In-Process Object-Oriented Database Design for .NET

Yanhao Zhu
Department of Computer Science
East Carolina University
Greenville, NC 27858
USA
01-252-3281327
ZHUYA@MAIL.ECU.EDU

James Crouch
Department of Computer Science
East Carolina University
Greenville, NC 27858
USA
01-252-3289691
JLC0627@MAIL.ECU.EDU

Mohammad H. N. Tabrizi
Department of Computer Science
East Carolina University
Greenville, NC 27858
USA
01-252-3289626
TABRIZIM@MAIL.ECU.EDU

ABSTRACT

In this paper, we introduce the development of an In-Process Object-Oriented Database (OODB) design for the .NET platform. Using an OODB design, one simple function call is needed to save, search, delete, or update .NET objects. Also little database setup is required, as opposed to defining the system's schema in relational database systems. In order to validate the efficiency of In-Process OODB design, an experiment was conducted involving a relational database system. The results show that the In-Process Object-Oriented Database design outperforms Microsoft SQL Server (running locally) for the queries based on the primary key fields.

Categories and Subject Descriptors

H.2.4 [**Database Management]:** Systems - *Object-oriented databases.*

General Terms

Algorithms, Performance, Design, Languages

Keywords

Object-oriented, database, design

1. INTRODUCTION

The rise of object-orientation including object-oriented analysis [1], [2], [3] and object-oriented design [4], [5] defines the history of computing. The earliest work in computing concerned itself exclusively with what we now think of as programming. Only later did a conscious concern with design and analysis as separate issues arise. Object-oriented programming first responded to this need, then object-orientation including object-oriented design, and, more recently, object-oriented analysis completed the design of the new system. As a result, object-orientation is now a firmly established and essential part of the software development culture. With increased emphasis on distributed systems, the object metaphor appears to be the most natural one to adopt, given its emphasis on encapsulation and message passing. In addition, increasing concern over maintenance costs may well lead to the recognition that reusability is a key issue in programming, design, and analysis.

Preventing the existing methodologies and technologies from becoming archaic, one has to look deeper into the application of the existing technologies. Let us just talk about the database technology itself. It seems that most existing technologies are using relational databases design. Although almost all applications are being developed using object-orientation programming, not much attention is being paid to the object-oriented analysis and design. Without foresight, many existing software created specifically for today's needs. The application of object-oriented database design, which continues from the need for persistent objects in object-oriented programming languages that has lead to the development of OODB. One of the major characteristics of OODB is its tight relationship with object-oriented programming languages. With the increasing popularity of object-oriented analysis and object-oriented design, more and more research realized that a relational database is not a good match for an object-oriented world. Object-Oriented Database Management Systems (OODBMS) have become an intensive research topic [6] followed by production of the prototypes and commercial products [7].

Compared with the relational database systems, the OODBMS has the following common properties:

- Transparent persistence: In OODBMS, users can access the persistence objects stored on disk in much the same way as they access the transient objects in the application memory. In contrast to the relational database, the smallest unit in OODBMS is an object, not rows or columns.
- No impedance mismatch: For the OODBMS users, there is no need to write code to map tables into objects or the other way around. As a result, the code of the application using OODBMS is quantitatively much less than that using the relational database.
- One data model: Most of the time, OODBMS is used with object-oriented programming language. So users can focus on their more familiar object model without any distraction. This can result in better application design/architecture.

- Better performance: Most of the OODBMS save the programming language object as is on disk. But the relational database system separates objects into different tables and connects the tables by various foreign keys. Packing complex objects into tables and unpacking objects from tables usually takes a lot of time. By just saving composite objects and relationships directly, OODBMS often perform better than relational database systems.

