

A Pattern Based Persistence Framework for Object Oriented Database Systems

Uzair Ahmad
mcs121@nu.edu.pk

Muhammad Ahmad Ghazali
ahmad.ghazali@nu.edu.pk

National University of Computer and Emerging Sciences (NUCES)
Block B, Faisal Town
Lahore, Pakistan

Abstract

Over the years database management systems (DBMS's) have evolved from simple file storage mechanisms to the complex world of relational and object oriented DBMS's. Today, DBMS's have progressed to a stage where they are built around recurring themes and/or building blocks that perform the same set of functionality regardless of the database type (whether relational, network or object). These building blocks include components such as Query Processor, Transaction Manager, Storage and Cache manager etc. The implementation of these components, however, changes according to the type of the database and the vendor providing the database suite. Standardization efforts have long been there in this context to define a uniform set of interfaces exposed by the databases. However, little material is available that deals with the identification and standardization of design aspects of such systems. This paper is an attempt to bridge this gap by identifying recurring themes in the design of the persistence layer of object DBMS's and capturing them in the form of design patterns.

1. Introduction

Over the years database management systems (DBMS's) have evolved from simple file storage mechanisms to the complex world of relational and object oriented database management systems. Today, DBMS's have progressed to a stage where they are built around recurring themes and/or building blocks that perform the same set of functionality regardless of the database type (whether relational, network or object). These building blocks include components such as Query Processor or Manager, Transaction Manager, Meta Data Manager, Storage and Cache manager etc [RAO94]. The implementation of these components however, changes according to the type of the database and the vendor providing the database suite.

Although Database management systems have matured and are among the best understood branches of Computer Science but, so far very little material is available which deals with the identification of patterns in this field [LAN00]. This paper is an attempt to bridge this gap by identifying recurring themes in the design of the persistence layer of *Object Database Management Systems* and capturing them in the form of design patterns.

ODBMS is a huge field and covering all aspects of this domain in a single attempt is virtually impossible. Thus we have restricted the scope of this paper to only the persistence layer responsible for the storage and retrieval of objects. To establish an overall context, this paper will initially provide a very simple high-level architectural view of the core components of an ODBMS (i.e. Query processor, Transaction manager, Object manager etc). After providing a brief overview of the responsibilities and steps involved, section 3 will introduce a number of patterns that can be identified in the persistence layer. Finally, section 4 will present the context of these patterns in an overall framework.

2. OODB Architectural Overview

The origins of object-oriented databases date back to the late 1970's and early 1980's. Today, various commercial object-oriented databases are available in the market and have established themselves into niches such as computer-aided design and manufacturing, graphics, system services, and various scientific and medical applications [CAT94]. The strength of object data model lies in its ability to manage complex relationships among data objects that are typical to these applications.

An ODBMS supports data persistence and population, data and transactions integrity, concurrency, security, and recovery like any other database system. In addition to these basic characteristics, it also supports the object oriented concepts of encapsulation, inheritance and object identity. This relationship can be illustrated as:

Object Orientation = Abstract Data Types + Inheritance + Object Identity
Object Oriented Database = Object Orientation + Database Capabilities

Typically an ODBMS consists of 6 major components namely the *Query Processor (QP)*, *Transaction Manager (TM)*, *Metadata Manager (MM)*, *Object Manager (OM)*, *Cache Manager (CM)* and *File Manager (FM)* as shown in Figure 1.

