INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

OBJECT CATEGORY DATA MODEL: A NEW DATA MODEL FOR OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS

by

Adane H. Zygtá

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfilment of the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

2000

© Copyright

2000, Adane H. Zygtá

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-52383-7

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis.

**OBJECT CATEGORY DATA MODEL: A NEW DATA MODEL FOR OBJECT-
ORIENTED DATABASE MANAGEMENT SYSTEMS**

submitted by
Adane H. Zygtu
in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

May 2000

Abstract

Object-oriented database management systems (OODBMS) attempt to integrate database and object-oriented (OO) technologies. However, a single data model has not emerged from this approach. Rather several research prototypes and commercial products have been developed using different models. These models reflect the particulars of their implementation, the programming language used and the set of applications they are targeted for. The thesis defines and illustrates a data model named Object Category (OC) that is based on the OO concepts. The data model is independent of any programming language, is suited to any general application, and is easily derived from an EER diagram. OC data model introduces the concept of category, which is a collection of object instances. The use of collections by an OODBMS implies a clear and easy way of manipulating persistent objects that is consistent with the concept of database and resembles the operations of insert, retrieve and remove objects.

Acknowledgements

First of all I would like to thank my wife, Salem Tewelde, for her support and understanding. This thesis would not have been finished otherwise.

I would also like to thank my supervisor, Prof. Ekow Otoo, for his ideas and assistance during the completion of this thesis.

Finally, I would like to thank the National Sciences and Engineering Research Council for the partial funding of the thesis under the NCE: GEOIDE project, and the staff of the School of Computer Science for their assistance during my graduate career.

Contents

Abstract	iii
Acknowledgment	iv
List of Abbreviations	ix
List of Figures	x
1 Introduction	1
1.1 Motivation	6
1.2 Goals	6
1.3 Main contributions	7
1.4 Thesis organization	7
2 Background	9
2.1 Data model	9
2.2 The relational model	11
2.2.1 Relation schema	11
2.2.2 Attribute	12
2.2.3 Tuple	12
2.2.4 Relation	13
2.2.5 Database	13
2.2.6 Derived relations	13
2.2.7 Advantages and limitations of the relational model	14
2.2.8 Approaches taken to overcome the limitations	16
2.3. Object-oriented concepts	18
2.3.1 Class	18
2.3.2 Object	20
2.3.3 Inheritance	20
2.3.4 Encapsulation	21
2.3.5 Polymorphism	21

2.3.6	Object identity -----	22
2.3.7	Advantages of OO modelling -----	22
2.3.8	Disadvantages -----	22
2.4	Object-oriented database management systems (OODBMS)---	24
2.4.1	Advantages of OODBMS -----	27
2.4.2	Problems with current implementations of OODBMS -	27
2.4.3	How does the OC data model solve the problems? -----	30
3	Object Category (OC) data model -----	31
3.1	The OC data model -----	31
3.1.1	Categories and classes -----	32
3.1.2	Inheritance-----	35
3.1.3	Internal representation -----	37
3.1.4	Indexes-----	37
3.1.5	Constraints -----	38
3.1.5.1	Referential Integrity Constraints-----	40
3.1.5.2	Keys-----	43
3.1.5.3	NOT NULL constraints -----	44
3.1.5.4	Triggers-----	45
3.1.5.5	Other constraints -----	46
3.1.6	Schema-----	47
3.1.7	Advantages of the OC data model -----	48
3.2	Object manipulation using the CategoryInterface -----	49
3.2.1	Attributes of categories-----	49
3.2.2	Making objects persistent -----	50
3.2.3	Retrieving persistent objects-----	50
3.2.4	Removing objects -----	51
3.2.5	Modifying objects-----	52
3.2.6	Other operations -----	52

3.3	Category specification language -----	53
3.3.1	Extent declaration -----	54
3.3.2	Relationship declaration-----	55
3.3.3	Key declaration -----	55
3.3.4	NOT NULL constraint declaration -----	56
3.3.5	Trigger declaration -----	56
3.4	Query Language -----	57
3.4.1	Return object type-----	57
4	Framework for a DBMS based on the OC data model -----	63
4.1	Architecture -----	63
4.2	Implementation issues -----	64
5	Related work: ODMG 2.0 standard -----	69
5.1	ODMG 2.0 standard -----	69
5.1.1	Logical view -----	71
5.1.2	Categories vs. implicit extents -----	71
5.1.3	Attributes or relationships -----	72
5.1.4	Constraints -----	73
5.1.5	Object query language-----	74
5.1.6	Keys -----	75
5.2	Unified Modeling Language-----	76
6	Illustration of an example application using OCS-----	78
6.1	The application -----	79
6.2	Defining categories using the ObjectCategory class -----	83
6.3	Data manipulation-----	85
6.3.1	Inserting objects -----	86
6.3.2	Retrieving objects -----	87
6.3.3	Removing objects -----	90
6.3.4	Modifying objects -----	90

6.4	Declaring constraints -----	91
6.4.1	Defining relationships -----	92
6.4.2	Defining keys -----	95
6.4.3	General purpose constraint definition -----	96
6.4.4	Creating triggers -----	99
6.5	Defining persistent class - OCPersistent class -----	101
7	Conclusion -----	102
7.1	Future works -----	106
7.1.1	Query optimization -----	106
7.1.2	Object migration -----	107
7.1.3	Views -----	108
Appendix		
A	The complete BNF for category specification language -----	110
B	Partial code source of the OCS package -----	113
References -----		125

List OF Abbreviations

ADT	Abstract data type
ANSI/SPARC	American National Standards Institute/Standards Planning and Requirements Committee
CAD	Computer-aided design
CAM	Computer-aided manufacturing
CASE	Computer aided software engineering
CSL	Category specification language
DBMS	Database management system
ER	Entity-Relationship
EER	Extended-ER
I/O	Input/Output
OC	Object category
ODBMS	Object database management system
ODM	Object-to-database mappings
ODMG	Object data management group
Oid	Object identity
OO	Object-oriented
OODBMS	Object-oriented database management system
OQL	Object query language
RMI	Remote method invocation
SQL	Structured query language
UML	Unified modeling language

List of Figures

Figure 2.1	The three level ANSI/SPARC architecture -----	26
Figure 3.1	Different categories defined over the same class -----	34
Figure 3.2	Example categories-----	36
Figure 3.3	Multiple inheritance-----	36
Figure 3.4	A relationship with additional attribute -----	43
Figure 3.5	OC database schema -----	47
Figure 4.1	Architecture of an OC data model based DBMS -----	65
Figure 5.1	UML – Class Diagram-----	77
Figure 6.1	EER diagram for a University's database -----	80

Chapter 1

1 Introduction

Since the early days of the database technology, the question of how to best represent real world objects and their relationship has been the driving force for new data models, hence new database technologies. The purpose of data models is to model the real world as closely as possible. The data models on which the database management systems (DBMS) are based upon distinguish them from each other. The data model used by a DBMS determines the data structures and operators that are available to the user. It also determines the features available for storage, retrieval and management of the data independently of any particular application. Almost all database systems that are commercially available today are based on one of the following data models: hierarchical [14,22, 30], network [14, 22,30,33], relational [8, 14, 22, 30, 32] and object-oriented [5, 21, 25].

Until the introduction of object-oriented (OO) concepts, the relational approach attracted the vast majority of database research and it represented the dominant trend in the market place. In fact, the relational model is considered, by many, as the single most important development in the history of the database field. However, the introduction of more powerful machines encouraged their use for new and complex applications that required

the services of a DBMS. The use of the relational model to satisfy the database requirements of these new applications is testing the model's capability to its limits.

Object-oriented database management systems (OODBMS), which are the integration of concepts from OO paradigm and DBMS, have been proposed as the solution to satisfy the requirements of the "new" applications and hence to overcome the limitations of the relational model. The "new" applications include computer-aided design (CAD), computer-aided manufacturing (CAM), computer-aided software engineering (CASE), geographic information system (GIS), and office automation. However, unlike the relational systems in which the data model preceded their implementation, OODBMS lacks an official standard data model. The proposed standard [6] published by the Object Database Management Group (ODMG¹) attempts to fill the gap. The following statement in [6] explains the lack of a generally agreed standard model better: "With respect to the specification of the system, we are taking a Darwinian approach. We hope that, out of the set of experimental prototypes being built, a fit model will emerge. We also hope that viable implementation technology for that model will evolve simultaneously".

Current implementations of OODBMS and data models proposed so far do not include collections as the logical view of the database. Furthermore, OO programming languages do not support the notion of class extent explicitly. Unlike relational systems, the lack of

¹ ODMG is a consortium of OODBMS vendors.

a simple collection based conceptual view makes OODBMSs hard to learn and difficult to apply for many application areas. The thesis defines and illustrates a new data model for OODBMS named Object Category data model, which uses collections as an integral part of OO databases. The data model is based on the concept of category [15]. A category is a typed collection of instance objects. The use of collections implies a clear and easy way of making objects persistent that is consistent with the concept of database and resembles the operations of insert, retrieve and remove objects. Unlike current data models, the OC data model makes a clear distinction between the concept of class as a data type and the concept of category as a collection of instance objects in the database.

A database based on the OC data model is seen as a set of interrelated categories. The objects in the database are organized as members of the different categories. A category's definition includes the type of objects it is supposed to maintain, and the set of integrity constraints its member objects must satisfy, including their relationship to other objects. The OC data model makes a clear distinction between class definitions and integrity constraints. Integrity constraints are properties of categories, and not classes. At least, four kinds of integrity constraints are supported: referential or relationship integrity, keys, NOT NULL constraints and triggers. Relationships are defined between categories. The data model supports binary relationship that may be one-to-one, one-to-many, many-to-one and many-to-many. Integrity constraints are added and/or removed from the categories dynamically. This approach of separating constraints from class definitions

makes class design very flexible. It also prevents class definitions from being changed and recompiled due to every minor change in the business rules, and makes maintenance of business rules much easier. A DBMS based on the OC model maintains the integrity of a database by subjecting the objects in the database to different integrity checks depending on their category membership.

The OC data model also provides a category/subcategory relationship between categories. The relationship defines the set inclusion semantics for categories. By definition, a category is a subset of its supercategory. Furthermore, its member objects are constrained by the integrity constraints defined over its supercategory. Users interact with the DBMS through the facilities provided by the Category interface. The interface includes all the methods that are required for insertion, retrieval and deletion of a single object as well as a collection of objects. Objects are made persistent and later removed from the database through the various insert and remove methods of the interface.

A framework for a DBMS based on the OC data model takes into consideration several issues related to the traditional DBMS aspects like transaction and concurrency control, as well as issues related to the complex structure of the objects and their organization in the database. DBMSs are based on client/server model architecture. The server side is divided into two big modules, storage manager and object manager, which at the same time is divided into several submodules. Each submodule is responsible for a portion of

the object manager's responsibility like transaction, concurrency control, caching, etc. Objects in a database are uniquely identified by their identity. Object identities could be pointers to physical addresses or logical identifiers that are independent of the objects' storage location. The object manager maps the objects from their internal representation to their in-memory representation. Furthermore, programming languages provide collection constructs that are equivalent to categories proposed for the OC data model. Therefore, a seamless integration of the programming environment and the DBMS would be one that "ties" the user specified memory resident containers with corresponding persistent containers on secondary storage that are defined as categories. Such mechanism would allow the user to access categories in the database as if they were an extension of the memory resident containers. To allow such functionality a number of issues, including buffering and transactional access, need to be addressed. Although a user accesses the data as if he/she has an exclusive control of it, objects in the persistent categories are subject to concurrent access by multiple users. As such a concurrency control mechanism that takes into consideration the complex structure of objects and the database's organization as a set of categories needs to be incorporated. Complex objects include attributes that reference other objects. However, propagating a complex object's lock to all the objects it references significantly reduces the system's ability to run concurrent processes. Therefore, objects are locked only when the users access them.

1.1 Motivation

This thesis grew out of an interest in current database research, mainly OODBMS, and the well-known fact that their implementations were not based on a well-formulated standard data model: In particular, a data model that is defined based on the OO concepts, that is independent of any programming language, and is suited to any application and not just to some. It is our hope that the data model proposed herein would help the ongoing effort to define a standard and pave the way for a new generation of database systems that combine the benefits of a DBMS in a truly OO environment.

1.2 Goals

The main objective of the thesis is to define and illustrate a new data model that better integrates OO concepts, as defined by OO programming languages, with database technology concepts. The data model fully adheres to the concepts of OO paradigm, is independent of any specific programming language, and is applicable for any general application. It is also simple to understand and use, and provides data independence of the application from the database. The goal includes outlining the different issues that need to be considered for an implementation of a DBMS based on the data model.

1.3 Main contributions

We have proposed a new data model for OODBMSs that meets the above stated goals. More specifically we have defined the OC data model, which introduces the concept of category as a collection of objects. The data model makes a distinction between classes as a data type definition mechanism and categories as collections of objects in the database. It provides class design flexibility and easy rule maintenance mechanism by separating class definition from constraint specification. It also provides data independence by establishing a clear distinction between the responsibilities of the DBMS and the applications that use it. The thesis outlines the different problems that are found in current implementations of OODBMSs and how the OC data model solves them. It also discusses several issues that need to be taken into consideration when implementing a DBMS that is based on the OC data model. We have also proposed a query language that can be used to select objects the same way SQL is used by relational systems. Finally, we have presented an example application that illustrates the various facets of the data model. The example utilizes a prototype implementation package that is based on the OC data model.

