterns are very useful in dealing with object relational mapping. I implemented them in my framework.

1.4.3 Known Frameworks

1.4.3.1 Java

ObJectRelationalBridge(OJB)[12] is for O/R mapping. It is now a part of the Jakarta project. OJB is an Object/Relational mapping tool that allows transparent persistence for Java Objects against relational databases.

Hibernate [13] is a very popular free O/R mapping framework for Java.

> Hibernate is a powerful, ultra-high performance object/relational persistence and query service for Java. Hibernate lets you develop persistent objects following common Java idiom - including association, inheritance, polymorphism, composition and the Java collections framework [13].

These frameworks only support Java. In 1.4.3.2, I will list two frameworks which support C++.

1.4.3.2 C++

1. SourcePro DB by RogueWave (used to be called DBTools.h++).

> "Rogue Wave® SourcePro™ DB is a complete solution for object-oriented relational database access in C++. SourcePro DB's layered architecture abstracts away the complexity of writing database applications, yet lets you drill down to the native database client libraries when necessary."
>
> "SourcePro DB offers significant benefits, whether you are working with a single database or multiple databases. Code to SourcePro DB's consistent, high-level C++ API, and you'll be able to quickly deliver applications that are reusable with databases from various vendors. There's no need to deal with the details of a particular database vendor's API! "
>
> "SourcePro DB provides general access via ODBC, as well as direct access to the following databases: MySQL Server, PostgreSQL Server, Microsoft SQL Server, Oracle, Sybase, DB2, Microsoft SQL Server, Informix" [14].

It is an object encapsulation of SQL more than a full fledged access layer.

2. DataObjects.NET from http://www.x-tensive.com. DataObjects.NET allows you to focus on the code of the business tier and application data model.

> "DataObjects.NET is an object persistence layer for the .NET Framework. It dramatically decreases development time by handling all persistence-related tasks transparently. It completely supports inheritance (including interfaces), relations and collections, transactions, object queries, full-text indexing and search, multilingual properties and a lot of other features. Moreover, it provides a set of unique services including automatic transaction management and built-in access control system. Its feature set allows building not only the data access tier with it, but the complete business tier of a complex application" [15].

All of these C++ frameworks (SourcePro DB and DataObjects.NET) are not free. They do not provide the source codes to developers. My thesis tries to provide a framework to developers with all of the source codes.

1.5 Conclusion

First, this chapter introduced the problem that this thesis deals with: object persistence. After discussed the background of possible solutions for object persistence, I explained why object relational mapping is important and useful. Then, I explained the motive of this thesis: design and implement a framework to simply work in object relational mapping. It supports most of relational database systems. There are some new ways of dealing with object relational mapping including mapping inheritance with inheritance, one solution for aggregation and associations mapping, cache of object references for constructing objects and name convention for preserving object maps for object creation.

In the second part, I have reviewed the most popular solutions of object relational mapping. Most of these solutions are widely used by Object-relational mapping vendors. Many ideas of mine come from these works.

In Chapter 2, I will discuss design of my framework and focus on my contributions in object relational mapping.

2. FRAMEWORK DESIGN

2.1 Framework

Although object modeling and relational modeling have very different concerns, they are actually extremely compatible. Relational theory is primarily concerned with knowledge and object techniques are primarily concerned with behavior. Mapping between the two models requires deciding how the two worlds can refer to each other. First, I will describe the design of the architecture.

2.2 Layer Architecture For Object Relational Mapping

Relational database programming is complex; storage subsystems are also complex but they are well known abstractions. Object-oriented programming languages are proven concepts. Relational database programming and Object-oriented programming have sufficient complexity. The easiest way is to separate the concepts of object-orientation from those of database programming and to separate the object-oriented database aspects from the relational database aspects. Many existing works are using the same architecture such as [5], [7] and [9].

To achieve this separation, I used Wolfgang Keller's pattern [7] of two layers: Object Layer and Storage Layer.

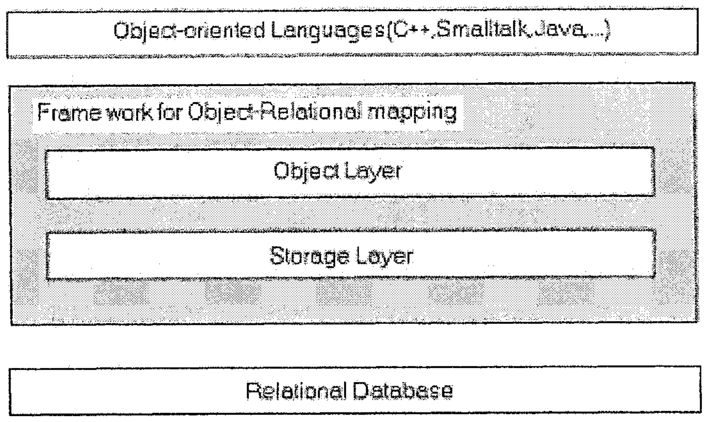

Object-oriented Languages(C++,Smalltalk,Java,...)
Frame work for Object-Relational mapping
Object Layer
Storage Layer
Relational Database

Figure 12 Object and Storage Layer

2.2.1 Object Layer

An "object layer" should have behavior and interfaces similar to an object-oriented database for reasons of convenience as object oriented databases like the "natural extension" of object-oriented languages with persistence features [6] [9]. This layer is the only interface that developers can use in my framework. Developers use these interfaces as functions of classes.

The object layer encapsulates the concepts of object orientation. This layer should hide the developer from the details of storing layout. It should pass persistent object information submitted by the developer to the storage layer (pushing down) and load the persistent object information from the storage layer (popping up). The persistent object information includes different type of data: object Identity, attributes, associations, inheritance, and so on. In my design, developers can use similar ways to deal with all of the complex relationships such as aggregation, associations and inheritance.

2.2.2 Storage Layer

The Storage layer provides an interface to a relational database. It has the following responsibilities: Persistence, Concurrency, Recovery, and Ad Hoc Query.

The Ad Hoc Query is a database concept that developers wrap at the level of their object-oriented language in order to offer their user the equivalent of SQL. Therefore developers have to deal with some form of Object SQL (also called Object Query Language (OQL) [ODMG93]) in both layers. The Ad Hoc Query is a very important reason for using object relational mapping because relational databases provide many functions for Ad Hoc query. To achieve Ad Hoc Query, there must be a parser between the object layer and the storage layer. It will translate OQL into SQL and *vise versa.*

The storage layer encapsulates all database operations related to object persistence. The storage layer provides an interface to the object layer. The object layer can ask the storage layer to save or retrieve objects without knowing the storage layout in the relational database.

To begin with, I will discuss how I deal with major problems in designing framework of object relational mapping. This discussion will concentrate on my original work.

2.2.3 Mapping Aggregation

In Chapter 1, I have reviewed two patterns for mapping aggregation: single table aggregation and foreign key aggregation. Although Ambler gives a lot of advantages of foreign key aggregation and also mentions that there is a pitfall in consisting of the database, additional work is needed to maintain consistency of the database. I try to modify the solution to avoid this additional work. I introduce a class called "relationobject", which is provided by my framework, to manage all of aggregations. This object is responsible for releasing the resources of the aggregated object.

2.2.4 Mapping Inheritance

In 1.4.1.2.2, I have introduced three approaches to mapping inheritance: a common parent table plus separate subclass tables, a separate self-contained table per class and one table per single inheritance hierarchy.

In my framework, I combined three ways together to give flexibility to developers. This idea took advantage of the concepts of object oriented programming, thus I am using inheritance to solve mapping inheritance. When objects need to be persistent, my framework will call their base object to make the persistent in advance. This calling procedure will follow the inheritance hierarchy. Moreover, I recommend a common parent table plus separate subclass tables for mapping inheritance because it not only saves database spaces but also preserve the relationship layout in database.