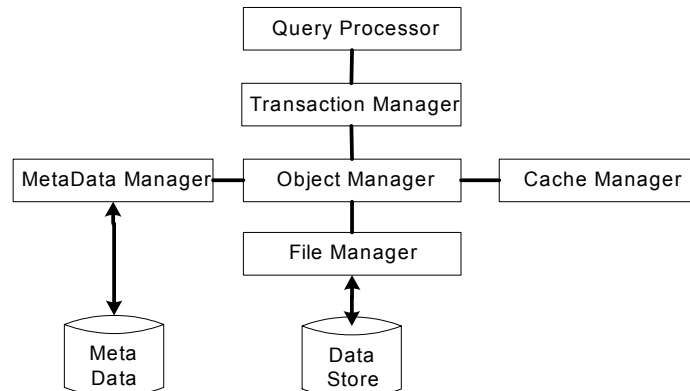


Figure 1: Components of an ODBMS

The names and responsibilities of these components may vary according to the vendor implementation but, in general, they provide the following services:

The **Query Processor** is used to translate queries from high level database languages into expressions that can be used at the physical level of the file system. This process can be divided into three steps: query parsing, semantic optimization, and the generation of query evaluation plan. The details of these steps can be found in [SIL02].

The **Transaction Manager** is responsible for managing all the access to the data. After the query is processed, the parameters are passed to the transaction manager, which then carries out the manipulation on the data already available in the object cache. It also records any changes made by the transaction, in a transaction log file and is responsible for data recovery in the event of an abnormal termination.

Metadata Manager, as the name implies, is responsible for maintaining the user access privileges, other control information and metadata of each class in the database such as its type, attribute details and size etc. All other components interact with it whenever they need some information about a particular class.

The **Object Manager** is responsible for the persistence, or the storage and retrieval of objects. It takes a reference to an object to be persisted, and converts it into a format suitable for storage in the underlying database storage structures. After the conversion, the object is handed over to the file manager for storage in the database (data store). Similarly when an object is to be restored in the memory, the OM retrieves it from the disk (through the FM) and places it in the object cache.

File Manager is responsible for storing and tracking objects on the disk/file. Thus, the object manager has the object view (i.e. it deals with objects), where as the file manager has the file level view of the object i.e. it simply stores and retrieves the object format (such as a serial object) handed it down by the object manager.

The **Cache Manager** is responsible for managing the object cache where all the persistent objects reside, after the object manager regenerates them. The transaction manager interacts with the cache manager for the objects/data needed by the query.

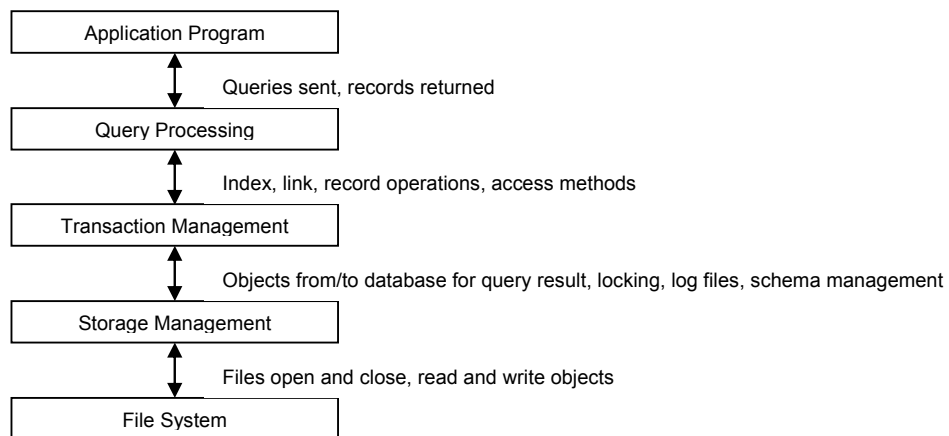


Figure 2: Operations performed against a query request [CAT94]

Figure 2 shows in general, the operations that occur when an application program requests some data from the database system. The request or query is initially parsed by the Query Processor to determine the sequence of events. The Transaction Manager then takes over and interacts with the Storage Management subsystem to ensure that the required data or objects are available in the memory. If some data is missing, object fault occurs and the Storage Manager is asked to fetch the data from disk [LOO95]. Storage Manager includes components as Object Manager, File Manager, and Cache Manager etc.