2. PROJECT DESCRIPTION

The aim of this project is to promote OODBMS that provides a simple mechanism for users to save the .NET objects directly to disk. This research project is designed in such a way that only one simple function call is needed to save an object. Almost all the existing OODB systems require users to derive from some special class to define the classes which will be saved into the database. Some others require users to implement a specific *interface* for the classes. There are no such requirements in the in-process OODB developed here. The .NET reflection (will be discussed later), which is a powerful technique used to manipulate types' metadata at runtime, is used to understand the various properties of a class and an object. To save some development time, the class to be saved to the database is required to have a "primary key field" so that different objects of that class will have different key field values. Users should specify the primary key field by providing attributes to the class during the class definition.

2.1 Saving information of an object

Although a class can have constructors, destructors, constants, methods, properties, indexes, fields, and other members in .NET; only values of the fields may be different for different objects of the same class. All other members are class-level specific and are therefore exactly the same for all the objects of that class. This simple fact suggests that to save an object, we only need to save all the fields' values of that object. Other information will be handled at class level.

2.2 Saving the information of a class

Only a very small portion of the class information needs to be stored. In the implementation of this project, only the class name and names of all fields are saved in the database. This is based on two facts.

- In the user program, the class has to be defined somewhere before it can be used. This means that the types of objects which the user wants to deal with have to be declared first in the program. In C#, VB.NET, VC++.NET and other strong typed .NET programming languages; the compilers will enforce this requirement without any exception. The classes of the stored objects, which will be used in the program, have to be defined.
- With .NET reflection, the metadata of any defined type/class in the user program is accessible at runtime.

2.3 Data structure for saving the information

B-tree was chosen to store the objects as well as the class information. B-trees are not only highly efficient, but also relatively easy to implement. Today it is one of the most frequently used database techniques for indexing. When the size of stored records is small, B-tree is often used for primary file organization [8]. In this project, B-tree was used as the average size of stored objects is not believed to be very high. Each entry in the B-tree node is of the form <key, data>, and inserting/searching/deleting is based on the key value. This is one of the reasons why the persistent class is required to have a "primary key field".

2.4 Project overview

A general purpose B-tree was designed and reused to save class and objects information. There is one B-tree which is dedicated to save class information in the system. For each persistent class, the class name is used as the key and an array of strings of the fields' names is saved in the data part of the node entry. For each stored class, there is a B-tree, called *object tree* which is used to save instances. In the *object tree*, the primary key field value of the object will be used as the key part of the tree node entry, and the data part contains the non-key field values. The pointer to the top node of the *object tree* is also saved in the *class information B-tree*.

The only problem of this approach is that the size of the node entry <key, data> cannot be fixed for the class information and objects. Even if the two objects are of the same class, the size of each object can be greatly different from the other. So each node in the *class information B-tree* and *object B-tree*s is a variable length segment of the *database file*. For any entry, there are two pointers at each side. The pointers actually are the segment IDs. Given the ID of a segment, its location and length should be easily found. So another B-tree, called *segment tree* (in *index file*), with segment ID as the key, is used to manage the segments in the *database file*. There is only one *segment tree* in a database. A segment can be moved around in the file as its size changes, leaving some empty holes in the *database file*. Another B-tree, which is called *space tree* (in *index file*), is used to keep track of these holes and try to reuse them if possible. Because the size of node entry in the *segment B-tree* and the *space B-tree* can be fixed, *segment tree* and *space tree* are saved in the *index file* which is separated into fixed length pages. Space allocation/deallocation for paged file is much easier than that for segmented file. A bitmap is used to keep track of the space usage in the *index file*. More details about how to get information about class, object at runtime, and how the internal storage works, will be discussed in the following section.

3. REFLECTION

3.1 Reflection for Class Information

Reflection is a fundamental facility of .NET CLR. In the .NET base class library, a whole namespace, namely *System.Reflection,* is dedicated to the reflection. The classes in that namespace, together with class *System.Type*, allow the user to obtain information about any type/class defined in the program at runtime. With reflection, one can not only create instances of type/class at run time, but can also invoke members of any object dynamically. The .NET base library class *System.Type* is the root of the reflection functionality as it is the primary way to access the class metadata [9]. Its members can be used to get information about a type declaration, such as the constructors, methods, fields, properties etc.