2.2.5 Mapping Associations

In chapter 1, there are two patterns to map associations between objects: Foreign Key Association and Association Table. My framework will use Association Table to map associations instead of Foreign Key Association because Foreign Key Association only supports 1:n associations. However, Association Table will face a problem about shared objects that is clarified in Section 2.2.5.1.

2.2.5.1 Shared Object

It is not difficult to understand the use of collection attributes to implement associations in the object model. Often, the contents of the collection are pointers that point to related objects. At runtime, objects within an object model are inherently shared - it is possible for two objects to point to the same object. As we do not want two copies of the same object to be in buffer at the same time (this could cause problems updating database), any pointers used must point to the same object in buffer. This presents us with a memory management problem.

My framework introduced another way to solve the problem which is to construct a cache to hold all references to the objects. In my framework, all of the persisting objects are derived from the same base object. I provided a base class called CpersistClass as structure of the base object.

A cache, a container of CpersistClass objects, is built when my framework starts. When an object needs to be constructed, the system first searches the cache to see if the requested object has already been constructed. If it has, the system passes the object reference to the caller; if not, the system constructs a new instance and saves the reference into the cache. Due to the association problem (shared object), object creation, which is an important issue in object relational mapping, becomes complex. In 2.2.5.2, I will discuss the solutions of this complex problem.

2.2.5.2 Object Creation

Loading an object from a row in a table is not a simple task, because we must determine the class of the loaded object. Next, I will introduce two of my solutions of loading an object (object creation).

One solution is to use the *Objects map.* One objects map must be established when the application starts. The map maps each class name to a static creation function. Another map, the *object-table map*, is needed to link class names to table names. The creation function creates a blank object instance and then fills its contents with a particular table according to the object-table map. Obviously, an object-table map has the same life cycle as the database application and extends beyond the application life cycle.

Since it interacts with database system, the storage layer must provide a mechanism to support transaction and concurrency control.

Figure 14 shows the result of instructor's example discussed in chapter 1.

My second way to create objects does not need any maps in memory. As

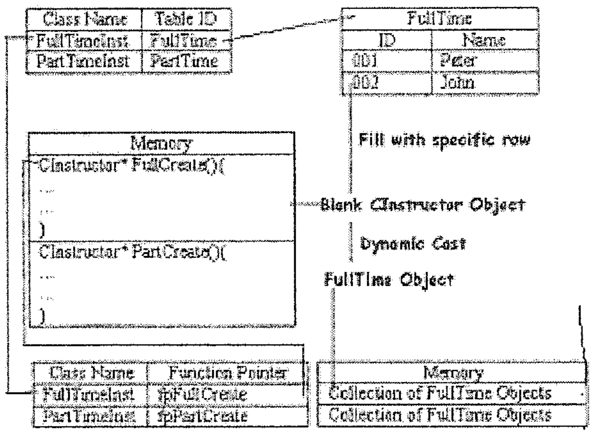


Figure13 Object Creation

all the objects are stored into a relational database, we can store the object's map into the relational database as well. This will hide the entire table layout in the storage layer from the application programmer and the object layer. To save the objects map into a relational database, we can use a naming convention to preserve the structure of the objects map. I used class names to define the table name and add a relation column to connect related objects table and save complex data type into new table connecting with certain column with certain name.

To explain clearly, I use instructor/courses association as example. Next is the result of instructor/course.

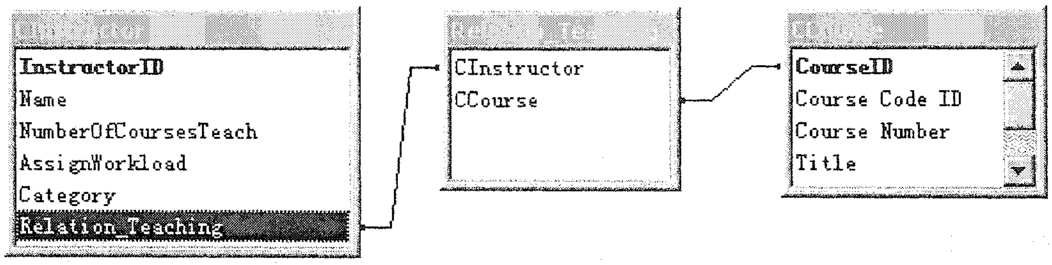


Figure 14 Name Convention

When developers pass the name of constructed object (“CInstructor”), my framework uses it as the table name to read attributes for constructed object. Then my framework constructs the association using “Relation_Teaching” as foreign key to retrieve all related object (CCourse) for CInstructor objects.

This concludes the discussion of my contributions of object relational mapping. These contributions were implemented in my framework design.

2.3 Conclusion

In this chapter, I have introduced some new solutions that have been implemented in my object relational mapping framework as contributions including mapping inheritance with inheritance, one solution for aggregation and associations mapping, cache of object references for constructing objects and name convention for preserving object maps for object creation.

In chapter 3, I will clarify the implementation of the design based on the discussion in this chapter.

3. Implementation of Design

Having finished the discussion of the design of framework for object relational mapping, I will explain how these principles are implemented. I will use top-down order, from the object layer to the storage layer.

3.1 Object layer

The object layer interacts with Object Oriented programming for mapping objects to and from the storage layer. Programmers use interfaces, provided by the object layer, to save and read object to and from a relational database. The relational model, unlike the object model, does not support object identifiers. Whenever developers construct an instance of an object from a class, the compiler creates a unique identifier (object identity) for it. Usually, the identifier is the object address, and developers can use it. Whenever an object is persistent, it is important to record the uniqueness of that object. Since all objects are unique in an object oriented system, it is important to give an object a unique object identifier called its *OID*. Consequently the persistent object must have the ability to maintain its OID during the mapping process. For saving and reading objects into/from relational database, the framework should provide a mechanism. For handling complex data type such as aggregation and association, the framework should also provide some mechanisms. As a beginning, I will discuss the design of pushing and popping data with a simple data type.

3.1.1 Push Down/Pop up

No matter what kind of complex attribute the persistent object has, the relational database supports only single value data types (ADT) (i.e., a column of a table). The persistent object's content will eventually be presented in the form of columns with a single data type. Each persistent object needs to push its content to the lower layer (storage layer) when saving, and to pop up its content from the lower layer when reading.

So, I introduce a broker for pushing down and popping up. This broker is a set of data structures (CColumnList) that holds all of the ADTs (Single Value) of the persistent object. Next is the definition.

```
class CColumn:public CObject
{
public:
        CColumn(void);
        CColumn(CColumn* pColumn);
        ~CColumn(void);
        CString sName;
        WORD  dtype;
        int    bPrecision;
        int    bScale;
        VARIANT value;
};
typedef CTypedPtrList<CObList,CColumn*>CColumnList;
```

Obviously, this schema (`CColumnList`) is compatible with table schema. It can be used as a broker between objects and tables.

Having finished the definition of data structure, I will introduce the implementation of object identifier management because all of the mapping mechanisms are based on it.

3.1.2 OID Manager

First my framework creates a unique number. Then, it combines the class name with unique number to generate a key. Before using the key, my framework checks it with persistent objects in the database to ensure that it is unique. In my framework, the OID management procedure is invisible to developers.

Next, I introduce a class that is responsible for completing push down/pop up and OID management in my framework.