2.1 Object Persistence, Population Steps

The objective of this paper is to present a framework design for the persistence or storage layer of an ODBMS. Before describing the patterns that can be identified in this domain, we need to first understand the steps involved in making an object persistent. A brief description of the events that occur during this process is as follows:

1. Essentially, to persist an object to the database we need a mechanism to map all of its data and relationship information into the database. For this purpose the object is first converted into a format suitable for disk storage, along with all its relationships (Association, Aggregation, and Inheritance). This task of converting an object to a disk storable format is accomplished by the Object Manager component.
2. An object has a globally unique Object ID (OID) assigned to it that is necessary to uniquely identify and locate the object. This object ID needs to be stored and managed by the database along with the object.
3. After an object is converted to a storable format (e.g. as a buffer) with all its attributes and relationships, the next task is to store this buffer in the database or the underlying filing infrastructure. The File Manager is responsible for this activity. Also the OID needs to be stored somewhere in the database along with this buffer to track which object is stored where.
4. Once an object has been stored in the database it will be accessed in the future and need to be retrieved in the memory. This task of populating an object from the database is accomplished by the object manager in association with the cache and file manager. The cache manager then keeps track of the objects currently residing in the memory.

The next section presents some design patterns that we identified in the Persistence layer of an ODBMS. Section 4 then discusses how these patterns may be used in an overall framework to achieve the storage and retrieval of objects.

3. A Patterns Based Approach

This section presents some design patterns involved in the persistence and population of objects to/from a datastore. Some of these patterns are actually a realization of the patterns presented by GOF, but are generic enough to be applied in any object database thus complying with the definition of a pattern. Although different forms have been used for documenting design patterns [RIS98] we simply capture the essential elements listed in the first chapter of [GOF95]. These include the patterns name, the problem description, the solution and its structure. The implementation of these patterns has been provided in Section 4 in the context of an overall framework.

3.1 Packing an Object's State

Context

An object instantiated in the memory needs to be stored or persisted on the disk by the ODBMS layer and vice versa.

Problem

Programming languages instantiate an object's instance in the memory and it needs to be stored on the disk for making it persistent. For this purpose, an objects state needs to be captured and externalized in a format suitable for persistent storage, so that it can be restored in memory when needed. The following forces influence this solution:

- The user/programmer needs to be shielded from the underlying dynamics of this storage and retrieval mechanism. Using relational databases with object oriented programming languages necessitate that the programmers define an explicit mapping of the object to the underlying relational structures. Object databases strive to avoid this unnecessary effort and provide a seamless integration between the language and the underlying database for the storage of objects.
- An object's persistent state should only contain the business data that is crucial for storage. Usually the compiler or language runtime engine (e.g. in Java) inserts some hidden pointers and/or fields in the instance

of the object; such as the pointer for virtual table. When the object moves to the persistent storage (i.e. the disk) these pointers become meaningless and thus are not a part of the objects state.

- The externalized object state must be in a standard format/form which is suitable for storage in the underlying database infrastructure.

Solution

An object can be made responsible for externalizing its state (consisting of only business data) in a standard format, recognizable by the file manager, which can then be saved in the database. When retrieving an object from the database the same structure can be recovered and used for re-instantiating an instance in the memory, effectively reflecting the object’s persistent state.

Structure

The procedures for saving and retrieving the state information of the object can be defined in the class itself as shown in Figure 3:

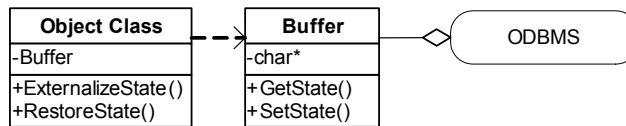


Figure 3: Pattern Structure

The object sets its state in a buffer which is then stored by the database. The buffer is retrieved and re-instantiated in memory when the persistent object is accessed.