3.2 Reflection for Object

There are two scenarios where reflection can be used to deal with objects. One is that when the user tries to save some object into the database, the values for all fields of that object should be collected. The other is that when the user tries to search for some objects, the matching objects have to be created dynamically. As discussed before, class *System.Type* is the root of .NET reflection. It has a method called "*InvokeMember*", which can be used to get value of any field of an object, and to create an instance of a class, and to set the value of any field of an object. In other words, this method itself is enough to deal with objects.

4. DEVELOPMENT of the INTERNAL STORAGE

Among the 36 different classes developed in this project, only six of them are not related to internal storage. In this chapter, the development of the internal storage will be discussed.

4.1 The General B-Tree

A general purpose B-tree was designed in this project. The B-tree can be used to save any kind of information as long as different records have different values of the key field. In fact, the order of the general B-tree can be any positive number. To make this possible, two *interfaces* are defined in C#, namely *IKey* and *IData*. A class can be used as the key in the B-tree if it implements the interface *IKey*. In the specification of the interface *IKey*, there is a property called Data. We can define any class to implement *IData* interface and use that class as the data field of *IKey*. Listing 4.1 shows the specifications of the *IKey* and *IData*.

```
// The keys in the B-tree have to implement this interface.
public interface IKey
{
            IData Data  {  get; set;}
            int CompareTo(IKey B); //compare this with
            another key, 0: this==B, >0: this>B, <0: this<b
            byte[] Serialize(); //serialize the key, including
             the data field,  into the byte array
            IKey Deserialize(byte[] bytes);
            IKey Deserialize(byte[] bytes, int offset, int
            count);
}

// All the user data part of the b-tree must implement this
interface.
public interface IData
{
            byte[] Serialize();
            IData Deserialize(byte[] bytes);
            IData Deserialize(byte[] bytes, int offset, int
count);
}
```

Listing 4.1 Interfaces IKey and IData

In the declaration of interface *IKey*, there is a method called "*CompareTo*", which is used to compare the keys during various B-tree operations. The "*Serialize*" and "*Deserialize*" methods are also very important. The "*Serialize*" method converts the key, including its data field, to an array of bytes. This method is used to save the key to disk during the serialization of B-tree node. The "*Deserialize*" method converts a bytes array back to a suitable *IKey* object. Thus it can be used to get the keys back from the disk during the "deserialization" of B-tree node. The B-tree node stored in disk is not exactly the same one as that in memory. The B-tree node in disk is the "serialized" version of that in memory. The B-tree node in memory is the "deserialized" version of that in disk.

A B-tree node is "serialized" when the node is evicted from the cache or during the database closing. The "serialization" of a B-tree node will convert all the information in the node into an array of bytes. For each key stored in the node, its "*serialize*" method will be called to get the byte array of the "serialized" version. A B-tree node will be "deserialized" when it is brought from the disk to the memory. During the "deserialization", each key's "*deserialize*" method will be called to get the key object back.

To speed-up the operations, the searching, insertion, deleting, and updating operations in the B-tree are implemented in a non-recursive way. To instantiate a B-tree object, a *B-tree node management system* has to be provided. It has to be a sub-class of the class *OOD.SegmentManager*. Reading a B-tree node from the disk and saving a B-tree node back to the disk are some of the typical tasks of the *node management system*.