3.1.3 CpersistClass

This class has a broker (CColumnList) to deal with push-down and pop-up. An OID manager is included as well. This class also involved the object creation that I will discuss later. Design of CpersistClass is as follows:

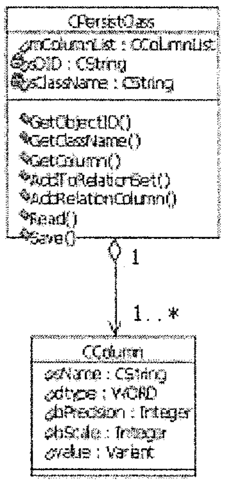


Figure 15 CPersistClass

The list of columns holds all of the ADTs (Single Value) of the persistent object. It provides the CpersistClass with the abilities to deal with one-to-one relation between an aggregating object type and an aggregated object type.

1. The attributes as follows:

sDID

is the unique identifier for the object.

sClassName

identifies the runtime class name of this object.

mColumnList

is the broker for pushing down and popping up.

2.The public methods

Save(Push Down)

Write the object data to the database. It will update or insert rows as necessary.

Read(Pop Up)

Return single instance of a class with data in the columns from the database.

The implementation is as follow:

```
void CPersistClass::Read()
{
        CColumn* pColumn=this->GetColumn("ObjectID");
        if(pColumn!=NULL)
                this->sDID=CString(pColumn->value.bstrVal);
        Prepare(); //Read OID
}
void CPersistClass::Save()
{
        ClearList();
        Prepare();    //Save OID
        CColumn* myColumn=new CColumn();
        myColumn->sName="ObjectID";
        myColumn->dtype=130;
        myColumn->bScale=100;
        myColumn->value.vt=VT_BSTR;
        myColumn->value.bstrVal=this->sDID.AllocSysString();
        this->mColumnList.AddHead(myColumn);
}
```

Each domain object is a subclass derived from this CpersistClass. Each object therefore inherits these methods (OID management and push down/pop up) to perform its mapping transformations. Although it requires some code in each domain class that is database specific, this code is segregated, and is easy to find and maintain.

3. Map Inheritance

The CpersistClass is sufficient for simple data type mapping, but it can map inheritance with different ways discussed in chapter 2.

Developers can override Save method in different ways to map inheritance. I will discuss the detail in next section because it will use the same standard interfaces of object layer.

To map other complex data types (association and aggregation), I will introduce a new class called RelationObject in next section.

3.1.4 RelationObject

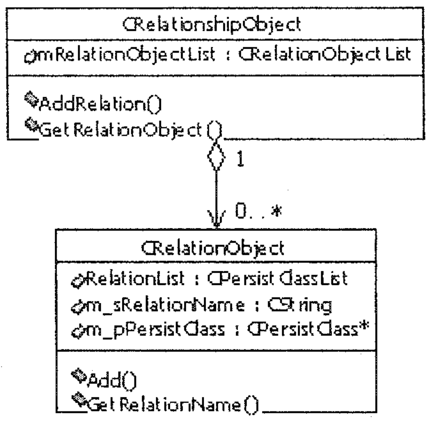

CRelationshipObject
mRelationObjectList : CRelationObjectList
AddRelation()
GetRelationObject()
1
0..*
CRelationObject
RelationList : CPersistClassList
m_sRelationName : CString
m_pPersistClass : CPersistClass*
Add()
GetRelationName()

Figure16 RaltionObject

The CpersistClass is unable to perform complex data type mapping so I introduce the RelationObject class. For mapping the complex data type, this class has a container to hold the complex data. The complex data types includes Sets, Lists, Array, Aggregation (class or list of class), Association (list of object reference). One RelationObject represents each complex data type in the object. To hold the complex data, I constructed a Set of RelationObject called CRelationshipObject. With CRelationshipObject, I can use one data structure to solve all of complex data including aggregation (4.4) and association (4.5).

Definition to RelationObject:

```
class CRelationObject : public CObject
{
public:
        CPersistClassList RelationList;
        CString m_sRelationName;
        CPersistClass* m_pPersistClass;
        void ClearList(void);
public:
        HRESULT Add(CPersistClass* pPersistClass);
        CString GetRelationName(void);
};
typedef  CTypedPtrList<CObList,CRelationObject*>CRelationObjectList;
```

Attributes:

RelationList	:List of CpersistClass as container.
m_sRelationName	: a identifier to indicate relation to the owner object.
m_pPersistClass	: Reference to the owner object.

Methods:

Add	: Push down an element into container.
GetRelationName	: Returns the Relation name.

Definition to CrelationshipObject:

```
class CRelationshipObject : public CObject
{
public:
        CRelationshipObject();
        virtual ~CRelationshipObject();
        CRelationObjectList mRelationObjectList;
public:
        void AddRelation(CPersistClass* PersistClass1,CPersistClass*
PersistClass2,CString sRelationName);
public:
        CRelationObject* GetRelationObject(CString sRelationName);
public:
        void ClearAll(void);
};
```

Attributes:

mRelationObjectList: Container of all of complex data.

Method:

AddRelation: Add a complex data to container.

GetRelationObject: Returns the data according relation name.

With the CrelationshipObject, my framework can map all of the data types that any object model could have. Figure 17 is the workflow of persisting object.

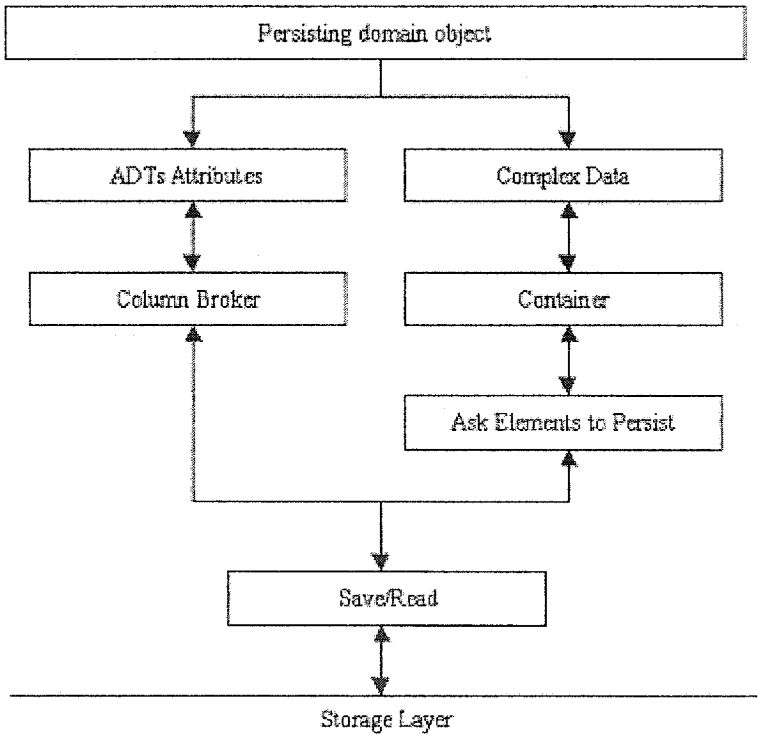


Figure17 Persisting object

3.1.5 Persisting Object

Principles:

Each domain object obtains the column broker by inheriting from the CpersistClass. A domain object with complex data can declare a variable of CrelationshipObject and can then add the push-down operation for the complex data into the save method as well as the pop-up operation in the read method.

Aggregation: Consider the following sample class:

```
class CAggregatingClass : public CPersistClass
{
       CAggregatedClass mAggregatedClass;        //Aggregation
       CRelationshipObject mRelationshipObject; //container for the
mAggregatedClass
}
```

CAggregatingClass has a mAggregatedClass variable with a class data type. For persistent CaggregatingClass, system adds a mRelationshipObject variable of CrelationshipObject. Then the overridable methods inherited from CpersistClass must be rewritten by users for implementing the mapping of mAggregatedClass into the database.

```
void CAggregatingClass::Save()
{
       CPersistClass::Save();     //class base class method
       mAggregatedClass ->Save(); // save aggregated object into
       database
       mRelationshipObject.AddRelation(mAggregatedClass); //link
       persisted aggregated object
}
```

Other kinds of complex data can be processed in the same way.