Note that the programmer does not have to define the *ExternalizeState()* and *RestoreState()* methods against each user defined class, since these can be automatically generated and inserted in the code by a postprocessor. A better approach would be to use this pattern in conjunction with object factories (see [LAR02] for details).

Known Uses

Object databases need to provide a seamless integration with the existing programming languages. Various approaches are provided by programming languages to achieve this end. Most persistent programming languages today allow an object to be persisted without requiring the programmer to write code for interaction with the underlying data store. One way to implement such a mechanism is by using a post-processor to generate and embed the code in each class for the persistence and population of objects. The specifications defined by Object Database Management Group (ODMG) to extend and support the persistence capabilities in languages like C++, Smalltalk etc proposed this mechanism. Modern languages such as Java provide more than one persistence mechanisms like, serialization, JDBC and JDO etc. Although serialization approach necessitates that the programmer inserts appropriate declarations in the code but post-processing is a common mechanism to achieve this end in other techniques.

3.2 Object Graph Traversal

Intent

To traverse object tree/graph in a uniform way, by treating individual objects and compositions of objects identically.

Motivation

The terms *Composition* and/or *Aggregation* are used to define a mechanism for forming a whole from component parts. In general, user defined Objects are formed by the composition or aggregation of Attributes. These attributes can be in different forms and may be classified as [CAT94]:

- Simple: include primitive objects or data types such as integers, floats etc.
- Composite or Complex: include collections and composed objects.

So, objects are aggregates of primitive objects and complex objects (as shown in Figures 4 and 5). Complex objects in turn are formed by applying object constructors to simple objects (integers, floats) that are provided by all systems. Thus the entire object space can be viewed as built on top of atomic objects, of the types integer, float, boolean and string [RAO94]. This forms a recursive tree/graph structure that represents a part-whole

hierarchy. This uniform hierarchy or pattern lets the client treat primitive and complex types in a uniform way when accessing the data in an object.

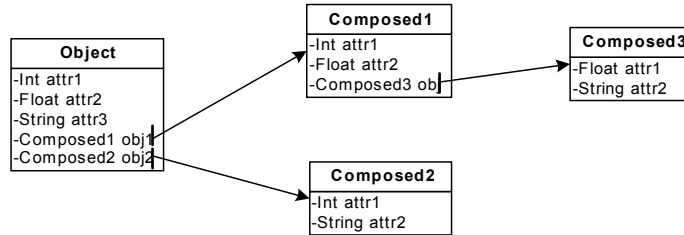


Figure 4: Aggregation of attributes in an object

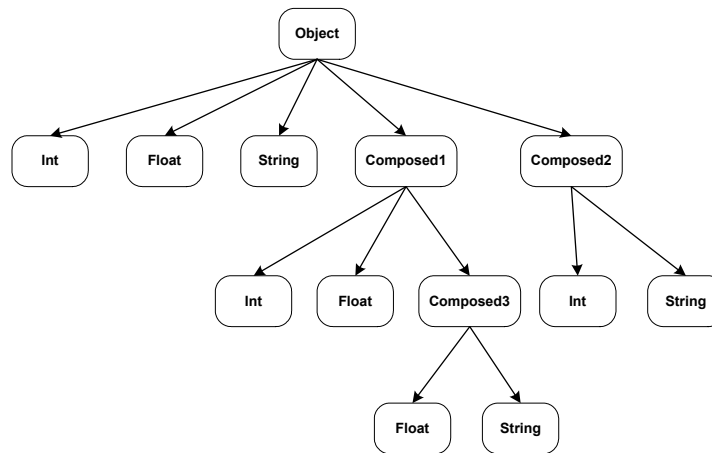


Figure 5: Part-Whole hierarchy of recursively composed objects

Usage

This hierarchical feature of objects is especially useful when making an object persistent. When an object is to be persisted it has to be converted from its in-memory representation to a format that is compatible with the underlying database storage structure (e.g. a buffer [STE03], a table etc). This conversion essentially flattens up the object tree/graph structure; say as a series of bytes that represents that object and all objects reachable from it, as done in serialization. Thus it removes the distinction between simple and complex objects and enables the underlying layers to treat the objects in a uniform way without worrying about their structure.