1.4 Thesis Organization

The thesis is divided into different logical groups. First, it starts by describing the concept of data model, followed by a short review of relational model, its limitations and

approaches taken to solve those problems. It then covers object-oriented concepts and the problems with current implementations of OODBMSs. Next, it defines the OC data model, the Category interface, and the category specification language. It then follows with a description of the architecture for a DBMS that is based on the OC data model. It also reviews the ODMG data model by comparing some aspects of it with the OC data model. Finally, it illustrates the OC data model by using an application that uses an implementation of the data model, and outlines directions for further future works.

Chapter 2

2 Background

The birth of OODBMSs is attributed to the good, reliable and easily diagnosable practices of the OO paradigm and the limitations of the relational model. It is also attributed to the need of OO programming languages to store and handle large volumes of data using a system that is more sophisticated than a simple file management system. The aim of this section is to give the background concepts that are important in order to understand the motivation of the thesis, the origin and current status of OODBMSs, and the OC data model. Particular attention is given to the concepts of the OO paradigm and their use by current implementations of OODBMS.

2.1 Data Model

A database is a self-describing collection of data elements that together with the relationships among those elements, presents a uniform service interface [18]. It is a logically organized and structured collection of interrelated data stored together. In order to describe the structure of a database, we need to first define the data model that it is based upon. Unfortunately, the notion of data model as it is now used in database technology and programming languages has been misused from its original definition [29].

In general, a data model is defined as a set of conceptual tools for describing the representation of information in terms of data [12, 31]. It is comprised of data types and structures, operations, and (consistency) constraints. In other words, it is a pattern according to which data are logically organized. It consists of named logical units of data and expresses the relationships among the data as determined by the interpretation of a model of the real world [28]. In order for a DBMS to understand the data model, it is essential that it is familiar with the concepts used to build that model. This is where the notion of database model emerges. Database models take into consideration the representation of real world objects as well as their organization and interrelationships within a database system. A database model is a specification of the concepts of the real world domain and elements of a particular application in a form understandable by a DBMS. A database model is specific to a particular application.

2.2 The Relational Model

Currently, the most widely used DBMSs are based on the relational model. The relational model is based on a simple and uniform data structure, the relation, and has a solid mathematical foundation. The relational data structure is based on the mathematical notion of a relation. In relational systems, the user perceives the database as a collection of relations. The SQL standard that defines the relational database specification uses the term table in place of relation. Each table has columns, and each column has a name and defined data type. The first normal form, imposed on the relational model, requires the values of each column to be *atomic*. They can not contain sets of any kind. Ironically the concept of what is atomic was never defined, but was generally taken relative to the smallest unit of information that can be accessed in a query. Each table contains zero or more rows. The content of each column in each row is represented as explicit data values. In particular, there are no pointers connecting one row to another. The relational model [8, 14, 22, 30, 32] is formally and precisely defined using the following terms:

2.2.1 Relation Schema

A *relation schema* R , denoted by $R (A_1, A_2, \dots, A_n)$ is made up of a relation name R and a list of attributes $A_1, A_2, \dots, A_n$. A relation schema is used to describe a relation. It defines what attributes are involved and the roles played by each attribute. For example:

the relation schema $\text{Person}(\text{SSN}, \text{Name}, \text{City})$ defines a person as some one with a social security number (SSN), who has a name (Name) and lives in a city (City). A relation schema is also called a relation intention.

2.2.2 Attribute

Each *attribute* A_i is associated with a domain, $\text{dom}(A_i)$, and it represents the role played by its domain in the relation schema. In the above relation schema, the domain "Names" could be associated with the attribute "Name: $\text{dom}(\text{Name})=\text{Names}$ ". A *domain* is a set of atomic values.

2.2.3 Tuple

A *tuple* t of the relation schema $R(A_1, A_2, \dots, A_n)$ is a set of (attribute:value) pairs $\{A_i:v_i\}$, where v_i is a value from $\text{dom}(A_i)$ for $1 \leq i \leq n$. The tuple (SSN:111 111 111, Name:Joe, City:Ottawa) is a tuple in the context of the above defined Person relation schema. There exists a person with social security number 111 111 111, who has the name Joe, and lives in a city named Ottawa.

2.2.4 Relation

A relation r of a relation schema $R (A_1, A_2, \dots, A_n)$, $r(R)$, is a set of n tuples of R . In mathematical terms, a relation is defined as a subset of the cartesian product of $\text{dom}(A_1)$, $\text{dom}(A_2)$, ..., and $\text{dom}(A_n)$. Adding a tuple to a relation or deleting a tuple from a relation results in a new relation. A relation is also called relation extension.

2.2.5 Database

For database management systems that are based on the relational model, a database is a collection of relation variables. It describes the complete state of an information model through its relation variables and can answer questions about its state. Similarly, a database has a schema and state. The schema describes the database. The data in the database at a particular moment in time defines the database's state.

2.2.6 Derived Relations

Derived relations are calculated from other relations known to the database. They are most commonly the result of requests to the database through relational expressions and queries. The requests can be permanently remembered (and recalculated) through views.

Herein after, unless a more precise definition is required, we will be using the terms table, row and column, and relation, tuple and attribute interchangeably.

2.2.7 Advantages and Limitations of the Relational Model

The relational model has its own advantages that made it very popular among the database community. These are:

- The underlying concept is based on a strong mathematical foundation.
- The simple tabular schemes and the uncomplicated relationship representation schema provide an easy to learn and consistent user interface.
- The comparatively simple and standard access language (SQL) shortens the learning curve.
- It provides data independence.

Despite all its advantages, the relational model has its weaknesses. The biggest problem with relational database systems, at least as they are implemented commercially, is the lack of mechanisms to define new domains or abstract data types (ADT) [9]. They are limited to data types implemented by the system: char, byte, string, integer, etc., and comparative operators on these types. These were assumed to be the types of atomic values. Some of the points that have been mentioned as shortcomings of the relational

systems in meeting the requirements of the "new" applications are given in [5, 7, 12, 21].

A summary of these major shortcomings is:

- Relational systems deal with a single object type: a relation. Furthermore, the attributes of a relation may come only from simple and fixed data type domains. Non-traditional applications require more complex object structures with nested objects.
- The relational model captures the factual data about applications, but not the "behavioral knowledge" or "domain-specific generic knowledge".
- Impedance mismatch between programming languages and DBMS. Applications that are written in a conventional programming language and that use relational systems retrieve data from a database using SQL. Since SQL is a declarative language and since data types used by programming languages are different from those used by relational systems, the data needs to be mapped from one data type to another.

The above can be summarized as the need for type level extensibility through the addition of new abstract data types to the database. Therefore, what is really needed is a way to extend the data type known to the database. The organization of the database as a set of containers (tables) is not the problem. If the relational model is not good enough to represent complex objects it is because of the limitations imposed on the definition of its relation schema. OO constructs allow us to define real-world objects, thus the relation schema, similarly but differently. The main problem with relational systems is not with how the objects are being organized (as members of a table) but how the objects are

defined. It is possible to overcome the problem by introducing a new way of defining objects, which is what classes provide. It is important to note that we are not trying to extend the relational model to solve the above mentioned problems. We just want to clarify the roles the table-like organization of the data plays in the data management. A table groups together a set of tuples and it defines the relationships and constraints its tuples must satisfy.

2.2.8 Approaches Taken to Overcome the Limitations

Two approaches are being taken to overcome the problems of relational model and develop OODBMSs [5, 21]:

1. Evolutionary: Since DBMSs based on relational model are the dominant product in the market place, this approach extends those systems with OO like constructs. It is being pursued by vendors of relational systems and is upward-compatible extension to the relational approach. They are called object-relational database systems.
2. Revolutionary: The other approach is to introduce database capabilities to already existing OO programming languages, or to develop an entirely new OO database language and DBMS with OO capabilities.

The first approach still does not solve the so called "impedance mismatch". The second approach suffers from the lack of standard database model for OODBMSs and mostly neglects the needs of the traditional business database applications. Herein after, we will use the term OODBMS to refer the ones that follow the second approach. Our proposed model, Object Category data model (OC data model) takes a new approach. It is evolutionary in the sense that it retains some of the concepts of the relational model, mainly the logical view of the database as a collection of containers. It also is revolutionary, because, although it is independent of any programming language's implementation, it fully supports OO concepts as they are used by programming languages. By doing so, we satisfy the requirements of both applications, traditional as well the new ones, and overcome the limitations of the relational model.

2.3 Object-Oriented Concepts

Object-orientation was first introduced in the context of programming languages. It provides practical benefits for the prototyping, modeling and programming of advanced applications, including high degree of code reuse. Object orientation allows a more natural representation of real world entities as well as abstract concepts. Most of the applications that currently use OO programming languages require the manipulation of complex and highly interrelated objects and most importantly they need to make use of database management systems. Some of the most fundamental concepts of the OO paradigm [13] are class, object, inheritance, encapsulation, and polymorphism. Although there is no a clear cut and agreed upon definition of those concepts [21], the next section attempts to give a general account of what those terms mean.

2.3.1 Class

Class is the main construct used by OO programming languages to define new types of objects that have the same structure and the same behavior. A class has a name and it defines the structural (set of properties or variables) and behavioral (set of methods)

aspects of its instances called objects. The data type of a property might be a class or a literal. For example: Person class could be defined, in Java² [17, 27], as follows:

```
public Class Person {
    static String    weightMeasurement;
    private String   lastName;
    private String   firstName;
    private Date     dateOfBirth;
    private City     city;
    private Double   weight;

    public Person (String lastName, String firstName,
                  Date dob, City city) {
        this.lastName = lastName;
        this.firstName = firstName;
        this.dateOfBirth = dob;
        this.city = city;
    }
    public String getLastName() { return this.lastName; }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getFirstName() { return this.firstName; }
    public String toString() {
        return getLastName() + ", " + getFirstName();
    }
    ...
}
```

The above Person class has six variables (weightMeasurement, lastName, firstName, dateOfBirth, livesIn, and weight), a constructor (Person()), and four methods (getLastName() and setLastName(), getFirstName() and toString()). A constructor is a special method used to create instances of the class. Distinction is made between instance variables and class variables (weightMeasurement). Changes made to a class variable

² Java is a registered trademark of Sun Microsystems

affect all instance objects of the class, while changes made to an instance variable affect only the instance object that is making the changes.

2.3.2 Object

All objects belonging to the same class have the same structural and behavioral aspects as defined by their class. Objects have states. The values and/or states of its properties determine the state of an object.

2.3.3 Inheritance

The concept of inheritance allows us to specialize and factorize definitions of classes.

Therefore, the class design consists of grouping the most general information into classes, which are then specialized into subclasses, specifying more specific behaviors with each step. Subclasses could inherit behavior as well as structure. For example, the Person class can be specialized into the Employee class as follows:

```
public class Employee extends Person {
    String    employeeNo;
    double    salary;

    public Person (String lastName, String firstName,
                  Date dob, City city, String employeeNo,
                  double salary) {
        super(lastName, firstName, dob, city);
        this.employeeNo = employeeNo;
        this.salary = salary;
    }
}
```

```
}  
    public double annualIncome() { ... }  
    public String toString() {  
        return super.toString() + ", " + this.annualIncome();  
    }  
}
```

The Employee subclass inherits all variables and methods of the Person class.

2.3.4 Encapsulation

Encapsulation provides a mechanism for objects to hide their implementation details from their users. Clients interact with an object through its public interfaces. For example, the instance variable *name* of the Person class can be read or updated through the public interfaces `getName()` and `setName()` only.

2.3.5 Polymorphism

Polymorphism refers to the use of the same message signature to address different methods in different classes. For example, a `toString()` message to a Person object prints the person's last name and first name. When the same message is sent to an Employee object, the printout includes the employee's salary.

2.3.6 Object Identity

Object identity has long existed in OO programming languages. The concept is even more critical for OODBMSs. Each object in a database is distinguished from all others by its identity, which is independent of the state of the object. The identity of an object remains the same during the object's lifetime within a database.

2.3.7 Advantages of OO Modeling

The OO paradigm has many advantages. Some of which are:

- It provides for more naturalness of the abstract representation of real world information than the relational model.
- It supports type extensibility through classes. The class structure encapsulates behavior as well as the traditional static attributes.
- The inheritance mechanism allows us to create new classes by subclassing, and thus reusing existing ones.

2.3.8 Disadvantages of OO paradigm

- It lacks a standard data model.
- The rich and extensible nature of the OO data model makes implementing OODBMSs more complex and quite challenging.

- The dynamic binding that is required by OO paradigm makes the programs slower to run.
- Data mining is harder because of the difficulty to discover new relationships between objects.