3.1.6 Object Creation

Object creation (Object loading) is one of difficult works in object relational mapping. It has two aspects: determine the class of the loaded object and deal with shared object. Next, I will explain how I solve these problems.

1. Developers determine the class of loaded object.

My framework is set of classes; developers who use my framework must include my framework into their source code. That means my framework is part of their application. Taking advantages of this, developers could communicate information of class between classes of the framework my frameworks. Consider the next example.

```
class_instructor* pInstructor=new class_instructor();
myPersistenceFrameWork.ReadObject((CPersistClass*)pInstructor);
```

First, developers construct a blank object and pass it to my framework.

Then, my framework loads the contents of the passed object from the storage layer and returns it to developers. This implementation does not need any objects map in memory.

2. Object loading.

My solution is to use a cache that holds all of constructed object references. Here is the definition of the cache.

```
CPersistClassList mPersistClassList;
```

Since all of persisting objects are derived from CPersistClass, this provides the capability of caching all objects references within a set.

CPersistenceFrameWork::ReadObject() is responsible for adding constructed object references into cache.

3.1.7 Standard interface

In my framework, the object layer implements some interfaces to developers. Programmers use these interfaces to save and read objects to and from a relational database. They are SaveObject, ReadObject, SaveRelationShip, AddToRelationSet, GetRelationShip and Execute.

SaveObject: Users save the contents in the passing column broker into database. The definition for the SaveObject is as follows:

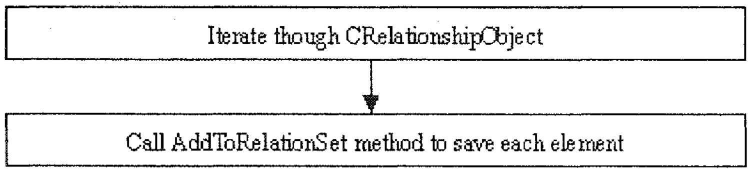
```
HRESULT CPersistenceFrameWork::SaveObject(CPersistClass* pPersistClass)
{
        SaveObject();
        GetDataBaseConnection();
        PrepareSQL();
        Execute();
}
```

ReadObject: In my design, the ReadObject is very straightforward. First, it searches for the required object within the sink of persisted objects using the passed OID. If the object is found, it returns a reference to it. Otherwise, it pushes the passed object reference into the cache. The definition for this method is as follows:

```
HRESULT ReadObject(CpersistClass* pPersistClass)
{
        CpersistClass* m_pPersistClass=GetPersistClass(pPersistClass);
        if(m_pPersistClass!=NULL)
        {
                delete pPersistClass;
                pPersistClass=m_pPersistClass;
        }
        else
                mPersistClassList.AddTail(pPersistClass);
        return S_OK;
}
```

SaveRelationShip: The method is used to save the complex data into the database. It iterates through each element in the CrelationshipObject to save them into database. The definition for SaveRelationShip is as follows:

HRESULT SaveRelationShip (CrelationshipObject* pRelationshipObject)


Iterate though CRelationshipObject
Call AddToRelationSet method to save each element

AddToRelationSet: This method links the owner object to the complex data using a relationship table.

With this method, developer can build the schema as shown in Figure 18. The declaration for AddToRelationSet is as follows:

```
HRESULT AddToRelationSet(CpersistClass* pOwner,CpersistClass*
pDependent,Cstring sRelationName),
```

GetRelationShip: This method is used to build a CrelationshipObject object for persistent object based on its mapping schema in the database.

It works closely with the database system via the database access layer and returns a CrelationshipObject object as the container that holds all complex data of the owner object. The declaration is as follows:

```
HRESULT GetRelationShip(CPersistClass* pOwner,CRelationshipObject*
pRelationshipObject);
```

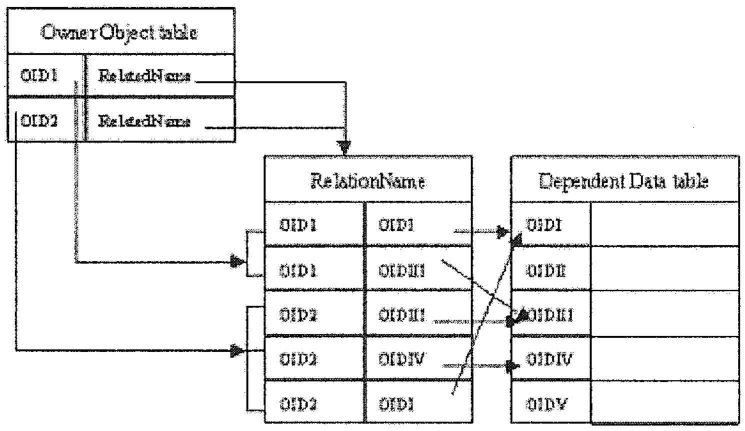


Figure 18 Mapping Schema

Execute: Fetching objects from database needs a collection data structure to contain the fetched objects. Furthermore, a kind of OOQL (Object Oriented Query Language) should be supported by the storage layer. The developer uses the OOQL to send a request to the storage layer. The storage layer returns a collection of requested objects. The developer's code will iterate the collection for fetching objects. So, I provide an Execute method for dealing with fetching. The declaration for Execute is as follows:

```
HRESULT Execute(Cstring sOql,CobjectSet* pObjectSet),
```

The CobjectSet is the collection containing objects as the result of execution of OOQL. The definition as follows:

```
class CpersistClass;
typedef CtypedPtrList<CobList,CpersistClass*>CpersistClassList;
class CobjectSet
{
public:
       long GetCount();
public:
       HRESULT GetAt(POSITION pos,CpersistClass* pPersistClass);
private:
       CpersistClassList mPersistClassList;
public:
       HRESULT Bind(void);
       POSITION MoveNext(POSITION pos);
       POSITION MoveFirst(void);
};
```

To fetch a set of objects, the developer can use the following code:

```
       CobjectSet* pCObjectSet=new CobjectSet(sSql);
       hr=pDoc->myPersistenceFrameWork.Execute(sOql,pCObjectSet);
       POSITION pos=pCObjectSet->MoveFirst();
       while(pos!=NULL)
       {
              Object* pObject=new Object();
              hr=pCObjectSet->GetAt(pos,(CpersistClass*)pObject);
              pos=pCObjectSet->MoveNext(pos);
       }
```

3.2 Storage Layer

As I discussed in the last section, all persistent objects use the standard interface of the storage layer. The standard interface provides read, save and delete operations. Using the standard interface, the developer can populate objects from a database and can save the corresponding data back to the database. In my previous design, a column broker conveyed the pushed down or popped up data. Consequently, the storage layer must be able to perform type conversions in order to convert the types of values between broker and database. To interact with the database, the storage layer also needs a SQL parser to build the actual SQL calls to the database. Under the SQL parser is a connection manager that sets up connections to the desired database. Using these internal layers in the storage layer, the developer can ignore the details of the storage mechanism.

Another technique to prevent objects from loading more than once is to construct a reference cache that holds references to all loaded objects. Whenever an object is required, the storage layer checks this cache before creating the required object.

3.2.1 Standard interface

OpenDataSource: As the connection manager, OpenDataSource method requires a connection string to connect the desired database. In my design, I use the OLEDB to build the database access layer, which is the lowest sub-layer in the storage layer. It interacts directly with the database system and hides all details of database operations from users. Users can access most database systems using this method.

GetObjectWithRelation: The object layer uses this interface to fetch related objects by name of relation.