Structure

To enable this mechanism, define an abstract class that represents both primitives and their aggregates. The *Save()/Restore()* methods enable persistence and retrieval of an object (as shown in Figure 6).

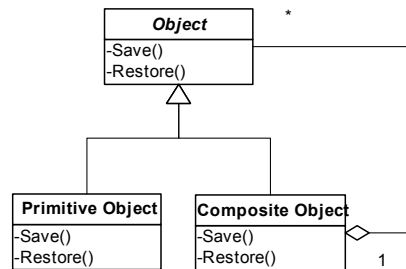


Figure 6: Pattern Structure

3.3 The Storage Manager Subsystem

Intent

Provide a unified interface to a set of interfaces in the Persistence component/layer of the database and make the underlying subsystem easier to use.

Motivation

The components of an OODB that construct the object persistence layer generally include the Object Manager (OM), Cache Manager (CM) and the File Manager (FM). The names and implementation details of these components may change with the vendor, but generally an essential set of activities is performed among them. OM is primarily responsible for converting the objects in a storable format and then saving it on the disk through the FM component. When saving an object for the first time an OID has to be assigned to the object and stored in the Global Table (GT) of the database. Similarly, when retrieving an object the OM requests the FM to retrieve the saved object and instantiates it in the memory through the CM. So, all these components work closely to implement the mechanism of storage and retrieval of objects from the memory. Also, the Transaction Manager (TM) component needs to interact with the CM component during the execution of a transaction.

So, while the underlying persistence components might be accessed by some specialized components/clients (such as TP) but mostly clients don't care about the details such as how the object is converted into a storable format, how the OID is generated and attached with the object and how the object is retrieved. Clients merely want to save and retrieve an object. For them the powerful but low level interfaces in the persistence layer only complicate the task.

Usage

To provide a higher-level interface that can shield clients from these different components, the persistence layer also includes a Storage Manger (SM) interface. This interface defines a unified view to the persistence layer functionality. This interface acts as a façade [GOF95]. It offers clients a single, simple interface to the persistence subsystem. It glues together the classes that implement the persistence functionality without hiding them completely. The SM facade makes life easier for most users without hiding the lower-level functionality from the few that need it.

Structure

The structure of the pattern is shown in Figure 7. Here Storage Manager acts as a façade, providing a uniform view to the underlying functionality. SM does not restrict other components like TM to interact directly with the cache manager when required.

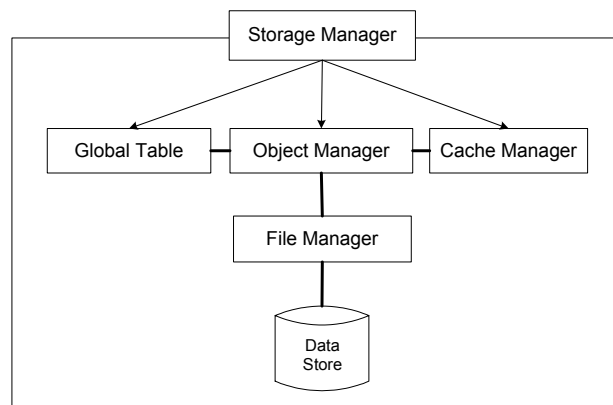


Figure 7: Pattern Structure

3.4 Global Object Table

Intent

To uniquely identify, and locate an object globally in the database.

Motivation

Both Object DBMS and object programming languages deal with object identifiers. Object programming languages use the physical location or address of the object as its identifier. This approach works in general but,

on its own, is insufficient to provide a generic mechanism suitable in all situations for referencing objects. For example, ODBMS's may support relocation and/or clustering of objects. In such cases it is possible that an object might be moved from one storage place in the database to a different place. In consequence, this means that the object identifier changes its value, as it now refers to a different storage place in the database. Similarly in a distributed ODBMS such a mechanism would be difficult to implement.