4.2 B-tree Node Manager and Cache

To use the general B-tree, one needs to define a class to implement the *IKey* interface and another one to implement the *IData* interface. Besides that, decisions on how to manage the nodes in this specialized B-tree have to be made. The size of the nodes stored in the disk usually varies for different B-trees and different mechanisms may be needed to manage the nodes in disk. So after the key/data which will be stored in the tree is decided, a sub-class of *OOD.SegmentManager* has to be defined to implement the reading node and saving node operations for the tree. Since *OOD.SegmentManager* is an abstract class, any class inherited from it has to override all the abstract members it defines. In fact, *OOD.SegmentManager* is quite simple in the sense that it just defines several abstract methods later used by the B-trees and the system. Listing 4.2 shows the complete listing of class *OOD.SegmentManager.* In the implementation of this project, a class for the B-tree node (*BNode*) is defined as a sub-class of the *Segment* class. This is based on the possibility that other data structures may be added in the future and the existing *B-tree node manager*s can be re-used.

If B-tree requires a node, it gets it by calling the method "*GetSegment*" of *node management system* to get it. If needed, the argument "*segFactory*" is used to "deserialize" the node from the byte array. The method "*GetNewSegment*" is used to create a new node, and the method "*FreeSegment*" frees the specified node. The "*Close*" method writes out all modified segments (B-tree nodes) back to disk and is used by the system during the database closing.

```
public abstract class SegmentManager
{
        public abstract Segment GetSegment(uint segId,
        Segment segFactory, object helper);
        public abstract void GetNewSegment(Segment
        seg);
        public abstract void FreeSegment(Segment seg);
        public abstract void Close(); //write out modified
        segment back to disk
}
```

Listing 4.2 Abstract class SegmentManager

Two sub-classes of *OOD.SegmentManager* are defined in this project. For each of them, a hash table is provided to cache the nodes in memory. When the cache is full, victim nodes to be evicted will be selected by the Least-Recently-Used (LRU) page replacement algorithm. During the operations of the B-tree, the top node of the tree (the most frequently used one) is guaranteed to be in cache by the LRU scheme. In general, the nodes at lower levels have a better chance to stay in the cache than those at higher levels once the LRU algorithm is used. In the insertion operation of a B-tree, the LRU scheme implies that nodes which need to be split most likely are in the cache already. Although the .NET base class library provides a generalized hash table, a hash table with the LRU scheme built-in support is designed

4.3 Index File

The main purpose of the *index file* is to monitor the segments (only tree nodes currently) in the *database file*. Two B-trees, the *segment tree* and the *space tree*, are stored in this file to keep track of the space usage in the *database file*. For the *segment B-tree*, each node entry is of the form <segment ID, (offset, length)>, where the segment ID serves as the key and the pair (offset, length) is the data field of the key. During the various B-tree operations, the *segment tree* is consulted to provide the address and length of needed "serialized" nodes on disk. To remember a free segment in the database file, only the offset and length of that segment need to be saved. So the node entry in the *space tree* is of the form <offset, length>. Two operations are needed for this *space tree*. 1) When a segment is freed in the database file, its offset and length should be inserted into this tree. If the segment which is immediately after this segment exists in the *space tree*, these two segments will be combined together. 2) Allocating a new segment is accomplished by searching for a big enough free space. Space allocation for the *database file* is handled in the first-fit manner. Each "serialized" key in the *space tree* is of size 8 bytes, so 36 is chosen to be the order of the *space tree*.

This *index file* is separated into fixed length of pages. Currently, each page has 512 bytes. The page is the smallest unit for space allocation/dealocation in this file. Each page is identified by a page ID, which equals the offset of the starting address (in bytes) of the page divided by 512. The file header is stored in the first page, namely the page 0. The page (segment) IDs of the top node for the *segment B-tree* (in the *index file*), the free *space B-tree* (in the *index file*), and the *class information B-tree* (in the database file) are kept in the file header. Most of the 512 bytes in the file header have not been used, and are reserved for the future.