2.4 Object-Oriented Database Management Systems

A DBMS is a collection of programs that enables users to create and maintain a database [14]. It provides facilities for the definition, construction and manipulation of databases for various applications. It allows users to define and create the database, and populate it with data that is later retrieved and manipulated most probably concurrently by multiple users. A DBMS is usually divided into different software modules that act on the data and perform different functions on different levels of the DBMS architecture. The ANSI/SPARC architecture, illustrated in figure 2.1, divides database systems into three different levels called internal level, conceptual level and external level. The internal level deals with storing and retrieving data to and from the storage media. The software modules at the conceptual level are responsible for organizing the whole database according to a defined logical data model. They hide the physical characteristics of the database from higher levels. They also include programs for data consistency, security, etc. The highest level of the architecture directly communicates with users. It provides the users with different views of the database that address the specific needs of the user or a group of users. A DBMS includes software that maps data from the external level to the conceptual level, from conceptual to physical, and vice versa.

The need for OO programming language users to store their data without suffering from the "impedance mismatch" and the lack of expressiveness in current data models when applied to the new generation of applications are two of the main factors that motivated development of OODBMSs. OODBMSs try to provide an OO image of the stored data that can be modified and used nearly as if it were in transient memory. An early solution to the problem was to extend the programming languages with simple persistence facilities allowing objects to persist between application sessions. Since then, much effort has gone into developing DBMSs that store objects, and into defining OO data models.

Object-oriented modeling provides constructs for more "naturalness" of the abstract representation of real-world information. However, OODBMSs in their current implementations are application-oriented extensions of OO programming languages. They do not directly address the problems faced by the integration of the two technologies: OO design methodology and DBMS. From the utilization of terms to data organization they have been heavily influenced by:

- The programming language used to develop the DBMS.
- The particular application the designers of the DBMS were aiming at.

Database management systems are used for managing the repositories of objects that are used in different applications. The database model of a database does not necessarily have to be similar (identical) to the database model of a specific application. OO data

models proposed so far do not include collections as the logical view of the database. The use of collections by a DBMS implies a clear and easy way of making objects persistent

Figure 2.1: The Three Level ANSI/SPARC Architecture

that is consistent with the concept of database and resembles the operations of insert, retrieve and remove objects. Current OODBMSs provide “roots” as database entry points [6, 14, 23]. Other persistent objects are accessed through navigation that starts from one of the root objects. The question is how many root objects do we need to logically represent the database’s schema. In addition, most of them use garbage

collection methods that remove objects that are no longer referenced from one of the root objects. This forces users to maintain explicit collections. For this reason OODBMSs provide constructors for collections (set, list, etc.). However, they leave the responsibility of their maintenance to the programmer. By doing so they place the burden on the programmer to insure the correct structuring of the database. This circumvents one of the primary objects of a DBMS, which is to facilitate the management of object repositories.

2.4.1 Advantages of OODBMSs

The integration of OO concepts and DBMS technology offers the following advantages:

- All the strengths of OO modeling are available: inheritance, polymorphism, encapsulation, etc.
- Type extensibility through classes, which encapsulate behavior as well as the traditional static attributes.
- A simple relationship mechanism through logical inclusion (direct object reference).
- Simplicity of development through almost direct correspondence between the application's objects and the database's objects.

2.4.2 Problems With Current Implementations of OODBMSs

What are the problems with OODBMSs as they are implemented currently?

- Unlike the relational model, which has a good theoretical basis and is well defined, OODBMSs lack a common standardized database model and formal foundations. Constructs used by OODBMSs are direct copies of the constructs used by the programming languages that influenced them. However, the notion of OO data model in database systems is much broader than that found in programming languages. Programming languages do not have such notions as relationship and constraint. In addition, there is no single OO paradigm, and therefore there are varieties of OO data models [5].
- One of the differences between the OO approach and that taken by the relational data model is that the relational model explicitly maintains the collection of tuples. Collections are a natural way of representing groups of objects from the real world. OODBMS can not be made to work for the general database case unless the user maintains collections of his/her own. Doing so makes applications more complex, since they have to take some of the responsibilities that previously were under the DBMS.
- Object-oriented database management systems must take into account not only the new applications, but also the traditional business data processing applications. However, existing OODBMSs seem to ignore the needs of the traditional and focus more on the "new" applications. In fact most of them are targeted at "new" applications (CAD/CAM, CASE, Office automation, etc.), and are more application oriented than technology oriented [2]. In the long run there is a need for a single

database system that satisfy all kinds of database requirements. Few applications fit neatly into just one category [1]. In addition, a company might have all kinds of users, thus would require a database system that could provide all the requirements [5].

- A class is the construct used by OO programming languages to define abstract data types. Unfortunately, the influence of OO programming languages has led to a confusing and contradictory use of the same term [21]. Does a class represent the set of all possible instances (intention) or just the variable containing the instances we happen to be interested in at the moment (extension)? Almost all OODBMSs treat classes as extensions instead of (or as well as) treating them as data types.
- Class definitions are used as black boxes that contain aspects of the application programs as well as that of the database. This means that relationships, constraints, etc. are all part of the class definition. This idea of putting every thing in one place makes class definitions very susceptible to unforeseen errors or changes in the business rule.

Object-oriented data models try to model real world objects. However, adding an additional construct to the database model, namely category, defines better the logical view of objects in a database and their organization. A database based on the OC data model is seen as a collection of categories as opposed to a collection of objects.

2.4.3 How Does the OC Data Model Solve the Above Problems?

- The data model is independent of any programming language, and it includes the notions that are found only in the context of a DBMS.
- The use of category adds more expressiveness to the data model without the additional burden placed on the programmer.
- By making collections integral parts of the database the OC data model satisfies the needs of the traditional as well the “new” applications.
- It makes a distinction between class as a mechanism to define abstract data types (domain) and the structure used to maintain collection of objects, category. A class represents a definition of a real-world object and as a definition it “represents” all possible occurrences (instances) of the real world object. A category is a collection of actual instances in the database.
- The constraints, including relationships, the objects must satisfy are defined at the category level. Objects are checked against those constraints depending on their category membership. This approach provides data independence, and class design flexibility without sacrificing the nearly transparent handling of persistent objects.

Chapter 3

3 Object Category Data Model

As mentioned before, the objective of this thesis is to integrate OO concepts, as defined by OO programming languages, with database technology. The approach followed is to make the database OO by retaining all the concepts of the OO paradigm, and making collections an integral part of the logical view of the database. The OC data model described is independent of any programming language implementation, and it can be applied to systems with single inheritance, multiple inheritance, typed or untyped. The data model adheres to the basic properties of OO paradigm explained above.

The chapter is divided into four sections that describe the different aspects of the data model. Section 3.1 defines the data model and its components. The Category interface and its functionality for object insertion and retrieval are described in section 3.2. Section 3.3 lists the BNF for a category specification language while section 3.4 explains aspects of a query language for the OC data model.

3.1 The OC Data Model

The notion of a collection is very important to the concept of database management. A single design project might not need all the functionality of a DBMS. However, a

reference to a database implies a reference to a collection of whatever information of an enterprise we would want to manage, including collections of designs. The OC data model is based on the concept of category. A category groups together a collection of objects of the same type hierarchy. A database based on the OC data model is conceived as a collection of categories. The organization of objects in the database through the use of categories allows us to separate most of the situations that arise only on the context of DBMS from class definitions.

Categories are to OODBMSs based on the OC data model, what relations are for relational systems. They form the logical view of the database. Categories are the only entry point to the database. The information content in the database, at any given time, is perceived in terms of objects in categories. Objects do not exist outside the categories. And the operators available to the user for dealing with the database are mainly operators that manipulate objects in categories.

3.1.1 Categories and Classes

One of the main functions of a DBMS is to process a large number of interrelated objects. None of the most popular OO programming languages supports the notion of a type or class extent explicitly. This leads to an interesting contrast between DBMSs and programming languages with respect to the use of types and classes [21]. The OC data

model makes distinction between a class, which is the set of all possible instances, and a category, that is a collection of a finite set of actual instances of a class in the database. A class is a type and not its extension. Unlike a class, a category is time-variant.

Under the OC data model, classes define data types. A category is analogous to a relation variable of the relational model, which is a time varying collection of objects. For example, the above Person class defines the attributes and behaviors that are common to all persons. A category defined over the Person class groups together instances of the Person class that are known to the database. It also defines their relationship to other objects in the database, and it captures their collective properties and attributes, or any integrity constraints that need to be satisfied by the collection.

A Class addresses general properties of objects while a category has knowledge of individual instances of objects. The class can answer requests about the general characteristics of its instances. The category answers questions about the set of individual objects it contains at a specific time. Classes are defined by the application's programming language for a modular development of applications, while categories are defined by the DBMS for an easy organization and maintenance of the database.

Categories are typed. A category maintains a collection of instance objects that are of the same class or type hierarchy. If class B is a subclass of class A, then objects that are

Figure 3.1: Three categories (Employees, Managers and Non-Managers) could have been defined over the same class (Employee). They are related to the Departments category, which is defined over the Department class, through two different relationships: *WorksFor* and *Manages*.

instances of class B can be inserted into a category defined over class A. Objects are grouped into categories according to their type and their relationship to other objects.

Under the OC data model, objects do not exist outside categories. A persistent object is at least a member of some category. The OC data model provides a large degree of flexibility in its concept of category membership. In particular, objects can be members of more than one category at the same time. A category in a database is uniquely identified by its name. Furthermore, several categories can be defined over the same class. For example, we could define three categories for the Employee class: a category of managers, a category for non-managers, and a third one that contains all employees. Each one might have its own set of constraints and relationships.

3.1.2 Inheritance

Like the OO paradigm introduced in chapter 2, the OC data model includes inheritance-based category/subcategory or category/supercategory relationships between categories. The relationship defines the set inclusion semantics for categories. A category is a subset of its supercategory. Therefore all its objects are considered to be members of the supercategory too. In figure 3.1, objects in the Managers and Non-Managers categories are members of the Employees category too. Therefore, given figure 3.1 and their contents in figure 3.2, `Employees.select(lastName="Mark")` would return all the three employees whose `lastName` equals Mark. A category inherits all the characteristics of its supercategory. Characteristics of a category include its relationship with other categories and all other constraints. Therefore, objects in the Managers and Non-Managers categories are constrained by constraints defined over the Employees category. Category/supercategory relationships are defined explicitly. That is the system does not infer the category/subcategory relationship by looking at the signatures of the types or class definitions. The user must declare this fact which asserts that the subcategory collection is a subset of the supercategory collection.

EMPLOYEES	
Last name	First name

MANAGERS	
Last name	First name
Mark	Frank
Solomon	Bahta

NON-MANAGERS	
Last name	First name
Samuel	George
Mark	Tiger

Figure 3.2 Shows the logical grouping of persistent objects in three categories that are related through the category/subcategory relationship.

The OC data model allows multiple inheritance between categories. A category could be a subcategory of one or more categories. For instance, a category for teaching assistants could be declared as inheriting from the category defined for students and another one

Figure 3.3: A partial EER diagram for a school database with multiple inheritance

defined for employees. In addition to the characteristics inherited from its supercategories a category could also define characteristics that are unique to the instance objects it

maintains. The category for teaching assistants is complemented with a relationship that reflects the fact that a teaching assistant assists one or more sections.

3.1.3 Internal Representation

The internal representation of a category could be a set, a list, an array, or any other container class. The type of collection to use could be determined by the DBMS depending on the user's need for sorting or other characteristics, or the user could explicitly specify it. Methods that convert the internal representation of a category from one type to another could be provided. Objects are selected from a container either by pointer (iteration), or associative retrieval (query).

3.1.4 Indexes

Collections are usually supported by indexes. Indexes [4] improve a collection's ability to find objects given attribute values for those that form the key. Collections are an integral part of a database based on the OC data model, therefore it is very important that indexes are made part of the integral part of the database too. Indexes understand the category hierarchy. For instance, an index for the Employees category, figure 3.1, would include references to all objects in the Employees, Managers and Non-Managers categories. Several indexes can be defined for the same category.

3.1.5 Constraints

The choice of appropriate data structures together with the related operations of the data model will often be too coarse to exactly represent miniworld semantics. In this case, additional constraints (generally called consistency constraints or semantic integrity constraints) have to be specified, and a data model is required to provide appropriate support [11]. The OC data model separates constraints from class definitions. The technique of separating design concerns into behavior and constraint designs makes the design process easier and less prone to unforeseen errors. Because of the flexibility of OO programming, it is possible to prevent integrity violations by coding the controlling logic as part of the applications. However, the approach does not allow separation of algorithms (application codes) from constraints, and places a heavy burden on the programmer.