Also, some ODBMS's provide database roots to locate application objects, starting with a named application object [RAO94]. After accessing the named root object, applications can navigate to other objects associated with that root. These features again do not scale well, if all or a set of related objects in the database need to be accessed.

In all these situations it becomes necessary to be able to look up an object not only by the object identifier of the database, but to have a constant identifier that does not change, even when the object changes its location or resides in a distributed environment.

Usage

To provide a global view of all the objects in the database, implement a scalable map or table for mapping unique identifiers to database locations. The map will contain an entry for every persistent object stored in the database. It can also store additional information such as the visibility of the object (as in the case of composed objects), the type of the object, its location etc (as shown in Figure 8).

OID	Type	Location	Visibility
123456	type1	0x934F	hidden
987654	type2	0x808F	visible
...

Figure 8: Possible instance of a global table

Structure

The Global Table stores the ID's of all the persistent objects in the database as shown in Figure 9.

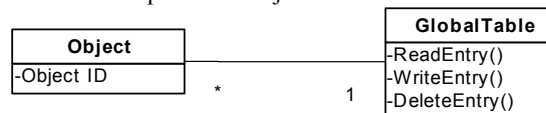


Figure 9: Pattern Structure

3.5 Lazy Swizzling using Proxies

Intent

Provide a surrogate or placeholder for a referenced object till it is actually accessed.

Motivation

An ODBMS implements a 'reference' by mapping an object identifier, which is its database address, to a physical address. Different ODBMS use different techniques for this mapping. In any case, when one object references another, the object DBMS must locate the second object and fetch it into the cache, where it can be accessed directly by the object programming language runtime.

Usage

An object references other objects. Consequently, when it is fetched in the memory its associated objects must also be brought in the memory at some point. It is sometimes desirable to defer the fetching of associated objects until absolutely required. This deferred fetching of associated objects is known as *Lazy Swizzling* [LOO95]. Lazy swizzling may be implemented by making object references as virtual proxies [GOF95]. A virtual proxy is a proxy for another object that fetches the real subject when it is first referenced. It is a lightweight object that stands for a real object that may or may not be fetched (as shown in Figure 10).

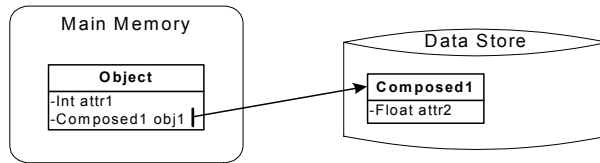


Figure 10: ‘Obj1’ acts as a reference to the instance of ‘Composed1’ in the data store

Structure

The object reference is a proxy that acts as a temporary placeholder for the Referenced Object till the object is actually accessed. The structure of this pattern is represented in Figure 11.

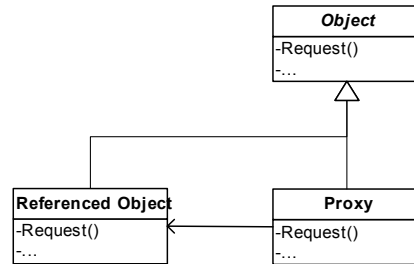


Figure 11: Pattern structure

3.6 Other Patterns

Other design patterns identifiable in the persistence layer include:

1. All major components in the systems such as the Storage Manger, Cache Manager, OID Generator etc are Singletons.
2. Also, the “Packing an Object’s State” pattern discussed in section 3.1 can be used in conjunction with object factories for the storage and retrieval (materialization and dematerialization) of objects. Larman [LAR02] presents a thorough discussion on the materialization of objects using **object factories** and **template method**.
3. The Global Table sometimes also needs to store the number of references pointing to a particular object. The object cannot be deleted unless the reference count becomes zero (a similar concept is used in Java Garbage Collection). This is an example of smart reference or proxy pattern.