From the page 1 to the page 32, the 16 KB memory usage bitmap is stored. Each bit of this bitmap represents a page in the *index file*. The value of each bit indicates whether a page is free or not. This allows up to 16*1024*8 = 131072 pages in the *index file*. There are 131039 possible nodes for the *segment tree* and the *space tree*. If assume 2/3 of the nodes are used for *segment tree* and the tree is 69 percent full, there will be about 1588696 segments in the *database file*. For each tree in *database file*, the order is 21, that all together allow more than 25000000 objects stored in the database.

It is important to notice that it is very easy to increase the size of bitmap dynamically at runtime. For example, if a free page cannot be found in current bitmap, additional 16KB memory will be allocated for the bitmap. The IDs of all the pages used by the bitmap can be saved in the file header. To keep the size of the *index file* as small as possible, the page allocation for this file is handled in the first-fit manner. A class *memSegmentManager* was defined to inherit from the abstract class *SegmentManager*. The class *memSegmentManager* uses the bitmap to manage the pages in the *index file*. The *segment tree* and the *space tree* employ this class to request a new page or free an unused page. The "deserialized" B-tree nodes are cached in a hash table. The page replacement algorithm used is the LRU scheme. When a B-tree node is evicted from the cache, all the references to that B-tree node are released. The .NET garbage collection will be eventually invoked and clean up that unused node.

4.4 Database File

The objects are stored in this file. There are two different kinds of B-trees, the *class information B-tree* is the first, which is used to store the information of each class whose objects are stored in the database. There is only one such a kind of B-tree in the file. The second kind is the *object tree*. For each persistent class, a clustering *object tree* for that class is created in this file too.

The full name of a class is used as the key for the *class information B-tree*. For this B-tree, class *KCatalog* was defined to implement the interface *IKey*. The "*IKey.CompareTo*" method is carried out by comparing the strings in the directory order. The class *DCatalog*, which implements the interface *IData*, is defined to remember the field names of a stored class. *DCatalog* also has a pointer which points to the top node of the clustering *object tree* for the persistent class stored.

For the clustering *object trees*, class *KClass*, which implemented the *IKey* interface, was defined to hold the value of the primary key field of an object. The type of the primary field is also saved in the *KClass*, which helps to compare two values of the primary field. *DClass* is defined to implement the interface *IData*. It is used to store the values of all other non-primary key fields of an object. The .NET reflection technique is used to convert the serialized object to a real object, and vice versa.

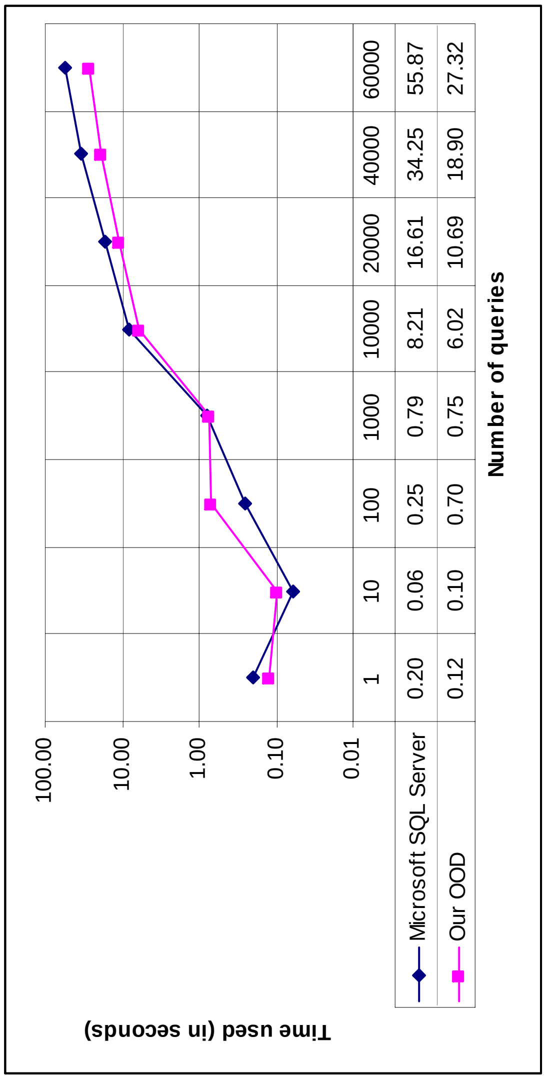
The B-tree node management in this file is accomplished with the help of the *index file*. Each B-tree node is a segment of the file. Whenever a B-tree node in memory needs to be “serialized” and saved to disk, the addressing information, the offset and the length, can be reached from the *segment tree* in the *index file*. It also makes it possible to compare the old size of the node with the size of the current “serialized” byte array. If the new size is bigger than the old one, the old segment needs to be freed and a bigger free space will be requested from the *space tree* (in the *index file*), and then the node will be saved to the new location. During the deletion of a B-tree, a used node may be freed. If this happens, its addressing information can be deleted from the *segment tree* and that segment can be inserted into the *space tree*. All the “deserialized” B-tree nodes from this file are cached in a hash table and the LRU scheme is used again here.