In a real application environment, support for accommodating changes to type structures is necessary. It is often not feasible to adjust the application code as type definitions evolve [26]. Making changes to design in OO systems can be expensive. OODBMSs try to create a seamless integration between the applications and the database. However, one of the main objectives of any DBMS should be to minimize the effects that changes to the database will have on the applications using it. Therefore, it is very important to achieve the first objective without neglecting the second one. In OC data model constraints are

defined over categories. The data model supports type change transparency by isolating constraint specification from the core definition of the class. Constraints may be added and/or removed dynamically. Although there are situations in which conversion of the existing data and re-compilation of all the applications is necessary, the approach followed by OC data model minimizes such situations. Key business rules can be coded as integrity constraints, and if a change takes place in the business rules, a single point of change updates the integrity rules for all applications, and ensures that rules are applied uniformly and consistently for all applications. Existing constraints can change and new ones can be inserted. An object's constraints have a dynamic nature that changes as the object moves in and out of categories. For example, the same employee object might be subject to different constraint checks depending on its membership to the Managers category or to the Non-Managers category. The separation of actual objects from the constraints that govern their participation in a category has the following advantages:

- Greater application independence: We define application independence as a degree of measuring the immunity of application to future changes in the business rules. The OC data model separates the actual objects from the rules governing their existence in the database, thus avoiding recompilation of all applications using the database due to changes in constraints. The constraints are plugged-in or unplugged dynamically.
- It captures better the inter-relationship between objects, which is only defined by the object's membership to one or more categories. Objects can be migrated from one

category to another. For example, an employee instance object could be removed from the Non-Managers category and inserted into the Managers category and vice versa.

- Makes rule maintenance much easier.

Constraints are predicates such that if not satisfied the system automatically raises exceptions for the user to handle. There are a number of integrity constraints that could be enforced on a database. The more integrity rules are enforced automatically, the better the system is. However, it is almost impossible to cover all of them [1, 21]. The integrity constraints discussed in this section include referential integrity constraints or relationships, unique or key constraints, Not Null constraints and triggers. A category is associated with a set of constraints.

3.1.5.1 Referential Integrity Constraints (Relationships)

Referential integrity for the relational model is a constraint requiring that a foreign key value must either (1) be null or (2) contain a value that actually exists in the referenced table. It implies that a tuple can not be deleted if its primary key is a foreign key for another tuple. OODBMSs should also handle referential integrity checks in a similar manner to avoid dangling pointers. A dangling pointer is one that is referencing a deleted object. Relationships in OC data model define the connection that exists between

categories. For example, the relationship *WorksFor* in figure 3.1 defines the connection between the Employees category and the Departments category.

It specifies the fact that each object in the category Employee works for some object in the Departments category. It is important to note that relationships are defined between categories. A relationship defined over a category applies only to its member objects.

Since it is possible to have different categories defined over the same class, instances of the same class might or not be subject to a specific integrity check, depending on their category membership. For example, employees in the Managers category are related to Departments through the *Manages* relationship, which is not true when applied to members of the Non-Managers category.

A relationship defined over the not necessarily distinct categories has properties that govern the way objects in the category participate in the relationship. We call these properties cardinality ratio of the relationship. The cardinality ratio specifies the minimum and maximum number of objects an object in a category can be related to. The most common types of relationship are one-to-one, one-to-many, many-to-one and many-to-many. The OC data model supports all four kinds of relationship.

The cardinality ratio of the relationship defined over the category C_1 in its relationship with category C_2 could be more precisely specified by a pair of numbers (i,j) where

$0 \leq i \leq j$ and $j > 0$. It means that objects in category C_1 can be related to at least i and at most j number of objects in category C_2 . The cardinality ratio of the relationship effectively defines referential as well as existential constraints on the objects that are members of the participating categories. The cardinality ratio of the *WorksFor* relationship between Employees and Departments could be specified as (1,1) and (0,n). The first one (1,1) means that an employee can only work for at least one and at most one department, while the cardinality for Departments (0,n) specifies that a department can have none or maximum of n number of employees working for it.

The (1,1) cardinality ratio effectively restricts the employees that can be members of the Employees category to those that work for departments in the Departments category. It also imposes some constraints on the remove operation for Departments. A department can not be removed from the category as long as there is at least one employee working for it. Or, if it is removed, then it should be propagated to the employees working for it. It means that removing a department implies the removal of all its employees. It is also possible to define relationships between objects in the same category. For example, the relationship between an employee and its supervisor can be defined over the same Employees category.

Relationships are not objects. Objects in categories are directly linked through their member attributes. For example, employees and departments could be linked through

department and *employees* attributes respectively. In [20] relationships are suggested to be objects by themselves, and to be maintained in their own category. The advantage of this approach is that it eliminates the need for a new class when the relationship has

Figure 3.4: A relationship with an additional attribute

additional attributes, figure 3.4. The disadvantages of doing so are:

- It makes the system slower, since it would require a join operation even for the simplest select operation.

In OC data model, relationships are defined between categories. Objects in categories are directly linked through their member attributes. A DBMS based on the OC data model provides mechanism for specifying relationships between categories.

3.1.5.2 Keys

Keys are used to uniquely identify an object given one or more of its attribute values.

They can also be used as a hint to provide faster searches when key values are specified.

For instance, employee number could be used as a key for employees working for a company. Keys that consist of more than one attribute are called composite keys. Object identities are internally defined and used by the DBMS to uniquely identify objects in the database. For a human being, key attributes are what object identities are to the DBMS.

Arguments have been made against the need of keys for OODBMSs. The user-defined keys, known from conventional databases, are unnecessary [24]. However more people believe that keys are essential ingredients of database query processing [6]. Without a key it would be very hard, although not impossible, to enforce the rule that specifies the fact that employee numbers should be unique within a company. In addition to object identities, objects in a category can be uniquely identified by the values of one or more of their attributes. A DBMS based on the OC data model provides a mechanism to define keys.

3.1.5.3 NOT NULL Constraints

Special case of domain integrity constraint is the NOT NULL constraint. NULL represents a missing attribute value. In many situations we want the value of an attribute to be neither empty nor NULL. For example, every person must have a name, therefore name should always be different from null and empty. The OC data model supports NOT NULL constraints.

3.1.5.4 Triggers

In practice there are different types of constraints that could be defined over a database. However, it is not practical to have a special integrity constraint type for each one of them. We need a general mechanism whereby users can express integrity rules and enforce them. The general mechanism should allow us to define almost all possible types of constraints, and should provide the constructs for the following actions: defining the integrity constraint and specifying the actions to be taken when an integrity rule is violated. The more general the mechanism is the more types of constraints it accommodates.

Triggers are special mechanisms for enforcing integrity constraints. In most cases when an integrity constraint is violated, the system will raise an exception that will probably end up interrupting the process that caused the violation. Triggers are a way to avoid such situation and correct the situation automatically. They are a combination of two things: a condition and an action. The condition is associated with the operations that a user can invoke on categories: insert object, remove object, modify object, and retrieve object. The condition can also be more precisely specified as before operation or after operation. For example, if we define that a department's manager salary should be the same or higher than the maximum of the other employees' salary, then modifying an employee's salary

should automatically fire a trigger that checks the managers' salary appropriately. The OC data model provides a mechanism for specifying triggers that are fired upon the invocation of operations that insert, modify and/or remove objects.

3.1.5.5 Other Constraints

In addition to the above mentioned explicit user defined constraints, there are other constraints that are implicitly and automatically enforced by the OC data model. For instance, the internal representation of a category could be a set, a list or an array. The type of collection used implicitly defines some constraints on the category. For example, if a set is used it implies that objects can not be duplicated on the same category.

3.1.6 Schema

For each database created, the DBMS stores and maintains a metadata or schema. The schema describes the database's structure and is used by the DBMS to perform its tasks. The information stored in the schema, figure 3.5, of a database based on the OC data model contains descriptions of the categories defined, the classes whose instances are

Figure 3.5: The Partial EER Diagram for the Schema of a Database Based on the OC Data Model

persistent, and the relationships between the categories. It also includes information on keys, constraints, and triggers, the indexes defined on the categories as well as other

information used to determine the physical and logical structures of objects. Security and user authorization information is also recorded. The schema can be stored in a collection of categories and maintained using the facilities provided for object manipulation.

3.1.7 Advantages of the OC Data Model

- The OC data model adheres to the basic properties of the OO paradigm. Therefore all the advantages of OODBMS, as mentioned above, are available in an OODBMS based on the OC data model.
- The model can be easily derived from the extended entity relationship diagram.
- It provides a framework for unifying value-based and identity-based access to objects.
- Offers a more natural, logical and consistent view of the database.
- By isolating constraint specification from the core definition of the class, it provides increased data independence, and flexible design approach.
- Satisfies the requirements of databases that combine the new applications (CAD/CAM, CASE, GIS, office automation, etc.) with the traditional business data processing applications.
- Reduces the overhead of working with collections, which are the basic components of any database.

3.2 Object Manipulation using the Category Interface

A database based on the OC data model is a collection of interrelated categories with a simple interface. The logical view of the database is composed by the categories it contains. Applications interact with the database through the categories and the operations defined in them. Objects are added to a category by using one of the insert methods. Once inserted, they are explicitly removed from the category using the remove operation. The Category interface includes operations that allow users to insert, retrieve and remove objects. These operations can be divided into single object operations and collection type operations. Collection type operations allow a collection of objects to be inserted, retrieved or removed in a single method call.

3.2.1 Attributes of Categories

Categories have two kinds of attributes: attributes defined by the class being maintained and attributes defined by the category itself. For example, the Employee class defines the attributes of each object, like name, birthday, etc., while the Employees category defines aggregate attributes like the number of objects in the category. For the purpose of querying, a category's attribute set includes only those attributes defined by the class it maintains.

3.2.2 Making Objects Persistent

Instances of a class can be persistent or transient. A DBMS based on the OC data model provides different mechanisms for objects to become persistent. Two of the possible mechanisms are explicit insert and object reachability. Through explicit insert, an object becomes persistent when it is explicitly inserted into a category through one of the different insert methods. At least two insert methods are provided: *insert (Object o)*, and *insert (Collection c)*.

3.2.3 Retrieving Persistent Objects

Access to persistent objects is done through categories. Any object in the database can be reached through one or more of the categories. The interface provides different ways of retrieving objects from a category. We can ask for a collection of objects that satisfy certain predicate or we can ask for a specific object. The number and type of access operations provided varies depending on the category's internal representation. For example, if a list is used, then it is possible to retrieve objects by their position in the list.

Another way of retrieving objects is through queries. Query is an important aspect of any DBMS. Usually a database stores and manages a large number of interrelated objects organized in a certain way. Queries are invoked against collections which categories are. Query is a mechanism that allows retrieval of objects by their value.

The Category interface provides operations that accept a predicate as an argument, like *select (Predicate p)* and *get(Predicate p)*. A predicate is a boolean expression that is evaluated against each object in the category. The result of invoking these methods on a category could be a collection of objects or a single object. By allowing value-based expressions to select objects, the OC data model provides a framework for unifying value-based and identity-based access.

3.2.4 Removing Objects

The lifetime of a persistent object spans from the moment it is created until it is completely removed from the database. Objects, once inserted into a category, are explicitly removed from the category using the remove operation. The set of remove operations includes operations that remove an object given its id, *remove (Object obj)*, and operations that remove objects given a predicate, *remove (Predicate p)*. An object continues being persistent until it is removed from every one of the categories it has been explicitly inserted into. Removing an object from the only category it is a member of automatically removes the object from the database.

3.2.5 Modifying Objects

The Category interface does not define an update method. To achieve a nearly transparent manipulation of persistent and transient objects, the preferred way for handling updates of objects is for the DBMS to automatically propagate the modifications to the database without the user having to call an update method.

3.2.6 Other Operations

In addition to the above-mentioned minimal functionality, the Category interface could be enhanced with operations that dynamically change a category's configuration.

Configuration of a category includes its internal representation, its relationship with other categories, and other constraints defined over the category.

3.3 Category Specification Language

From the user's point of view, there are two aspects to the definition of an application's schema for an OODBMS based on the OC data model: class definition and category specification. Class definition is the declaration of classes whose instance objects will be persistently stored in the database. Class definition can be written in the programming language's class definition language, or it can be written in an object definition language that is independent of any programming language, as in [6]. Class definitions are subject to the features and limitations of the host programming language used to write the application. The OC data model does not define a class definition language. The approach to be used is left for implementers of OODBMS.

Category specification language (CSL) is a language to be used for the declaration of categories. Two different approaches could be used for category declaration: through a specification language, or through low-level calls provided by the DBMS. The use of a specification language would require the use of a pre-processor that would accept category specifications and add them to the database's schema. The second approach avoids the need for a specification language and a pre-processor.

The CSL' syntax includes semantics for the specification of the following aspects:

category name, class name, internal representation, relationships, key constraints, NOT NULL constraints, and triggers. The top-level BNF for CSL is as follows:

```

<category_declaration> ::= <category_header>
                           { <category_body> }

<category_header> ::= category <category_name>
                     [extends <category_name_list>]

<category_body> ::= <extent_declaration>
                   [{ <relationship_declaration> }]
                   [{ <key_declaration> }]
                   [<not_null_declaration>]
                   [{ <trigger_declaration> }]

```

The category_header specifies the category name that is unique within a database and the categories from which it inherits. The body part, which starts with { and ends with } is composed of five parts. For the complete BNF of CSL see Appendix A.