RemoveObject: The object layer uses this interface to delete objects.

OpenTable, CreateTable and GetTable: Object layer uses these interfaces to get and save information from/into relational database.

After introducing these principles, I will describe their implementations in the next section.

3.3 Conlusion

This chapter focuses on my works on implementation to design of framework for object relational mapping. The purpose of implementation is to hide the relational model layout from application developer and make developer take advantage of both OO programming and relational database systems. In chapter 4, I will use the teaching assignment planner project as a case study to test my framework.

4. Case Study

In this chapter, I will use one of my projects: Teaching assignment planner to demonstrate how my framework worked in an OOP application for save/read objects to/from relational database. A requirements document for this project, written by Peter Grogono, is provided in Appendix A.

4.1 The Object Oriented Data Model

Figure 99 shows the object data model of teaching assignment planner project in the form of a class diagram.

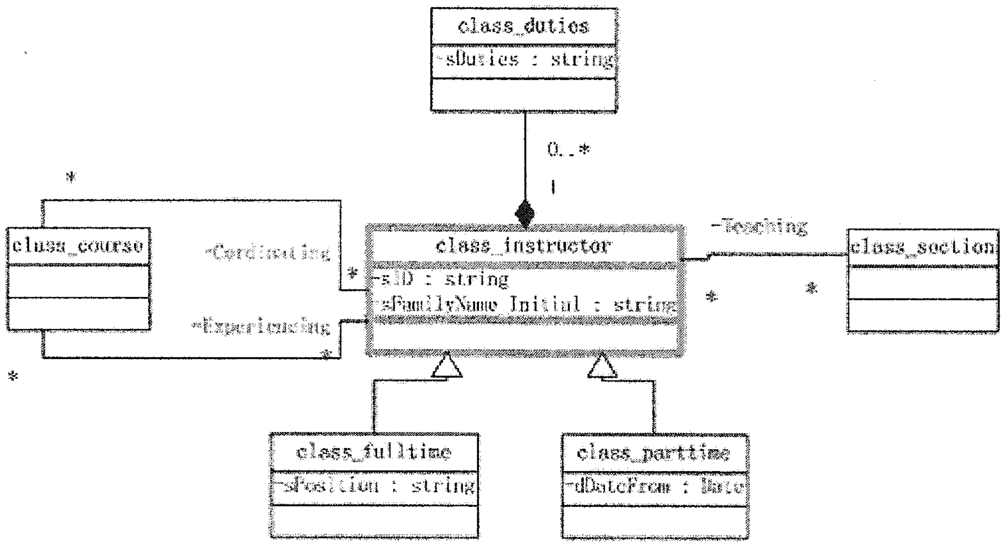


Figure19 Object Oriented Data Model

4.2 Mapping Simple Data Type

In this section, I will demonstrate how to map simple data types in class instructor.

Class Definition:

```
class class_instructor : public CPersistClass
{
        virtual void Read;  //override method
        virtual void Save();//override method
private:
        string sID;         //attribute with simple data type
        …
}
void Read()
{
        CPersistClass::Read();      //OID Management
        //Pop up InstructorID Column
        CColumn* myColumn=this->GetColumn("InstructorID");
        if(myColumn!=NULL)
                this->sID=CString(myColumn->value.bstrVal);
        ...
}
void Save()
{
        CPersistClass::Save();      //OID Management
        //Push down InstructorID Column
        CColumn* myColumn=new CColumn();
        myColumn->sName=_T("InstructorID");
        myColumn->value.bstrVal=this->sID.AllocSysString();
        this->mColumnList.AddTail(myColumn);
        ...
}
```

CPersistClass::Read() and CPersistClass::Save() work with the OID manager to maintain the object identifier for the persisting object. Rests of codes are straightforward and readable.

4.3 Mapping Inheritance

Class instructor has two subclasses, fulltime and part-time instructor. For mapping these objects into relational database, developers could follow this pattern:

Class definition

```
class class_instructor : public CPersistClass
{...}
class class_fulltime : public class_instructor
{...}
class class_parttime : public class_instructor
{...}
void class_fulltime::Read()
{
        class_instructor::Read();
        CColumn* myColumn=this->GetColumn("Duties");
        if(myColumn!=NULL)
                this->sDuties=CString(myColumn->value.bstrVal);
   ...
}
void class_parttime::Read()
{
        class_instructor::Read();
        CColumn* myColumn=this->GetColumn("DateFrom");
        if(myColumn!=NULL)
                this->sDateFrom=CString(myColumn->value.bstrVal);
   ...
}
```

Subclasses call their base class read/save method to save the attributes that inherit from base class into the same table.

Developers may map these classes a common parent table plus separate subclass tables using RelationObject class to build a connection between class instructor and its sub classes. In section 4.4, I will demonstrate how to use RelationObject to build relation between tables.

4.4 Mapping aggregation

The Instructor class has the most complex data structure. It has aggregation (duty). There is also a class_duties to present duty information. The instructor class has an attribute with data type of class_duties.

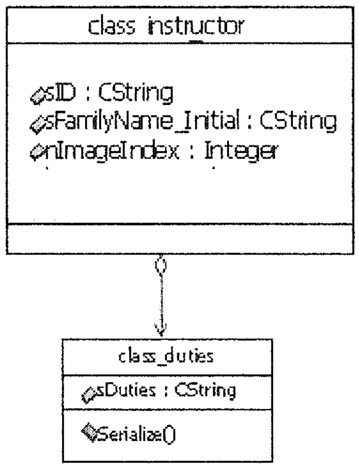


Figure20 Aggregation Example

Definition of Classes

Make classes derive from CpersistClass.

1.class_instructor

```
class class_instructor : public CPersistClass
{
       class_duties c_duties;
       ...
}
```

2.class_duties

```
class class_duties : public CPersistClass
{...}
```

Override Save/Read method

```
void class_instructor::Save()
{
        CPersistClass::Save();
        ...
        //saving Duties
        c_duties.Save();
        mRelationshipObject.AddRelation((CPersistClass*)this,
        (CPersistClass*)&c_duties,"Duty");
        ...
}
```

First, we call the c_duties's Save method to save it's contents into database. Then, we build a relation called “Duty” to link instructor and duty object.

Comparing with Save method, the Read method is more complex.

```
void class_instructor::Read(CPersistenceFrameWork* pPersistenceFrameWork)
{
        CPersistClass::Read();
        pPersistenceFrameWork->GetRelationShip((CPersistClass*)this,&this-
        >mRelationshipObject);
        CRelationObject*
        pRelationObject=mRelationshipObject.GetRelationObject("Duty");
        POSITION pos=pRelationObject->RelationList.GetHeadPosition();
        while(pos!=NULL)
        {
                CPersistClass* pPersistClass=pRelationObject-
                >RelationList.GetNext(pos);
                c_Duties =(class_duties*)pPersistenceFrameWork-
                >GetObjectWithRelation(&c_Duties,pPersistClass);
                c_Duties->Read();
                break;
            }
    }
```

In my framework, I use CrelationshipObject structure to hold all the complex data types. For retrieving complex data from storage layer, developers can use CrelationshipObject.

First, we call GetRelationShip method to obtain the CrelationshipObject object of instructor. Then, because we want to fetch the related duty object, we use “Duty” as parameter to obtain the duty RelationObject. Using the RelationObject(Duty), we can get the related duty object by calling GetObjectWithRelation provided by my framework.

4.5 Mapping Associations

I will explain the implementation of mapping associations with the Instructor object type and Section object type. One instructor may have zero or more sections to teach (Teaching Association).