Using two B-trees in the *index file* to keep track of segments in the *database file* seems to involve too much overhead. But given the speed of B-tree and the cache scheme used, it performs well. The major reason for selecting this approach is to save some development time. An alternative method we tried is that the size of the B-tree node is fixed, and the key can have the variable length. Whether the node is full or not is not based on the number of keys in the node; instead the node is full whenever there is no room for a new key. The problem of this approach is that it does not scale well. If the length of the key is very large, the speed of B-tree will be decreased significantly[10].

5. PERFORMANCE VALIDATION

In order to validate the efficiency of the In-Process OOD design, an experiment was conducted. The results show that operations based on the primary key field are extremely efficient. In the experiment using a simple class of several fields and the primary key field of type integer, querying a random object by its primary field took about 0.15 seconds. To get a better idea about the speed of this In-Process OOD, a comparison with Microsoft SQL server was carried out. Since the most important operation for a database is searching, only query on the primary key field was conducted.

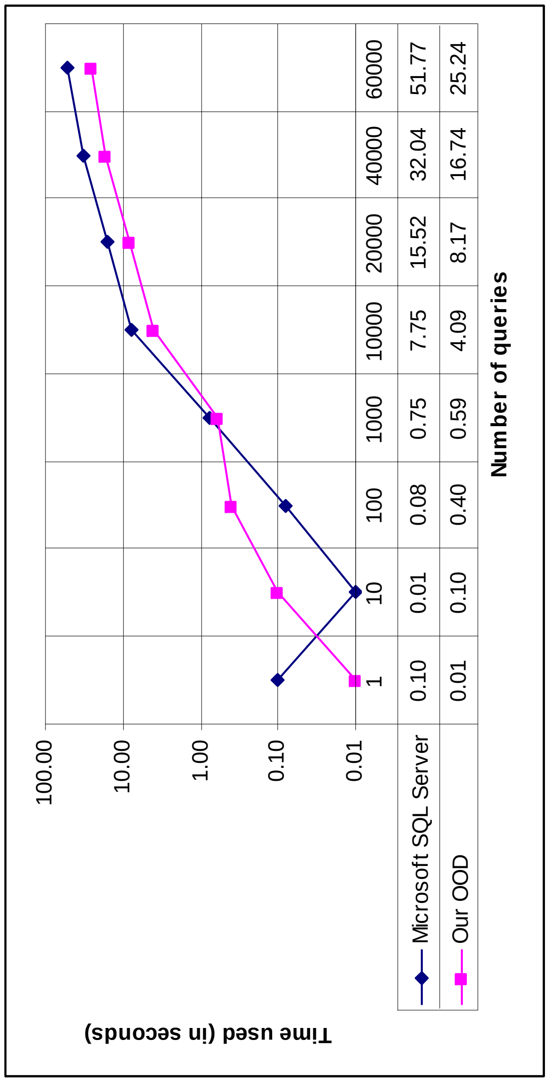
To test performance of the developed in-process OOD, a simple class with fields, *m_id*, *m_name*, *m_sex*, *m_birthDate*, and *m_age*, namely *Student*, was defined. The field *m_id* is the primary key field; different objects of *Student* class are required to have different IDs. 60,000 different *Student* objects were first inserted into the database, and then the objects were queried back by their IDs. On the SQL server side, a table was first created with five columns, which have the same names as the fields of *Student* class. Then a batch query was written to insert the 60,000 student information into the table. An equivalent program was used to query the student information back and to pack the information to *Student* objects.