3.3.1 Extent Declaration

As mentioned above, different collection types can be used as an internal representation of a category. In addition, a category is typed to a specific class. The BNF for extent declaration is as follows:

```

<extent_declaration> ::= <set_declaration> |
                        <list_declaration> |
                        <array_declaration>

```

It specifies the internal representation of the category as a set, list or array. The type of collection used determines the additional functionality of the category. For instance, the list declaration would require the specification of the attributes by which the category will be sorted out and the order of sorting. The BNF for list declaration is as follows:

```

<list_declaration>    ::=    list <class_name> [ <order_declaration> ]
<order_declaration> ::=    orderedBy <order_list>
<order_list>         ::=    <attribute_name> [ <order> ] [, <order_list> ]
<order>              ::=    ASC | DESC

```

3.3.2 Relationship Declaration

The specification of a relationship between two categories includes the related categories and the attributes used for the relationship. It should also include the cardinality of the relationship. A category can be related to one or more categories. The BNF of relationship is as follows:

```

<relationship_declaration> ::=    relationship <cardinality>
                                   <related_category_name>
                                   link <attribute_name> inverse
                                   <related_attribute_name>

<cardinality>              ::=    <one_to_one> | <one_to_many> |
                                   <many_to_one> | <many_to_many>

```

3.3.3 Key Declaration

Objects in a category are uniquely identified by the values of one or more of their attributes. The BNF of key specification is as follows:

<key_declaration> ::= key <attribute_list>

<attribute_list> ::= <attribute_name> [, <attribute_list>]

3.3.4 Not Null Declaration

One or more of the attributes of the objects in a category may be constrained to be not null. The BNF for Not Null declaration is as follows:

<not_null_declaration> ::= notnull <attribute_list>

3.3.5 Trigger Declaration

A trigger is an action that is fired every time an operation is invoked on a category. A trigger could be fired as a consequence of the insert, remove, modify or retrieve operation. It can also be exactly specified when it should be fired: before the operation or after the operation. The BNF for triggers is as follows:

**<trigger_declaration> ::= trigger <on_declaration>
<trigger_name> [<when>]**

<on_declaration> ::= onInsert | onRemove | onModify | onRetrieve

<when> ::= before | after

3.4 Query Language

Object Category data model organizes objects as members of categories, which is very similar to the way tuples are organized in relational systems. Because of this similarity the query language used for relational systems, SQL [10], can easily be adapted to fit the OC data model. However, there are one major issue that needs to be addressed when trying to adapt SQL to the OC data model: return objects type. When a select query that gets its attributes' values from more than one category is invoked, it returns objects of a specific type that are dynamically constructed by the query processor. We use the application described in chapter 5 to illustrate the proposed query language with examples.

3.4.1 Return Objects Type

Relational systems are based on only one data structure, i.e. relation. The result of a select query is always a relation. Unlike relational systems, OO systems provide mechanisms to define new data types based on the concept of class. OO databases, including OC data model, store and manage objects that are instances of classes. Most OODBMSs provide query facilities that span a single class and/or a single collection of objects [5]. As illustrated in section 3.2. since a category is a collection of objects of the same type, a query language that spans a single category would only require the

specification of the predicate. However, to achieve the objectives of SQL in relational models, we need to provide a mechanism for the execution of queries that freely combine attributes from objects in different categories, and/or different classes.

A select query in OO databases returns objects that are instances of an existing class. The values for the attributes of the objects might be extracted from one or more classes. However, SQL's select statement does not specify the kind of objects to be returned. Therefore, the new select statement for OODBMSs should state the attributes to be returned as well as the type of objects to be created. For instance, let us say that given the Students category and the Professors category we are interested in finding students whose last name is some professor's last name. However, we are not interested in retrieving neither student objects nor professor objects. We only want to retrieve some attributes from students (last name and section) and some attributes from professors namely education and section. SQL-like select statement to retrieve the information we are interested in would be:

```
Select s.lastName, s.sections, p.sections, p.education  
From Students s, Professors p  
Where s.lastName = p.lastName
```

The OC data model and all OO data models work at object level. However by reading the above statement it is not clear what kind of objects will be returned. We should be able to include in the statement an indication of what kind of objects we want to be created. We

should also be able to specify how the participating attributes (s.lastName, s.sections, p.sections and p.education) should be combined to construct objects. Our proposal is to define a query constructor that accepts three arguments: the select statement, the object type and the collection type. We define a Query class that is provided with the facilities needed to parse the select statement, execute it and return the objects in a specific format.

Let us first define the Std_Prfl class that we use to create the objects to be returned.

```
public class Std_Prfl {
    String      lastName;
    Sections    stdSections;
    Sections    prfSections;
    String      education;

    public Std_Prfl(String lastName, Sections stdSections,
                    Sections prfSections, String education) {
        this.lastName = lastName;
        this.stdSections = stdSections;
        this.prfSections = prfSections;
        education = education;
    }

    public String toString() {
        return lastName + ", " + stdSections.toString() + ", "
            + prf.sections.toString() + ", " + education;
    }
    ...
}
```

Now we can create a query and pass the above class name as an argument to the query constructor as follows:


```

public class TestQuery {

...
String qry = "Select s.lastName, s.sections stdSections,
             p.sections prfSections, p.education
From Students s, Professors p
Where s.lastName = p.lastName"
Query q = new Query(qry, Std_Prfl, Query.vector);
Vector rslt = (Vector) q.execute();
for (i=1; i<rslt.size(); i++) {
    System.out.println(((Std_Prfl)
(rslt.elementAt(i))).toString())
}
...
}

```

The query statement (qry) is a statement very close to the SQL's select statement. The space between "s.sections" and "stdSections" tells the query constructor that values from "s.sections" are to be stored in the attribute "stdSections". If both, the origin and destination, attributes have the same name, then the second attribute name may be omitted. The second argument specifies the type of complex objects to be created. The last argument, "Query.vector", indicates the collection type to be returned. The Query class could be provided with capabilities to return any kind of collection or iterator. In contrast to JDBC's SQL mechanism [19], the above query clearly states the attributes and the type of objects to return. Every returned tuple forms an object and is manipulated through the interface provided by some class, in this case by the Std_Prfl class. The class defines the structure and behavior of the objects returned. Therefore, there is no need for JDBC like *getString*, *getInt*, etc. methods to retrieve the attributes from the result set.

```

public class TestJDBC {

...
String qry = "Select s.lastName, s.sections as
stdSections, p.sections as prfSections, p.education
  From Students s, Professors p
  Where s.lastName = p.lastName"
Statement stm = connection.createStatement();
ResultSet rslt = stm.executeQuery(qry);
while (rslt.next()) {
    String str += rslt.getString("lastName") + " " +
        rslt.getObject("stdSections") + " " +
        rslt.getObject("prfSections") + " " +
        rslt.getString("education");
    System.out.println(str);
    str = "";
}
...
}

```

It is obvious from the above JDBC example that the returned attributes are treated as independent and not as part of an object. Furthermore, using JDBC we can not specify the behavior of the returned objects, nor the collection type. Therefore, whatever we want to do with the result (the behavior) has to be hard coded as part of the application, defeating the notion of encapsulation and OO. The only requirement for the proposed query mechanism would be that all attributes and methods that participate in the query must be publicly accessible. In addition, the return type class, in this case Std_Prfl, must provide either a zero argument constructor, or a constructor whose arguments are similar to the one specified in the attribute list. Following SQL's select-from-where format, the general syntax for select statement could be defined as follows:

```
Select <expression> [<attribute_name>] [{ , <expression> <attribute_name> }]
From <category> [{ , <category> } ]
[Where <predicate>]
```

<expression> could be any expression that yields a value of type that is compatible with the type of <attribute_name>. In particular, it could be an attribute's name or a method call. The **Where** part might be omitted in which case all objects will be retrieved.

CHAPTER 4

4 Framework for a DBMS Based on the OC Data Model

A DBMS that is based on the OC data model is used to manage databases that are collections of interrelated categories, which at the same time are collections of objects that are concurrently accessed by multiple users. To allow an efficient and concurrent use of the database, and at the same time maintain the consistency of the database, several issues need to be addressed. This section discusses the issues that arise when implementing the conceptual level of a DBMS based on the OC data model. The implementation takes into consideration aspects related to the traditional DBMS functionality like transaction and concurrency control, and others that are unique to the OC data model and its OO nature.

4.1 Architecture

The architecture, illustrated in figure 4.1, is based on client/server model and is divided into three levels according to the ANSI/SPARC standard: storage level, object level and application level. The storage level software is responsible for low level disk I/O operations, while the application level deals with a single user's requests. In between is the object manager, which at the same time is divided into different submodules related to transaction, concurrency control, recovery, security, caching, clustering, object

manipulation and schema management. Since each submodule could by itself be a subject of a thesis, it is not possible to cover all issues and implementation alternatives available for the different components.

4.2 Implementation Issues

As mentioned in the OC data model definition, objects in a database are uniquely identified by their identities. Because of this, persistent objects have two kinds of representation: in memory and in disk storage. The object manager handles users' request by transparently transforming the object's representation from in-memory to oid or vice versa. A table hashed by oid can be used to access an object in memory given its identity. Object identifiers can be pointers to physical addresses or logical ones. Using physical pointers makes object access much faster since the system already knows the physical address. However, the system has to be provided with additional functionality to allow object relocation. Using logical pointers has the additional overhead of maintaining a table that maps the logical pointers to physical addresses. Therefore, reading an object might imply two readings, one to get the object's physical address and the second one to retrieve the object itself.

Objects in a database based on the OC data model are organized as members of persistent categories. Furthermore, categories are the only entry points to the database, and an object in the database can be accessed through at least one of the categories. This fact can be exploited by the DBMS by first caching the categories into the users' main memory,

Figure 4.1: A Three Level Architecture for a DBMS Based on the OC Data Model

and then transparently redirecting object requests through them. Such mechanism would provide an easy and transparent way of moving objects from and to the database.

Furthermore programming languages provide container classes that are similar to the persistent categories used by the OC data model. What is required and to be provided by a DBMS is a mechanism that extends the memory resident containers in a way that allows them to span the structures of persistent categories. This mechanism of "tying" a memory resident container class with a persistent category would allow the user to traverse the data in the database as if it were in memory. To allow this seamless integration of memory resident container classes and persistent categories, issues like buffering mechanism and transactional access need to be addressed.

Databases are used to manage a huge number of interrelated objects. It is almost impossible to load a database's content into the main memory. Therefore we need a buffering mechanism that transparently sends and brings data to and from the secondary storage. Such buffering mechanism would use a cache to control the flow of data from and to the persistent categories. The first time a category is accessed the user is given a handle to the category. Alternatively, the category could be "tied" with an existing memory resident container object. The container object is then used to traverse the persistent category it is "tied" with as though the category was in memory. To achieve the transparent traversal of persistent categories, a buffering mechanism process can be running that manages the cache and controls the flow of data, sending data from and to the server as required. The other important issue that needs to be addressed is a mechanism for a transactional access of the objects in the persistent categories. It should

be recognized that while an individual programming environment accesses the data as if it has exclusive access to the data, the persistent collection of objects in the categories are shared resources and are subject to concurrent access (reading and writing) by multiple users. When uncontrolled this could create consistency problems to the database. Thus a concurrency control mechanism that prevents such problems from happening needs to be included. Such mechanism would need to take into consideration several issues.

Categories are collections of objects with complex nature. As such, an object can have attributes that reference other objects. Propagating an object's lock request to all the other objects that can navigationally be reached from it would significantly reduce the system's ability to handle concurrent requests. A mechanism that locks objects only when the user accesses them is difficult to implement and is prone to deadlocks. However, it significantly increases the number of processes that can run concurrently. Furthermore, concurrency control mechanisms require a constant request-reply interaction between the user and the server. Depending on the way users interact with the server, the mechanism could create a network traffic overhead. One solution proposed in [3] uses a "semi-optimistic" technique and attempts to reduce the overhead by minimizing the number of messages that are exchanged between the two processes. In a "semi-optimistic" interaction, the user sends a lock request to the server. However, the server does not send a reply to the user unless the request can not be satisfied.

Related to the buffering mechanism and transactional access discussed above is the problem of obsolete cache. As mentioned above objects in persistent categories are subject to concurrent access by multiple users. This means that an object in a user's cache can be modified by another concurrently running user making the object in the first user's cache obsolete. The solution to this problem requires a callback mechanism that allows the system to notify the user of the changes to the object so that the cache is updated.

Chapter 5

5 Related Work

5.1 ODMG 2.0 Standard

ODMG 2.0 is a standard for object database management systems (ODBMS) proposed by Object Data Management Group (ODMG). The group is composed of representatives of the major vendors of ODBMS/ODM products. The standard [6] defines from the simplest aspects, object and literal, to the most complicated aspect of a DBMS that must be satisfied by products that comply with the standard. The major components of the standard are:

- Object model
- Object specification language
- Object query language
- C++ binding
- Smalltalk binding
- Java binding

The object model defines the common data model, including the metadata, to be supported by all implementations. The object specification language includes an object definition language (ODL) and object interchange format (OIF). Object query language is

a declarative language that is used for querying and updating database objects. The last three components define the object model's binding to three programming languages: C++, Smalltalk, and Java.