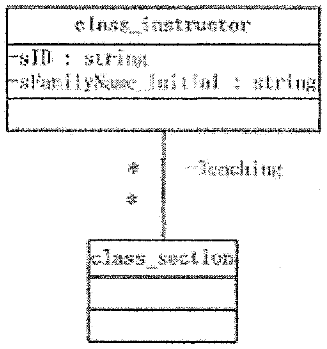


Figure21 Association:Teaching

Definition to Class

Make classes derive from CpersistClass.

1. class_instructor

```
class class_instructor : public CPersistClass
{
       CSectionList listOfSectionTaught;
       ...
}
```

2. class_section

```
class class_section : public CPersistClass
{...}
typedef CTypedPtrList<CObList,class_section*>CSectionList;
```

Override Save/Read method

```
void class_instructor::Save()
{
        CPersistClass::Save();
        ...
        //saving associated sections
        POSITION pos=this->listOfSectionTaught.GetHeadPosition();
        while(pos!=NULL)
        {
            class_section*  pSection=this->listOfSectionTaught.GetNext(pos);
            pSection->Save();
            mRelationshipObject.AddRelation((CPersistClass*)this,(Cpersist
            Class*)pSection,"Teaching");
        }
}
```

In the discussion about principles of mapping associations, I showed that associations in object model are often presented in the form of a collection attributes in object oriented programming. To construct associations between class_instructor and class_section, I use a type-safe list (listOfSectionTaught). This structure stores all of related sections' references. Developers may iterate the collection and call the Save method for every item in this collection.

The Read method is very similar to the one in mapping aggregation (just remove "break;").

Implementing class_section in the same way, we can build a n:m association between class_instructor and class_section.

As we can see, I used a similar way to map aggregation and associations. Developers can take advantages of this way to simplify their implementation.

Figure 22 shows the result of mapping.

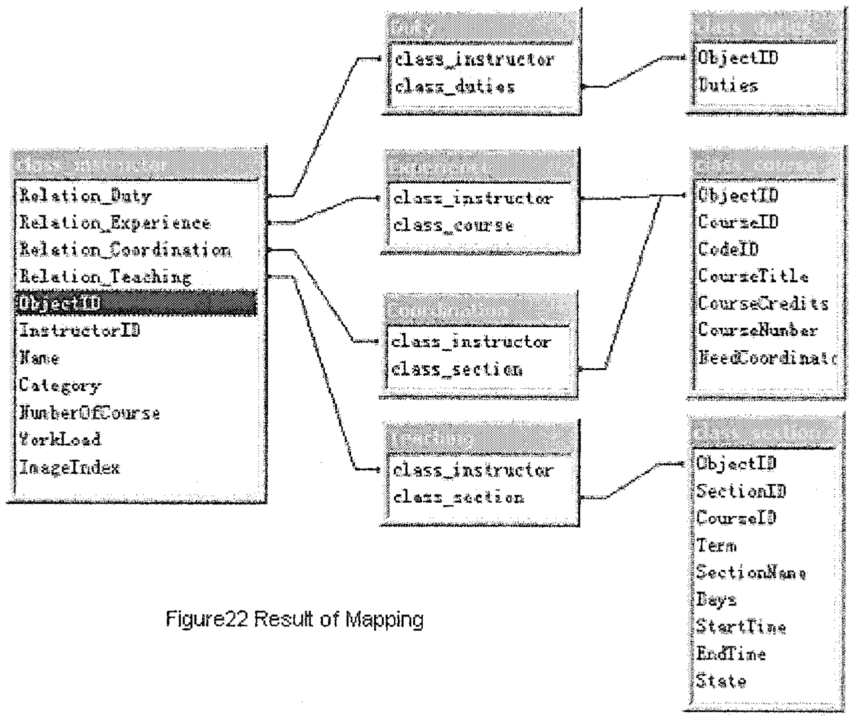


Figure22 Result of Mapping

4.6 Performance Results

Having finished introducing the mapping work in teaching assignment planner project, I will analysis the performance of my framework: how my framework helps developers in dealing with object persistence with relational database.

As we can see in section 4.2, developers save and read attributes of class instructor using the broker that my framework provided without having to use relational database programming. On the other hand, developers do not need to define a table in the database containing instructor objects. My framework also maintains object identifier for developers.

For mapping complex data types into database, developers ask each element of complex data types to map itself and my framework to build the relationship to each of them. According to the developers' request, my framework builds tables for accommodating each data type and creates associate tables to link object tables together. When fetching objects' complex data types from database, developers do not worry about the problem of shared objects. My framework is responsible for handling this problem. There is a cache that holds all of created objects references in my framework. Developers can take advantage of this cache for memory management.

To gain advantage from my framework, developers must derive their objects from the same base class (CPersistClass) and override the save/read methods. As a result, the persistence mechanism was encapsulated into class definition. The resulting code is easy to maintain.

4.7 Conlusion

In this chapter, I used the teaching assignment planner project as an example to demonstrate my framework's implementation. The purpose of this chapter is to prove that my framework is useful in object relational mapping work.

In chapter 5, I want to clarify some mapping problems in respect of multiple user environment.

5. SERVER SIDE OBJECT

One significant advantage of object relational mapping is making objects accessible to multiple users. Client applications can operate objects persisted in relational database system running on a certain server. Every client is only getting a subset of the server side objects so the major issue of the server side objects is to manage the state of objects on each of the client and the server.

5.1 ObjectSet

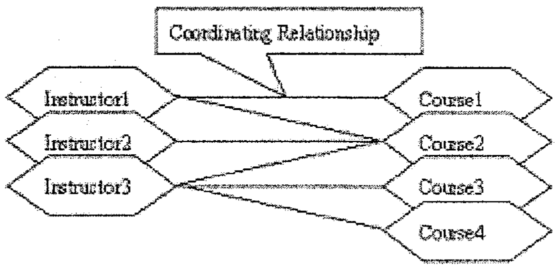


Figure23 ObjectSet

An ObjectSet is a collection of objects based on the relationship between these objects. An ObjectSet is self-contained and isolated from other ObjectSets. There is no communication between two ObjectSets. An example of ObjectSet is the Instructor objects and related coordinated course objects.

5.2 Server Side Object

When objects were persisted into a relational database on a certain server, they can be accessed by all of clients connecting to the database server. The completed ObjectSet on that server is called *server side ObjectSet*.

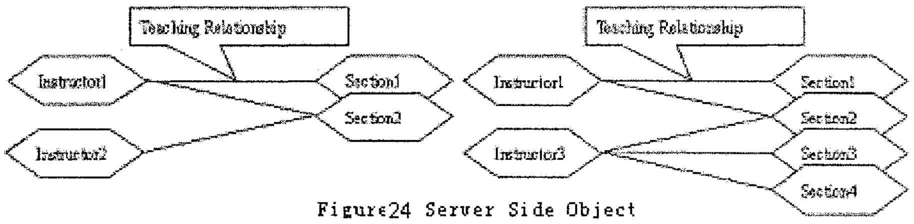


Figure24 Server Side Object

Each client has its own ObjectSet but this is only a temporary working-copy of the server's true ObjectSet. Each object in a client's ObjectSet is a replica of a server set object, and the whole client ObjectSet forms a partial or complete replica of the server's ObjectSet. The next example shows two ObjectSet belong to the different clients that derived from the same server side ObjectSet.

If each client deals with a non-intersecting subset of the servers ObjectSet then we can have easy and "perfect" concurrency: clients can cause changes to their ObjectSet replicas and propagate these to the server without worry about a conflict with another client.

For most applications it is very unlikely that clients will always using a non-intersecting subsets. For the cases where the ObjectSet on clients overlap there must be some type of concurrency control between the clients and the server. Concurrency controls can depend on granularity, visibility, pessimism, functional dependency, and many other axes[9].