4. A Pattern Based Persistence Framework

This section presents a framework for the persistence layer and shows how the patterns documented above, fit in this framework. The class diagram of the framework is shown in Figure 12 and discussed below.

The persistence framework reflects only the behavior required for the persistence and population of objects, as described in section 2.1. When an object is to be persisted, it is first converted in a disk storable format i.e. a buffer in our case. All the primitive attributes and composed objects need to be packed in a buffer for storage on the disk. For this purpose we need to traverse the object graph/tree recursively, buffering and saving each composite/contained object first, and then placing its reference or OID in the containing object buffer. The “Object Graph Traversal” and “Packing an Objects State” patterns work in unison to achieve this objective. When the *Save()* method is called on an *Object* the object stores all its primitive attributes in a buffer and calls the *Save()* method recursively on its composed objects. The resulting effect is that the composed objects are persisted first; their reference is placed in the containing object buffer and this buffer is then stored on the disk.

All the access to the persistence layer or subsystem is controlled by the “Storage Manger Sub-system”. When an object is to be saved, the *SaveObject()* method of the *Storage Manager* is invoked which in turn requests the *Object Manager* to convert the object in a byte stream or buffer. OM invokes the *ExternalizeState()* function of the object which places all the primitive attributes in a buffer and recursively calls the *Save()* method on its composed objects. Finally this buffer is passed to the File Manger for storage on the disk file.

For keeping track of the persistent objects “Object Lookup Map” or *Global Table* stores the *Object ID*’s for all the objects. When an object is to be retrieved from the datastore, its *OID* is searched in the global table to get the exact location where it is saved. The OM is then requested to retrieve the object buffer from the disk and restore it in the object cache. The *Cache Manager* keeps track of all the objects currently present in the cache.