A data model for DBMSs should be characterized by organization structure, valid operations and constraints that must be enforced by the DBMS. The attempt by ODMG to define a standard for ODBMSs is a big step towards the fast development and acceptability of the technology. However, as important as it is, the standard is plagued by the same problems that are found in current implementations of OODBMSs. The document states that "We have picked and combined the best features of implementations we had available to us". Current OODBMS' implementations are application oriented. As such, they address only aspects that are relevant in the context of the specific set of applications they are targeted for. Instead of attempting to integrate the different implementations, a better approach would have been to come up with a new database model that better integrates OO concepts into the DBMS technology. A new data model for new generation of OODBMSs that can be used for any kind of application.

The OC data model does not define what an object is. It adheres to the generally agreed use of OO terms by programming languages. In this section, we describe some aspects of the ODMG data model and their counterpart in the OC data model. The standard by ODMG is still a continuing work and as such it will be revised and enhanced.

5.1.1 Logical View

The OC data model includes collections as the logical view of a database. A database based on the OC data model is seen as a collection of categories, which are the only entry points to the database. The ODMG 2.0 states that “the application’s object model is the database’s (logical) schema”.

5.1.2 Categories vs. Implicit Extents

The ODMG document states that “Many people believe that extents are essential ingredient of database query processing. Implicit extents and keys would be preferable to explicit mechanisms involving named collections, yet there are challenging engineering issues that must be faced to rationalize these capabilities with the notions of transitive persistence and dynamic storage management herein presented”.

It first completely agrees with our belief that collections are essential part of a DBMS. However, it differs from our use of categories. There are, of course, challenging issues that must be faced by OODBMS implementers. However, this is not a reason to shy away from their use. The use of categories, as described in this document, has advantages over the implicit extents. Some of the advantages are:

- Categories are more than a collection of objects. They define the relationships between member objects, and offer operations for object manipulation and inter-category communication.
- Implicit extents do not give the user the freedom to organize the database as he/she sees it fit.
- OC data model allows objects to be members of more than one category, which allows them to be subject to different constraint checks. This approach offers some support for object migration, which is not available if we use implicit extents.

5.1.3 Attribute or Relationships

According to the standard “Two kinds of properties are defined in the ODMG Object Model: attribute and relationship. An attribute is of one type. A relationship is defined between two types, each of which must have instances that are referenceable by object identifiers”. It also states that “Relationships are not directly supported by C++. Instead, they are supported in ODMG by including instances of specific template classes that provide maintenance of the relationship. The relation itself is implemented as a reference (one-to-one relation) or as a collection (one-to-many) embedded in the objects.” An attribute might be used by the DBMS to enforce referential integrity, but it is still a normal attribute. The OC data model defines relationships as properties of categories and not objects. As such, an application’s data definition does not have to change to

accommodate the DBMS's requirements. Furthermore, embedding the relationship in the class definition prevents the user from using that particular attribute the same way as he/she uses any other attribute. Relationships are more than direct references between objects. They are integrity constraints that must be enforced by the DBMS. Programming languages do not support the notion of relationship. A standard for persistency of OO programming language objects in databases should accept an application's objects as they are defined by the programming language used, and complement them with the minimum possible extension without altering the way they are usually used.

5.1.4 Constraints

The document says nothing about constraints that are neither relationship nor keys. It is the class designer's responsibility to include every constraint check as part of the class definition. This means that every simple rule change would require the recoding of the class definition, recompiling the application, and changing the schema in the database. Doing so is a very high price to pay because of the fact that changes to OO systems, in fact to any system by that matter, are very expensive. A data model for OODBMS should make a clear distinction between the static and volatile parts of object definition. The OC data model associates constraints with categories and not class definitions.

5.1.5 Object Query Language (OQL)

The ODMG 2.0 standard defines an object query language. In general queries are mainly used to select objects from collections, which categories are. Relational systems use SQL to access any object in a database. The language is relatively easy because of the uniform organization of the database. Tables are the only entry points to a relational database.

Similarly, the OC data model defines a database as a collection of categories, which are collections of objects. Therefore, it lends itself perfectly to the use of a query language in the manner SQL is used by the relational model. The same can not be said about ODBMSs based on the ODMG 2.0. An OO select statement has five parts and is of the following form:

`(collectionType<class>) (C1, C2, C3, ..., Cn).select (attributeList, predicate)`

The “collectionType” and “class” specify the type of collection used as a container and the type of objects to be returned respectively. C_i, 1 ≤ i ≤ n, indicate the categories that participate in the query. The “attributeList” is a list of attributes that together will form objects of type “class”. The “predicate” is used to filter the objects to be returned.

ODMG’s OQL is defined to “query denotable objects starting from their names, which act as entry points into a database. A name may denote any kind of object, i.e., atomic, structure, collection, or literal”. The lack of uniformity of input objects makes the use of OQL in ODMG 2.0 less intuitive.

5.1.6 Keys

Keys are used to uniquely identify objects from a collection of objects. In the OC data model keys are declared against categories, which by definition are collections of objects. The ODMG standard states that “The scope of uniqueness is the extent of the type: thus a type must have an extent to have a key”. Unlike ODMG 2.0, in OC data model collections and keys are an integral part of the database.

5.2 Unified Modeling Language (UML)

UML[16] is a modeling language to express the results of an OO analysis and design. It includes diagrams to represent classes, use cases, activity diagrams, ... etc. In contrast to the OC data model and the ODMG 2.0 standard, UML is not concerned on how to design and develop an OODBMS. However, as a general purpose OO modeling language, it can be used to draw the diagrams of an OO database model. Specifically class diagrams, which are used to illustrate class models, are comfortably similar to database models. In this section we will illustrate how UML notations, instead of EER diagrams, could have been used to express the design of most of our examples.

A class diagram describes a system in terms of objects and the relationships among them. As mentioned before, a database is a logically organized and structured collection of interrelated data objects stored together. The class diagram modeling technique includes notations to express the objects design (attributes and operations) and the relationships among them. The two principal kinds of relationships that can be established between objects are associations and subtypes. Associations are used to express referential integrity constraints, including cardinality ratio (multiplicity), between participating objects. Subtypes specify the IS-A relationship between objects. For example, the one to many *worksFor* relationship (figure 3.1) between employees and departments can be

represented as an association between objects of type Employee and Department. (figure 5.1)

Figure 5.1 UML – Class diagram

The above example also illustrates the notations to express subtype relationships and constraints. As with EER, the OC data model can also be easily derived from a database's class diagram.

Chapter 6

6 Illustration of an Example Application using OCS

The information in this section presents an example application that illustrates the various facets of the OC data model, and describes the implementation aspects of the OCS. OCS is a package based on the OC data model that provides higher level abstractions for manipulation of typed persistent categories, and for stable storage of objects. It uses Java as the programming language and ObjectStore³ as the persistent store. Since the objective of the thesis is rather to define a data model and prove that it is realizable, we used ObjectStore as the repository manager. The public interface of OCS consists of a set of operations mostly oriented towards inserting and retrieving of objects from categories. The basic operations are operations that add, select, and remove a single object. In addition, we have included collection-oriented operations that do the same. It also includes operations needed for creation and removal of categories. The system has also implemented operations to dynamically add and/or remove relationships and other constraints from categories. The two main classes of the OCS package are ObjectCategory and Category. The ObjectCategory class is used to create and manipulate categories, while the Category class is basically used to insert, retrieve and remove objects. The system maintains references to all categories that are created.

³ ObjectStore is a registered trademark of ObjectDesign Inc.

6.1 The Application

Let us imagine we are developing a database for a University. The main players in a University are the students, the professors, other staff members and the courses offered by the University. Usually courses have requirements, and they are divided into sections and taught by different professors. Furthermore, some of the students work as teaching assistants helping students and marking grades. The EER diagram for the database, illustrated in figure 6.1, is composed of seven entities (Course, Section, Student, TA, Staff, Professor and Employee). The relationships defined between the entities are:

- a course has one or more required courses
- a section belongs to a course, and a course has many sections
- a student takes one or more sections
- a TA assists in only one section, and a section may have many TAs
- a TA is a student as well as a staff
- a professor teaches one or more sections
- a professor is an employee
- a staff is an employee
- staff and professor are disjoint (not overlapping)

Figure 6.1: Partial EER Diagram for a University Database

Once we define the EER diagram, we need to decide and define the classes and categories to be created and maintained by the database. For this application, we define a class (Course, Section, Student, TA, Professor, Staff, Employee) per entity. The Professor class and the Staff class would be defined as subtypes of the Employee class. The TA class would inherit from the Student class and the Employee class. For instance, using the Java

programming language the definitions for Employee and Professor classes would look like the following:

```
public class Employee {

    String    firstName;
    String    lastName;
    String    no;
    String    phone;
    double    salary;

    Employee (String no, String firstName,
String lastName) {
        this.no = no;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public void setPhone (String phone) {
        this.phone = phone; }
    public String getPhone () { return phone; }
    public void setSalary (String salary) {
        this.salary = salary; }
    public double getSalary () { return salary; }
    ...
}

public class Professor extends Employee {
    String    education;
    String    tenure;
    Vector    sections;

    Professor (String no, String lastName,
String firstName) {
        super (no, lastName, firstName);
        this.section = new Vector();
    }
}
```

```
public void setEducation (String education) {
    this.education = education; }
public String getEducation () { return education; }
public void setTenure (String tenure) {
    this.tenure = tenure; }
public String getTenure () { return tenure; }
public boolean addSection(Section section) {
    sections.add (section); }
    public boolean removeSection (Section section) {
    sections.remove (section) }
    ...
}
```

The next step is to define the categories that will contain the objects. For this application, we create one category (Courses, Sections, Students, TAs, Professors, Staff, Employees) for every class defined. The category hierarchy would resemble the subtyping hierarchy between the classes. Therefore, the category for teaching assistants will be a subcategory of the Students category and the Staffs category. In addition, the categories for staff and professors will be subcategories of the Employees category. Using the category specification language listed in appendix A, for instance, the Employees category and the Professors category are specified as follows:

```
category Employees {
    extent set Employee indexedBy employeeNo;
    key employeeNo;
}

category Professors extends Employees {
    extent set Professor;
    relationship one2Many Sections link sections inverse
    professor;
}
```

The Employees category is defined as the container for objects of type Employee. The collection will be indexed by *employeeNo*, which should be unique within the category. The Professors category is defined as extending the Employees category. Therefore, all objects in the Professors category will automatically be members of the Employees category, and the key constraint, *employeeNo*, defined on Employees category would also apply to Professors category. Furthermore, the Professors category defines a relationship with the Sections category through the *sections* and *professor* attributes found in the Section and Professor class respectively.

6.2 Defining Categories Using ObjectCategory Class

The OCS package provides the ObjectCategory class, which defines operations that are not specific to any category. The operations are used to manage categories. They include methods that add, get and remove categories. All the methods are static. The class provides *createCategory()*, *getCategory()* and *removeCategory()* operations to create, get and remove a category respectively. Since a Category instance is an object by itself it can be stored in the database like any other object. The *createCategory* operation requires two arguments: category name and class type. An overloaded *createCategory* operation requires a third argument, an array of category names, that specifies the category's supercategories. The operation raises a *CategoryNameAlreadyExists* exception if there exists a category with the same name. The other two operations only require the

category's name as argument. They both raise the *CategoryNameNotFound* exception if a category by that name could not be located. The class also defines static constants that are used by the system. The removal of a category implies two things: removal of all objects from the category and its subcategories, and removal of the category and its subcategories from the database's schema. For instance, using the *ObjectCategory* class we can easily define *Employees*, *Sections* and *Professors* categories as follows:

```
Public class TestCreateCategory {  
  
    ...  
    // create the Employees category  
    Category ec = ObjectCategory.createCategory(db,  
        "Employees", "Employee");  
  
    // create the Professors category  
    Category pe = ObjectCategory.createCategory(db,  
        "Professors", "Professor", "Employees");  
  
    // create the Courses category  
    Category cc = ObjectCategory.createCategory(db,  
        "Courses", "Course");  
    ...  
}
```

The *createCategory(db, categoryName, className)* method and its overloaded version *createCategory(db, categoryName, className, superCategoriesName)* create a persistent category named *categoryName* in the *db* database. The category is typed to *className* and inherits from the categories specified in *superCategoriesName*. Only instances of the *className* class or its subclasses may be inserted into the category. A category groups

objects of the same type (in the sense of the `isInstanceOf()` method of `java.lang`). The method throws a *CategoryNameAlreadyExistsException* if a category by the name *categoryName* already exists in the database. The argument *db* is a reference to the database where the category is to be created. The signatures of the methods *createCategory*, *getCategory* and *deleteCategory* are as follows::

```
public static Category createCategory(Database db, String
    categoryName, String className) throws
    CategoryNameAlreadyExistsException
```

```
public static Category createCategory(Database db, String
    categoryName, String className, String
    superCategoriesName) throws
    CategoryNameAlreadyExistsException
```

```
public static Category getCategory(Database db, String
    categoryName) throws CategoryNotFoundException
```

```
public static void deleteCategory(Database db, String
    categoryName) throws CategoryNotFoundException,
    ConstraintViolationException
```

6.3 Data Manipulation

The *Category* class is the most important class of the system. It provides most of the functionality required for data manipulation, including maintenance of the integrity constraints.