5.3 Concurrency

The major problem of server side object is to keep concurrency between clients and server. In the last example, there is an overlap between these two ObjectSet. For instance, any changes to Instructor1 in both ObjectSet could interfere with each other. Next I will discuss 2 solutions for coping with this problem.

Solution:

1. After loading an object from server, the client marks the object as locked. Any locked object can not be loaded until that client unlocks it. In previous sample, after Client A loaded Instructor1 and it's related Section1 and Section2, Client A would lock these loaded objects. When Client B wants to build its ObjectSet, Client B could not load Instructor1, Section1 and Section2. Figure 25 shows The Client B's ObjectSet.

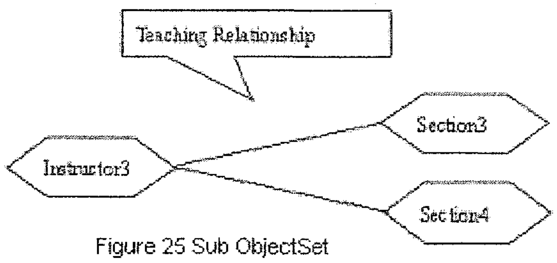


Figure 25 Sub ObjectSet

2. Add a timestamp attribute, which contains a unique time value, to each server side object. When client load an object, the

framework changes the timestamp attribute with a unique value. When a client attempts to write changes to the object, the framework matches the timestamp attribute. If it matched, commit change.

Obviously, the second solution is better than the first one, because two clients can load the same object at the same time in the second solution.

6. Conclusion and Future Work

6.1 Conclusion

Different ways (serialization, object oriented database and relational database) can be used for persisting objects in object oriented programming. Although relational modeling and object modeling are different paradigms of programming, we can develop a object relational mapping framework to connect them together.

In this thesis, I introduce a framework to achieve Object-Relational Mapping that is composed of two layers: Object Layer and Storage Layer.

In the discussion of design, I have contributed some new techniques: mapping inheritance with inheritance, a solution for aggregation and associations mapping, a cache of object references for constructing objects and name convention for preserving object maps. These new ideas have been analyzed and demonstrated in this thesis.

For testing my framework's performance, I used teaching assignment planner project as an example.

Another interesting part of my thesis is about server side object. The major issue in this part is concurrency. I introduced two different ways to deal with it.

6.2 Future Work

This object relational mapping framework may further be enhanced to meet the future needs by following three steps:

1. Implementing consistency control and supporting the OQL.

Some mechanisms will be needed to prevent users from interfering with one another's data. Two important concepts in maintaining data consistency are transaction and locking. It may need a transaction class and lock manager to finish. The transaction class encapsulates the transaction object provided by relational database. It lets developers utilizes database transaction in application level. Because a mapping operation may have database related operations more than once, developers need a mechanism to deal with atomicity. To control concurrency, we can use lock manager. It maintains states of all objects to deal with concurrency. We can take advantage of the cache of all constructed objects in my previous design to implement lock manager.

Supporting OQL is very useful in Ad Hoc query. The relational database is good at Ad Hoc query. We can design a parser class to handle the translation between OQL and SQL. It would provide more functionality in enterprise system design. Also the object loading design of my framework should be modified to support OQL.

2. Programming Macro for simplify implementation.

Although the actual code is simple and readable, for example, the Save and Read method, developers may find something boring to use. Macro is a right solution for this. Next is some of ideas.

```
BEGIN_MAP(class_instructor, CPersistClass)
     ADT_MAP(sCode,int,10)              //Map ADT type
     AGG_MAP(c_duties,class_duties,"Duty")    //Map Aggregation
     ASS_MAP(mListSectionTaught,class_section,"Teaching")
     //Map Associations
END_MAP()
```

If we complete this enhancement, we could start the step 3.

3. Providing a software engineering tool

In the step3 I plan to provide an application for automatically generating skeleton code based on given class structure. The ideal application could work together with other engineering software such as Rational Rose and Microsoft Visio and generate code integrated with object relational mapping.

GLOSSARY

ADTs:

Abstract data types.

Aggregation

A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part. [19]

Association

The semantic relationship between two or more classifiers that specifies connections among their instances. [19]

JDO:

Java Data Objects is a standard for Java Object persistence that's currently out in version 1.x - Version 2.0 is under development.

ObjectSet:

An ObjectSet is a collection of objects based on the relationship between within these objects.

ODMG:

Object Data Management Group.

OID:

Object Identifier

Persistence:

In Object Orientated Programming (OOP) the process of storing and retrieving objects (or more accurately their attributes) is called persistence.

Popping up:

Load the persistent object information from the storage layer.

Pushing down:

Passes persistent object information submitted by the developer to the storage layer.

Serialization:

Serialization is the process of writing or reading an object to or from a persistent storage medium such as a disk file.

SQL:

Structure Query Language.

Transaction:

Transaction is a collection of operations that form a logical unit of work.

REFERENCES

1. Srinivasan and D. T. Chang “Object persistence in object-oriented applications” IBM Systems Journal Volume 36, Number 1, 1997. Pages 66-87.
2. “Persistence: Implementing Objects over a Relational Database Version 1.0” Ratio Group Ltd, 2002 Pages 9-10. http://www.ratio.co.uk/
3. “OODBMS articles and products” Barry & Associates, Inc. 2003 http://www.service-architecture.com/oodbms/
4. Joseph W. Yoder, Ralph E. Johnson “Connecting Business Objects to Relational Databases”, Fifth Conference on Patterns Languages of Programs Monticello, Illinois, August 1998. Technical report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, September 1998.
5. Malcolm P. Atkinson, François Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier, Stanley B. Zdonik: "The Object-Oriented Database System Manifesto. In "Deductive and Object-Oriented Databases", Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD'89), Pages 223-240.
6. Wolfgang Keller, Christian Mitterbauer, Klaus Wagner. “Object-Oriented Data Integration Running Several Generations of Database Technology in Parallel”. 2001.
7. Wolfgang Keller: “Object/Relational Access Layers”, A Roadmap, Missing Links and More Patterns Wolfgang Keller, Third European Conference on Pattern Languages of Programming and Computing (EuroPLoP) Bad Irsee,

Germany 1998.

8. Mark L. Fussell "Foundations of Object Relational Mapping", Mark.Fussell@ChiMu.com, www.chimu.com ,1997-2002.

9. Klaus Renzel,Wolegang Keller: "Three Layer Architecture" in Manfred Broy, Ernst Denert, Klaus Renzel, Monika Schmidt (Eds.) Software Architectures and Design Patterns in Business Applications 1997.

10. Scott W. Ambler. Mapping Object to Relational Database. URL: http://www.AmbySoft.com/mappingObjects.pdf

11. Bertino, Elisa "Object-oriented database systems : concepts and architectures / Elisa Bertino, Lorenzo Martino" Wokingham, England ; Reading, Mass. : Addison-Wesley Pub. Co., c1993.

12. ObJectRelationalBridge (OJB) "The Apache DB Project". http://db.apache.org/ojb/

13. Hibernate "Relational Persistence For Idiomatic Java". http://www.hibernate.org/

14. Rogue Wave Software, Inc "SourcePro DB". http://www.roguewave.com

15. X-tensive.com "DataObjects.NET". http://www.x-tensive.com/Products/

16. Wolfgang Keller "Mapping Objects to Tables -- A Pattern Language", Proc. Of European Conference on Pattern Languages of Programming Conference (EuroPLOP)"97, Bushman, F. and Riehle, D.; (eds), Irsee, Germany, 1997.

17. ChiMu Corporation "Objects integrated into the Relational Model", http://www.chimu.com/,1997-2002

18. D. Edelson and I. Pohl. "A Copying Collector for C++", Usenix C++

Conference, 1991, pages 85-102.