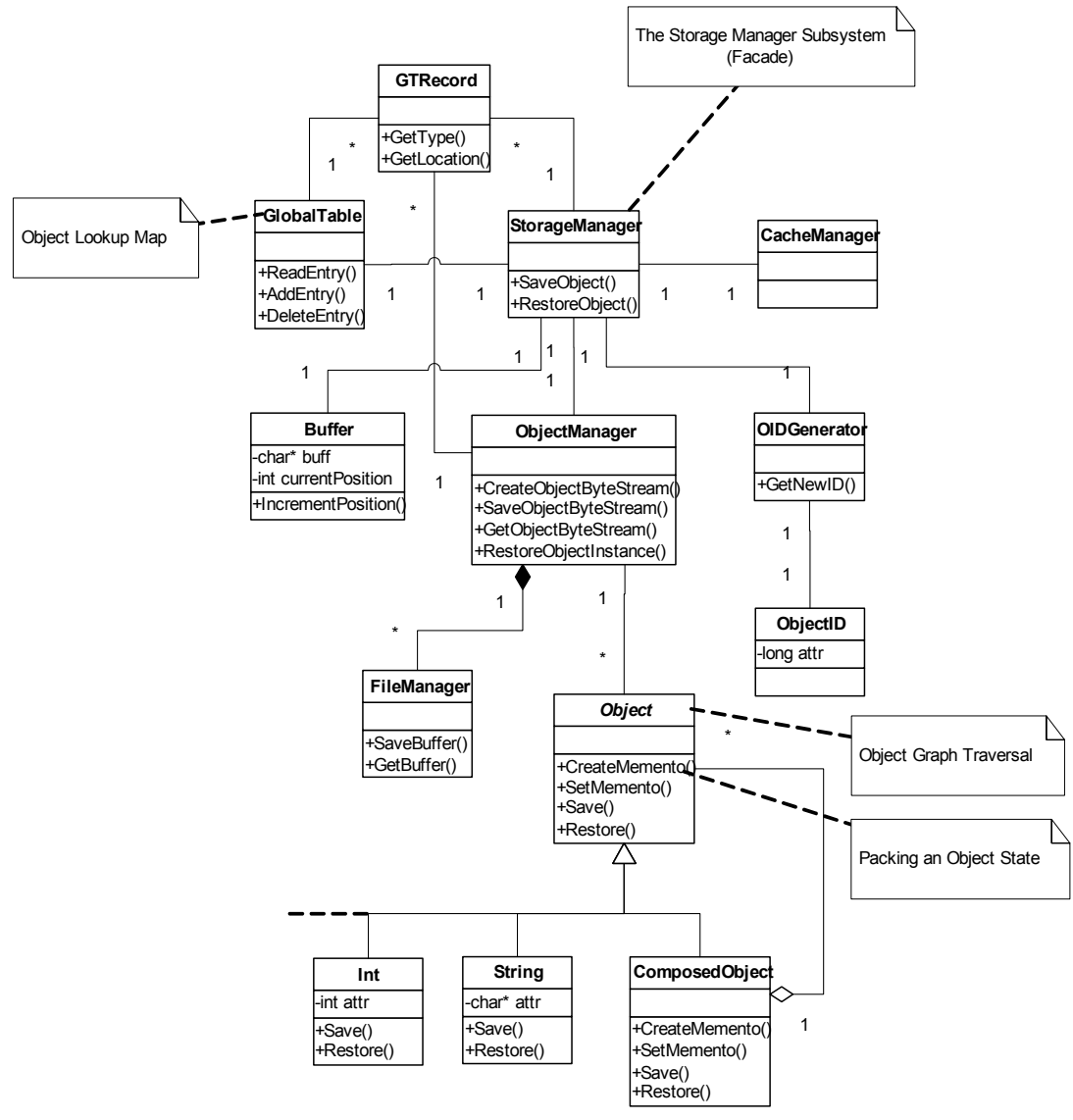


Figure 12: Class Diagram of the Persistence Framework

5. Conclusion

This paper presented a patterns based persistence framework for an ODBMS. Object databases are a huge domain and this paper only touches the surface. The main objective of this study is to initiate a thinking process towards identifying recurring design solutions in this domain. Also it is important to realize that there may be many alternative design patterns that can solve the same recurring problem. The challenge lies in selecting the right pattern according to the context. The important thing is the designers experience; the more experienced you are the more flexible design you will create.

6. References

- [CAT94] R.G.G. Cattell, "*Object Data Management, object oriented and extended relational database systems*" (Revised Edition), Addison-Wesley Publishing 1994.
- [GOF95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.
- [LAN00] Manfred Lange, "*A Collection of Patterns for Object-Oriented Databases*", Submission for preliminary conference proceedings of EuroPLOP, 2000.
- [LAR02] Craig Larman, "*Applying UML and Patterns*", 2nd Edition, Prentice Hall Publishers, 2002.
- [LOO95] Mary E.S. Loomis, "*Object Databases, the Essentials*", Addison-Wesley Publishing 1995.
- [RAO94] Bindu R. Rao, "*Object Oriented Databases, Technology, Applications and Products*", McGraw-Hill Publishing 1994.
- [RIS98] Linda Rising, "*Pattern Writing*", in Rising, L. (Ed.), *The Patterns Handbook: Techniques, Strategies, and Applications*, Cambridge University Press, New York, January 1998, pp. 69-81.
- [SIL02] Abraham Silberschatz, Henry F. Korth, S. Sudarshan, "*Database System Concepts*" (4th Edition), McGraw-Hill Publishing 2002.
- [STE03] Dr. Jeffrey S. Steinman, Jennifer W. Wong, "*The SPEEDS persistence Framework and the Standard Simulation Architecture*", Proceeding of the 17th workshop on Parallel and Distributed simulation (PADS'03), 2003.