Number of queries	1	10	100	1000	10000	20000	40000	60000
◆ Microsoft SQL Server	0.20	0.06	0.25	0.79	8.21	16.61	34.25	55.87
■ Our OOD	0.12	0.10	0.70	0.75	6.02	10.69	18.90	27.32

Figure 5.1. Query on random IDs

Figure 5.1 shows the time used to randomly query a certain number of *Student* objects by IDs. The database initialization time, in the case of SQL server the time used to create the connection, was not included in the experiment. Time used to search a number of objects in the ascendant order of IDs is compared in Figure 5.2. It is interesting to note that when the number of queries is very small, the performance of SQL server is about the same as the In-Process OOD. But the in-process OOD outperforms Microsoft SQL Server as the number of queries increases. One possible reason for the superior performance of the in-process OOD with regards to large number of queries is the implementation of cache scheme, where queries enable more nodes to be brought into cache, and the LRU node replacement algorithm guarantees the most needed nodes are always kept in cache. Another reason could be that the in-process OOD is running in the same process as the user program, there is no inter-process communication overhead.



	1	10	100	1000	10000	20000	40000	60000
◆ Microsoft SQL Server	0.10	0.01	0.08	0.75	7.75	15.52	32.04	51.77
■ Our OOD	0.01	0.10	0.40	0.59	4.09	8.17	16.74	25.24

Figure 5.2. Ordered query on IDs

6. CONCLUSIONS

Application of the existing technologies including the database technology is the subject of this study. Although almost all applications are being developed using object-orientation programming, not much attention is being paid to the object-oriented analysis design. Without foresight, many existing software created specifically for today's needs. Given the inefficiency of packing/unpacking objects, the object-oriented database system seems to be the natural choice. In this project, the development of a single-user in-process database for the .NET platform is described. Single-user in-process refers to the fact that this project will use an embedded database engine for standalone .NET applications. Using the in-process OOD, the user should be able to save, query, delete and update .NET objects through a simple interface.

7. REFERENCES

[1] Meyer, B. Objected-Oriented Software Construction, Prentice Hall, 1988.

[2] Shlaer, S. and Mellor, S.J. Object-Oriented Systems Analysis: Modeling the World in Data. Prentice Hall, 1988.

[3] Lee, S. and Carver, D.L. "Object-oriented analysis and specification: A knowledge base approach". J. Object-Oriented Program. Jan. 1991, pp. 35-43.

[4] Coad, P. and Yourdon, E. Object-Oriented Design. Prentice Hall, 1991.

[5] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. Objected-Oriented Modeling and Design. Prentice Hall, 1991.

[6] Elisa Bertino and Lorenzo Martino, Object-Oriented Database System—Concepts and Architectures, Addison-wesley, 1993, p 8.

[7] ObjectStore (Progress Software Corp.), Cache´ (InterSystems Corp.), FastObjects (FastObjects Inc.), db4o, etc. Please check http://www.service-architecture.com/products/object-oriented_databases.html for a more complete list.

[8] http://www.bluerwhite.org/btree/.

[9] MSDN online .NET documenting: http://msdn.microsoft.com/library/.

[10] Donald E. Knuth, The art of computer programming, Volume 3, 1973, p475.