19. OMG. “Unified Modeling Language Specification”, March 2003 Version 1.5 formal/03-03-0 Glossary.

APPENDIX A

Teaching Assignment Planner

Requirements

Peter Grogono

August 2001

1 Introduction

Each year, the Department of Computer Science assigns instructors to about 200 courses. Each course requires an instructor, and the instructors must be assigned in a way that reflects various constraints. Some of the constraints are easy to understand and simple to check. For example, an instructor cannot teach two courses at the same time. Other constraints are more complicated and must be carefully prioritized. For example, a professor who does not like teaching in the evenings may want a graduate course, but the only available graduate courses are scheduled in the evening.

The proposed program would not perform the task of teaching assignment but, instead, would simplify the task of a person, or people, doing the assignment.

2 Entities and Relationships

An assignment is a link connecting an instructor to a particular section of a particular course. In this section, we describe the data that is associated with each of these entities.

2.1 Course

A course is described by the following data (the third column gives typical examples):

Code	A four-letter string.	COMP, ENCS.
Number	A three- or four-digit string.	248, 5421.
Title	The name of the course.	Arti_cial Intelligence.
Credits	A number in the range 0 to 5.	1.5, 3.0, 4.75.
Sections	A list of `sections' (see below).	2/AA, 2/P, 4/XX.
Needs coordinator.	A boolean value true,	false.

In practice, the Sections list of a course might be implemented with pointers rather than with codes such as 2/AA.

2.2 Section

Each course is offered in zero or more sections. Introductory courses usually have several sections each term. Advanced and graduate courses typically have only one section. Course that are offered only occasionally may have no sections at all in a particular year.

A **section** is described by the following data:

Session	A single digit indicating the term .	1, 2, 3, 4.
Code	A one- or two-letter string.	A, XX.
Days	One or two days on which the course is taught.	T, WF.
Start	The start time of the lecture.	10:15.
Finish	The finishing time of the lecture.	11:30.

The session codes are interpreted as follows: 1 is a Summer course; 2 is a Fall course; 3 is a Fall and Winter course; and 4 is a Winter course. Note that a section is always associated with a course. Here is an example:

COMP 472 Artificial Intelligence (4 credits)

/2 X MW 11:45 { 13:00

/4 YY W 17:45 { 20:15

2.3 Instructor

An instructor has some fixed characteristics and some data that changes as the user makes assignments. Instructors both teach and coordinate courses; this is explained further below. The fixed (or given) characteristics are:

Name	A string of characters.
Number of courses	The number of courses that the instructor should teach.
Assigned Workload	The number of \points" for the instructor's workload.
Experience	A list of the courses that the instructor might teach.

The variable component of the data is described as follows:

Teaching A list of sections.

Coordinating A list of course/session pairs.

Actual Workload The number of equivalent credits that the instructor is performing.

2.4 Relationships

The essential relationship for this application links an instructor to a section and is called a teaching assignment. For example, the link \Peter Grogono teaches SOEN 341/4 Section S" assigns a course to me.

There is also a relationship between an instructor and a course; this is the coordination assignment and indicates that an instructor is the coordinator of a course.

There are various constraints on the assignment relations:

_ An instructor cannot be in two places at once. In fact, it is preferable to ensure that an instructor has at least one hour between classes.

_ A section can be taught by at most one instructor.

_ A course for which Needs coordination is false cannot have a coordinator.

_ A course for which Needs coordination is true may have 0, 1, or 2 coordinators. If there are two coordinators, they must be different people.

_ The courses that an instructor teaches are normally chosen from the Experience list of the instructor. However, this constraint can be overridden.

_ The instructor's assigned workload is calculated by adding 3 points for each teaching assignment and 1 point for each coordination assignment. A typical workload is 14 points and workloads do not normally exceed 16 points. The calculated workload should be approximately equal to the Assigned Workload for the instructor.

3 Use Cases

This section provides an informal set of requirements for the program by describing typical user behaviour in terms of use cases.

1. Load database. The program loads either the course database or instructor

database.

It may be possible to obtain a course database from the Department's _les. However, this database will probably be out of date and will require modification. For example, we hope to start assigning instructors for the academic year 2002{3 in November 2001 but, at this time, the Department's data will be for the academic year 2001. We will probably set up an instructor database and modify it slightly from year to year.

2. Modify database. The user can make modifications to the course database or the instructor database.

3. Load state. The current state of the program, including instructors, courses, sections, and assignments, can be restored from a _le. As usual, this can be done by selecting File/Open.

4. Save state. The current state of the program, as described in the previous use case, is saved to a file. As usual, there should be two options: File/Save and File/Save as....

5. View instructors. The program displays all instructors, together with their fixed and variable data. This enables the user to check progress and to see which instructors are still short of work. The display should include a computed column showing

Actual Workload Of Assigned Workload

A negative value indicates that more work should be assigned to the instructor and a positive value indicates that the instructor already has too much work.

6. View sections. The program displays all sections. Each section is listed with the instructor's name if an instructor has been assigned; this field is blank if no instructor has been assigned.

7. View unassigned sections. The program displays all sections for which no instructor has been assigned.

8. Suggest instructor. Assume that the program is displaying unassigned sections, as in the previous use case. If the user selects a section, the program should suggest suitable instructors (that is, the instructors whose Experience field includes this course).

The display of suggested instructors should distinguish between instructors who already have a complete workload (e.g., by displaying grey text) and instructors who need more work (e.g., by displaying black text).

Teaching Assignment Planner Requirements 4

9. Add teaching assignment. The user selects an instructor and a section; the program assigns the section to the instructor, checks for conflicts, and recalculates the instructor's workload.

Assume that the program is displaying a section and a list of suggested instructors, as in the previous use case. Clicking on an instructor assigns the instructor to that section.

10. Delete teaching assignment.

(a) The user selects a section to which an instructor has been assigned and deletes the instructor.

(b) The user selects an instructor and deletes a section from his or her teaching load.

In either case, the program recalculates the instructor's workload.

11. Add coordination assignment. The user selects an instructor and a course; the program assigns the course to the instructor, checks for conflicts, and recalculates the instructor's workload.

12. Delete coordination assignment.

(a) The user selects a course to which an instructor has been assigned as coordinator and deletes the instructor.

(b) The user selects an instructor and deletes a course from his coordination load.

In either case, the program recalculates the instructor's workload.

13. Report instructors. The program generates a report of instructors and their workloads, sorted alphabetically by instructors' names.

14. Report assignments. The program generates a report of courses and sections. The report includes the coordinator assigned to each course and the instructor assigned to each section, with blanks if no assignment has been made. Several options are provided:

(a) Sequence:

i. Sort by session, course, and section.

ii. Sort by course, term, and section.

(b) Selection:

i. List assigned sections only.

ii. List unassigned sections only.

iii. List courses and coordinators only.

15. Summaries. The program provides summary data on request. It would be even better to display the summary data on the screen at all times.

(a) Number of instructors.

(b) Number of courses.

(c) Number of courses which require coordination.

(d) Number of courses with a coordinator assigned.

(e) Number of sections.

(f) Number of sections with an instructor assigned.

Teaching Assignment Planner Requirements 5

4 Normal Use

The program should be designed so that the following operations (which will probably be typical of normal use) are easy and intuitive. Bracketed numbers refer to use cases.

1. [3] Start the program and load a complete \state" (instructors, courses, sections, and assignments).

2. [7] View unassigned sections.

3. Repeat:

(a) [8] Look at the suggested instructors for a particular section.

(b) [9] Assign an instructor to the section, noting conflicts.

(c) [10] (Occasionally) delete assignments.

4. [4] Save the current state and exit.