6.3.1 Inserting Objects

The `Category` class implements operations that are used to add objects to a category. Two methods are implemented: *add (Object obj)*, which inserts a single object to the category, and *add(Collection c)* that adds a collection of objects.

```
public void add (Object obj) throws
    WrongObjectTypeException, ConstraintViolationException

public void add (Collection c) throws
    WrongObjectTypeException, ConstraintViolationException
```

Both methods check whether the object or objects in question are instances of an acceptable class, and raise *WrongObjectTypeException* if not. For instance, to insert a new `Professor` object to the database we first get a reference to the category and call the *add* method as follows:

```
public class TestAdd {

    ...
    // Get a handle to the Professors category
    Category pc = ObjectCategory.getCategory(db,
        "Professors");
    //Create a new Professor object
    Professor prf = new Professor("Mark", "Rider", "Phd");
    // Add the object to the Professors category
    pc.add(prf);
    ...
}
```

6.3.2 Retrieving Objects

The system provides a mechanism for retrieving objects through queries or iterators.

Query is performed by passing a predicate expression as an argument to the *select* operation of a category object.

```
public Set select (String predicate)
```

A category applies the predicate expression (an expression that evaluates to a boolean result) to all elements it contains, including objects in its subcategories, and returns a collection of all objects for which the expression results true. For instance, the following example retrieves all employees whose last name is "Mark". Since Staff and Professors are subcategories of Employees, the query would search in all three categories.

```
public class TestSelect {  
    ...  
    Category emp = ObjectCategory.getCategory(db,  
        "Employees");  
    Set marks = emp.select("lastName == \"Mark\"");  
    ...  
}
```

If the same method were sent to the Professors category, it would search in the Professors category only. The only requirement is that the attribute *lastName* must be public. All attributes and methods referred to in a predicate's expression must be public. If a method is specified, then it must return something. The *select* method returns a Set that contains the objects that satisfy the predicate. If the category does not find any matching object, it

returns an empty Set. An equivalent *get* operation retrieves the first object that satisfies the predicate expression. If no object in the category satisfies the predicate, the method returns null.

```
public Object get (String predicate)
```

Objects can also be retrieved through the *search* operation. *Search* allows the user to find an object without prior knowledge of its category membership. Unlike the *select* and *get* operations, which retrieve objects from the category receiving the message, the result of *search* operation is the union of all objects of type *classType* that satisfy the predicate. All categories that store objects of type *classType* are searched.

```
public Set search (Class classType, String predicate)
```

The *classType* parameter provides the context in which the predicate should be evaluated. The message can be sent to any one of the categories in the database. For instance, the following instruction retrieves all employees in the database whose last name is "Mark".

```
Public class TestSearch {  
  
    ...  
    Category std = ObjectCategory.getCategory(db,  
        "Students");  
    Set marks = std.search (class.forName("Employee"),  
        "lastName == \"Mark\"");  
    ...  
}
```

It gets the same result as the above *select* instruction. However, it is not as fast as the above one since it has to first determine the search strategy. To eliminate duplicates and minimize processing, the program first starts analyzing the most specialized categories. An alternative way of accessing objects in a category is through the Iterator class, which helps navigate through the category's objects. The *iterator()* method of the Category class returns a handle to an Iterator instance. The returned Iterator object is an instance of the OIterator class that is an implementation of the java.util.Iterator interface. Iterators can be used to traverse categories as well as to remove objects from categories. The signature is as follows:

```
public Iterator iterator()
```

For instance, the following code retrieves all objects in the Employees category and prints them to the standard output.

```
public class TestIterator {
    public static void main(String argv[]) {
        ...
        Iterator itr = ObjectCategory.getCategory(db,
            "Employees").iterator();
        while (itr.hasNext()) {
            Employee emp = (Employee) itr.next();
            System.out.println(emp.toString());
        }
        ...
    }
}
```

6.3.3 Removing Objects

Removing objects is done through the operations of *remove* and *removeAll*. In both cases, if the object being removed is not a member of any other category, it will automatically be removed from the database. The operation *removeAll()* removes all elements from the category. However, it does not remove the category from the database. It only removes the objects from the category. If the removal of an object violates a constraint, the methods throw a *ConstraintViolationException* exception. The signatures of the *remove* and *removeAll* methods are:

```
public boolean remove (Object obj) throws
    ConstraintViolationException

public boolean remove (Collection c) throws
    ConstraintViolationException

public boolean removeAll () throws
    ConstraintViolationException
```

6.3.4 Modifying Objects

The system maintains the database's integrity automatically as objects are added and/or removed from categories. However, it is the user's responsibility to call the *update* method when a persistent object's members are modified. The *update* method is needed because OCS being a library has no way to automatically detect when changes are made to an object. Furthermore, objects can be members of more than one category. This

means, whenever an object is modified, all constraints defined on all categories the object is member of must be checked. OCS handles this by keeping, with each persistent object, a list of the categories the object is member of. Whenever an object is inserted, modified or deleted the list is consulted to make sure that all constraints are enforced. For example, suppose professor Mark adds "cs95.501" to its list of courses. To make sure that the database's consistency is maintained the application should call the *update* method as follows:

```
Public class TestUpdate {  
  
    ...  
    Professor prf = (Professor)  
        ObjectCategory.getCategory(db, "Professors").get("last  
        Name == \"Mark\");  
    Course crs = (Course)  
        ObjectCategory.getCategory(db, "Courses").get("courseN  
        o == \"cs95.101\");  
    prf.addCourse(crs);  
    //call update on the modified object  
    prf.update();  
    ...  
}
```

The *update* method is defined in the OCPersistent class described in section 6.5.

6.4 Declaring Constraints

There are different mechanisms to enforce integrity constraints on a database. Some special integrity constraints could be supported automatically by the system whereas

others can be added to the database through a mechanism that allows users to express and enforce new integrity rules. Constraints could be violated only by operations that modify the state of the database (updates, inserts and deletes). An object's operation is said to satisfy its constraints if its insertion, modification or removal does not violate the integrity of the database. Such operation is said to be legal. OCS provides facilities to specify and enforce referential integrities, keys, domain integrity constraints and triggers.

6.4.1 Defining Relationships

Relationships are a special case of constraints called referential integrity constraints.

Referential integrity constraints are one of the most important and most frequently used mechanisms for maintaining a database's integrity. The purpose of referential integrity constraints is to indicate that a reference to an object is legal only if the referenced object exists in a particular collection. For example, a referential integrity constraint between the Sections category and the Professors category could be that, the *professor* attribute for a course object should be limited to one of the professors in Professors category. The inverse is, that any section in the *sections* attribute of a professor object must be a member of the Sections category.

Referential integrity constraints are enforced by the system with a little input from the user. The user's responsibility is limited to specify the related categories and the

attributes through which they are related. Relationships are defined between categories. Only binary relationships are supported. The attribute on the many side of a relationship should always be of type Collection or one of its subclasses. Four classes that represent the four types of relationships that are commonly used are implemented: one-to-one, one-to-many, many-to-one, and many-to-many. The following is a partial listing of the Relationship class.

```
public class Relationship {
    private String name;
    private int type;
    private String attributeName;
    private String relatedCategoryName;
    private String inverseAttributeName;

    Relationship (String nm, int type, String atrnm,
        String rcn, String inv_atrnm) {
        this.name = nm;
        ...
    }
    ...
}
```

A category can be related to more than one category. One instance of the Relationship class is maintained for each of the relationships the category is involved in. The relationship type is constrained to be One2One, One2Many, Many2One or Many2Many. The constants are defined in the ObjectCategory class. The components are passed as arguments to the *addRelationship* method.

```
public void addRelationship (String relationshipName, int
    relationshipType, String attributeName, String
```

```
relatedCategoryName, String inverseAttributeName)
throws, RelationshipNameAlreadyExistsException,
RelationshipAlreadyExistsException
```

For instance, the following statement declares a one-to-many relationship named `Pr_Sec` between the Professors category and the Sections category through the inverse attributes of *sections* and *professor*.

```
public class TestRelationship {

    ...
    // Get a handle to the Professors category
    Category pc = ObjectCategory.getCategory(db,
    "Professors");
    // Establish relationship with the Sections category
    pc.addRelationship("Pr_Sec", ObjectCategory.One2Many,
    "sections", "Sections", "professor");
    ...
}
```

Relationships need to be specified only once. The system inserts and/or removes the inverse part of the relationship automatically. For instance, the relationship between Sections and Professors categories need be specified only once, either through the Sections category or the Professors category. Relationships are uniquely identified by their names, which is used as the only argument to the *removeRelationship* method.

```
public void removeRelationship (String relationshipName)
    throws RelationshipNotFoundException
```

The above inserted relationship is removed from the database with the following statement:

```
...
ObjectCategory.getCategory(db,
    "Professors").removeRelationship("Pr_Crs")
...
```

There are other four relationship-related classes implemented by the system:

Relationship121, Relationship12M, RelationshipM21 and RelationshipM2M. They correspond to the four types of relationship the system supports. They all implement the RelationshipInterface interface. Whenever required, the system dynamically creates an instance object of the right relationship class and calls the object's *validate* method.

6.4.2 Defining Keys

Keys are relevant for any DBMS that uses collections. They allow objects to be uniquely identified given the values and/or states of one or more of their attributes. OODBMSs use object identities to uniquely identify objects. However, those identities are usually hidden from the user. OCS offers a mechanism to add and remove keys from categories. Keys in a category are uniquely identified by their names. They can consist of one or more attributes. The *addKey* method is used to add a key constraint to a category.

```
public void addKey (String keyName, String[]
    attributeNames) throws KeyNameAlreadyExistsException
```

The method takes as arguments the name of the key, which should be unique within a category, and an array of attribute names. Only attributes that are defined by the

contained class are accepted. Attributes that are part of the key should all be publicly accessible. The *removeKey* method is used to remove a key from a category.

```
public void removeKey (String keyName) throws  
    KeyNotFoundException
```

For instance, to make sure that the employees' no is unique through the database we define a key on the Employees category with the attribute *employeeNo* that is defined in the Employee class.

```
public class TestKey {  
  
    ...  
    String[] keyAttributes = {"employeeNo"};  
    ObjectCategory.getCategory(db,  
        "Employees").addKey("EmployeeNumber", keyAttributes);  
    ...  
}
```

From now on, the system makes sure that that the *employeeNo* is always unique when an employee is inserted to Employees, Staff or Professors categories, or when an employee object is updated. The system uses Java's reflection API to dynamically retrieve attribute values.

6.4.3 General Purpose Constraint Definition

Keeping faith with the objective of the thesis, a new and general mechanism of defining constraints has been implemented. A constraint is implemented as a class definition that implements the *ConstraintInterface* interface. The interface defines the only method that

has to be implemented by constraint classes, *validate (object, when)*. The parameter *object* is a reference to the object in question. The parameter *when* contains a value that specifies the current phase in the object's processing. The current phase could be *Before_Insert*, *After_Insert*, *Before_Remove*, etc. The constants are defined in the *ObjectCategory* class.

```
public void validate(Object object, int when) throws  
    ConstraintViolationException.
```

A constraint is a predicate that has to be complied with by all objects in the category. A constraint could range from a simple domain check, like the fact that an employee's salary has to be above or equal to some minimum salary, to a more complicated one, like conditional constraint. Because a constraint is a full-fledged object by itself, its definition is not limited by the expression construction rules of the DBMS. The special case of NOT NULL constraint can easily be enforced using this general mechanism. For each constraint, a class that implements the *ConstraintInterface* interface should be defined. Once it is defined, the *addConstraint* method should be invoked against the category where the constraint is to be enforced.

```
public void addConstraint (String constName, String  
    className, int when) throws  
    ConstraintNameAlreadyExistsException
```

The parameters are the constraint's name, the name of the class that defines the constraint and timing. The method raises a *ConstraintNameAlreadyExistsException* exception if the

constraint set of the category already includes a constraint by the name specified. Every time an object is added to the category, or whenever an object that is already in the category is modified or removed, the system, at run time, creates a new instance of the constraint class and calls its *validate* method. The *validate* method of a constraint object should raise a *ConstraintViolationException* to notify the system that a constraint has been violated. For instance, to enforce the constraint that an employee's salary should not be less than the minimum salary, let us say \$7.00 an hour, we first define a class that implements the *ConstraintInterface* interface.

```
public class MinSalaryConstraint implements
    ConstraintInterface {

    ...
    // implement the validate method that will
    // automatically be called by the system

    public validate(Object obj, int timing) throws
        ConstraintViolationException {
        if (((Employee)obj).getSalary() < 7.0)
            throw new ConstraintViolationException("Salary can
                not be less than seven")
        }
    }
}
```

After compiling the above class, we register the binary file with a specific category by calling the *addConstraint*. For instance, if we want to apply the constraint to all employees then we send the message to the *Employees* category as follows:

```
ObjectCategory.getCategory(db,  
    "Employees").addConstraint("MinimumSalary",  
    "MinSalaryConstraint", ObjectCategory.Before_Update)
```

The system stores the constraint's name, the constraint's class name and the timing information only. Once the constraint is added to the category, whenever needed the system dynamically creates an instance of the class `MinSalaryConstraint` and calls its *validate* method passing the object to be checked as an argument. It also traps the *ConstraintViolationException* thrown by the `MinSalaryConstraint` object, which it passes to the user's process. The same constraint can be inserted into different categories, or added to the same category with different name and timing. The only restriction is that the constraint's name within a category should be unique. The *removeConstraint* method removes a constraint from a category. If the constraint in question is not found, the method raises a *ConstraintNotFoundException* exception.

```
public void removeConstraint (String constName) throws  
    ConstraintNotFoundException
```

6.4.4 Creating Triggers

Triggers, like constraints, are objects by themselves. Therefore, they allow us to enforce integrity rules that are not automatically enforced by the system. A trigger is always associated with a constraint in a category, and it is fired whenever the constraint evaluates to true. A constraint is said to evaluate to true if it raises a

ConstraintViolationException exception. The *TriggerInterface* defines the only method that has to be implemented by trigger classes:

```
public void fire(Object obj) throws  
    ObjectCategoryException
```

The operations *addTrigger* and *removeTrigger* are used to add and remove a trigger respectively. A trigger can be associated with a constraint that has already been added to a category. Alternatively, the trigger and the constraint can be inserted to a category at the same time. The trigger execution mechanism is basically the same to the one used for constraints. However, before the trigger is fired the system makes sure that the constraint attached to it raises a *ConstraintViolationException*. Triggers within a category are uniquely identified by their name. Assuming that we have already defined a trigger class called *SetMinimumSalary*, the following instruction attaches the above-defined constraint *MinSalaryConstraint* with the *SetMinimumSalary* trigger, and registers it with the *Employees* category.

```
ObjectCategory.getCategory(db,  
    "Employees").addTrigger("RectifyMinimumSalary",  
    "SetMinimumSalary", "MinSalary",  
    ObjectCategory.Before_Update)
```

6.5 Defining Persistent Classes - OCPersistent Class

As mentioned above, OCS has no way to automatically detect when changes are made to an object. The OCPersistent class defines the attributes and methods that are required by the system for persistent object manipulation. All persistent classes must inherit from the OCPersistent class. The class defines an attribute, membership, and a set of methods.

From the user's point of view the two important methods are the no-argument constructor and the *update* method. The *update* method is used by the user to notify the system of modifications made to persistent objects. The following is a partial listing of the OCPersistent class.

```
public abstract class OCPersistent {  
    private OSVector membership  
  
    public OCPersistent()  
        this.membership = new OSVector();  
    }  
  
    public final void update () throws  
        ConstraintViolationException {  
  
        ...  
    }  
}
```

Chapter 7

7 Conclusion

OODBMSs have come a long way from a simple mechanism for the storage and retrieval of persistent objects to a DBMS with many of the facilities that are required by an OODBMS. However, their development and acceptance have been hindered by many factors. The lack of a standard data model which makes them harder to learn and use, and the fact that they are targeted towards a specific set of applications have been two of the major limitations of OODBMSs. Their target area, the “new” applications, excludes the largest users of databases: traditional business applications. However, in the long run there is a need for single database systems that satisfy all kinds of a company’s database requirements. Few applications fit neatly into just one category [1].

Current implementations of OODBMS are based on the concept of the OO data model. The OO data model is not characterized by rules that define the structure, valid operations and constraints on the context of databases. Therefore, current OODBMSs are rather characterized by the underlying OO programming language upon which their implementation is based and by the target application. A new data model is required for new kind of DBMS; A DBMS that is not application-oriented, and integrates the best features of DBMS technology with the OO concepts.

This thesis attempted to achieve the above objectives by defining a new data model, Object Category data model, that is independent of programming languages and adheres to the OO concepts almost all OODBMSs agree on. The main characteristics of the data model are:

- It introduces a new concept, category, and it distinguishes it from the concept of class. A class represents the set of all possible instances, while a category is a collection of actual instances of a class in a database. It also defines operations on categories.
- It is fully OO, because it supports the OO concepts as they are used by OO programming languages.
- It can easily be derived from the EER data model.
- It makes a clear distinction between class definition and integrity constraints. Integrity constraints, including relationships, are defined against categories.

This layered approach, separation of application aspects from database aspects, provides increased data independence and flexible design approach. The seamless integration of programming languages with DBMSs, as preferable as it is, should not come at the expense of core advantages that are usually expected from a DBMS: data independence, programming language independence, etc. Different programming languages support different type construction facilities. Therefore, the tighter a DBMS is to a programming

language the further it is from the other programming languages. Databases based on the same data model should provide the same interface independently of their implementation programming language and independently of the programming language used to develop the application. Objects in a database that is based on the OC data model are organized as members of categories. The use of category by the OC data model provides the users with a clear and consistent view of the databases. It also lends itself to the use of a query language much similar to the way SQL is used by relational systems. Categories are the only entry points to the database. Furthermore, the Category interface is provided with operations that are used for insertion, retrieval and removal of objects as well as integrity constraints.

The thesis defined a new way of executing queries that is in sink with the OO nature of the data model. It also outlined a framework for a DBMS that is based on the OC data model and addressed the issues that need to be taken into consideration for its implementation. The architecture uses the client/server paradigm, and it divides the server side of the system into two major parts: object manager and storage manager. The storage manager deals with low level disk I/O operations. The object manager is composed of different submodules that perform specific tasks like transaction, concurrency control, etc. Objects in a database based on the OC data model are uniquely identified by their identities. The object manager translates the objects' representation from in-memory address to oid and vice versa. A mechanism for a transparent access of

persistent categories as if they were in the main memory was also addressed. Two major issues were discussed: buffering mechanism and transactional access of objects in the database. The buffering mechanism deals with the transparent flow of data from persistent categories to memory based container classes and vice versa. It was also noted that although users access objects in persistent categories as if they were the only users and as if they have an exclusive access to it, objects in categories are subject to concurrent access by multiple users. As such, a concurrency control mechanism is required to prevent consistency problems. The mechanism takes into consideration the complex nature of objects and the problem of obsolete cache.

We also discussed some aspects of the ODMG data model in contrast to the OC data model. Finally we illustrated the OC data model by describing an example application that uses the OCS package. OCS is a Java implementation package of the OC data model. The package includes the `ObjectCategory` class that provides facilities to create, get and remove categories from databases. A `Category` class that implements the `CategoryInterface` interface is also provided. The interface defines the operations that allow users to insert, retrieve and delete objects. It also provides methods that are used to specify constraints, which are automatically maintained by the system. Four kinds of constraints are supported by the system. Constraints and triggers are defined as classes allowing the user to specify and enforce almost all kinds of constraints. The system also

supports the four kinds of relationships: one-to-one, one-to-many-, many-to-one, and many-to-many, and key constraints.

7.1 Future Work

OODBMS technology is a relatively new entrant to the DBMS field. As a new area, it is almost impossible to list all aspects of OODBMS that need further research. There is not even a universally agreed and uniform use of the term OODBMS. In fact, there is very little true consensus and much disagreement, even at the most basic level [8]. Therefore, we will list only those aspects that are very important to the completeness and improvement of the OC data model.

7.1.1 Query Optimization

Because of the way relationships between objects in categories are maintained, through direct pointers, and/or because different users may have different points of view when looking at a database's schema, queries can be specified in different ways. For example, to retrieve employees who work for a certain department we could send the request to:

1. Employees category. This would require a traversal through all the employees, unless the Employees category maintains an index on *department* attribute.
2. Departments category. This would require first locating the department, which is faster, then retrieving the employees following its links.

The question is how could a query optimizer transform the inefficient form of a query into the corresponding efficient one. Unlike the relational model, in which relationships are maintained through key and foreign key attribute values, relationships between objects are established and maintained through direct pointers between participating objects. The user or the DBMS can exploit this aspect of the relationships for query optimization. It is our belief that, because of the organization of objects through categories (as tables are used in relational systems) query optimization methods used by relational systems could easily be adapted and used by DBMSs based on the OC data model. In addition, prior knowledge of the schema and the way the database is organized can be used to optimize queries.

7.1.2 Object Migration

An object in OODBMSs is an instance of a certain class that is defined by the application program. An object has identity that stays the same through the object's lifetime.

However, in the real-world applications, objects do not always belong to the same group of objects. For instance, in a company's organizational hierarchy a secretary could, in the future, become a manager. OO data model does not allow such change of type. The migration of an object from one group to another without changing its identity, is an important aspect that should be supported by OODBMSs. The OC data model inherently

supports some level of object migration by virtue of separation of object definition from constraints. Most of the times object migration is required to accommodate changes on constraints that must be enforced to one group but not to others. For instance, a distinction between an employee object and a manager object might be, that the manager object should be related to a department through the attribute *manages*. The OC data model supports a limited object migration capability without type change. However, because of its importance, there is a need for an in-depth research on this area. The research could focus on object migration in general and in particular in the context of OC data model.

7.1.3 Views

Views are a very essential part of DBMSs based on relational model. Views, in addition to data independence, provide different applications with different views of the same database. The tight integration of programming languages and DBMSs, as useful and attractive as it is, raises some important questions. In particular, an application program (by definition) is intended to solve some specific problem, whereas a database (by definition) is intended to solve a variety of different problems, some of which might not even be known at the time the database is established [8]. Views are the result of requests to the database through predicate expressions and queries that are permanently remembered. They allow us to define different external views for different users, and

limit their access to the database. An enhancement to the application level interface could be to complement it with a view definition and manipulation mechanism.

Annex A

Category Specification Language (CSL)

The following is the complete list of BNF for CSL.

<category_declaration>	::=	<category_header> { <category_body> }
<category_header>	::=	category <category_name> extends <category_name_list>
<category_name_list>	::=	<category_name> [, <category_name_list>]
<category_name>	::=	<name>
<category_body>	::=	<extent_declaration> [{ <relationship_declaration> }] [{ <key_declaration> }] [<not_null_declaration>] [{ <trigger_declaration> }]
<extent_declaration>	::=	<set_declaration> <list_declaration> <array_declaration>
<set_declaration>	::=	set <class_name> [<access_mode>]
<access_mode>	::=	<index_declaration> <hash_declaration>
<index_declaration>	::=	indexedBy <index_method> (<attribute_list>)
<index_method>	::=	the name of one of the indexing methods provided by the DBMS
<hash_declaration>	::=	hashedUsing <hash_method> (<attribute_list>)

<hash_method>	::=	the name of one of the hashing functions provided by the DBMS
<list_declaration>	::=	list <class_name> [<order_declaration>]
<class_name>	::=	<name>
<order_declaration>	::=	orderBy <order_list>
<order_list>	::=	<attribute_name> [<order>] [, <order_list>]
<attribute_name>	::=	<name>
<order>	::=	ASC DESC
<array_declaration>	::=	array <class_name> <attribute_list>
<relationship_declaration>	::=	relationship <relationship_type> <related_category_name> link <attribute_name> inverse <related_attribute_name>
< relationship_type >	::=	One2One One2Many Many2One Many2Many
<related_category_name>	::=	<name>
<related_attribute_name>	::=	<attribute_name>
<key_declaration>	::=	key <attribute_list>
<attribute_list>	::=	<attribute_name> [, <attribute_list>]
<not_null_declaration>	::=	notnull <attribute_list>
<trigger_declaration>	::=	trigger <on_declaration> <trigger_name> [<when>]
<on_declaration>	::=	onInsert onRemove onModify onRetrieve

<trigger_name>	::=	<name>
<when>	::=	before after
<name>	::=	<alpha> [<string>]
<string>	::=	<character> [string]
<character>	::=	<alpha> <digit> <symbols>
<alpha>	::=	<upper_case_letter> <lowe_case_letter>
<upper_case_letter>	::=	upper case leeter from A to Z
<lower_case_letters>	::=	lower case letters from a to z
<digit>	::=	decimal digits from 0 to 9
<sysmbol>	::=	any other symbol that could be part of a name

Annex B

OCS Code Source

The ObjectCategory implements the static methods that create, retrieve and remove categories from a database. It also defines some constants that are used throughout the package.

```
package ocs.oc;

import java.util.*;
import com.odi.*;
import com.odi.util.*;
import com.odi.util.query.*;

public class ObjectCategory {

    public static final int Before_Insert = 1;
    public static final int After_Insert = 2;
    public static final int Before_Modify = 3;
    public static final int After_Modify = 4;
    public static final int Before_Remove = 5;
    public static final int After_Remove = 6;

    public static final int One2One = 1;
    public static final int One2Many = 2;
    public static final int Many2One = 3;
    public static final int Many2Many = 4;

    // the variable that contains all the categories in the database
    static Categories categories;

    // creates the root entry to the database
    static Categories initialize(Database db) {
        try {
            // gets the root object of all the categories
            categories = (Categories) db.getRoot("CategoryOfCategories");
            return categories;
        }
        catch (DatabaseRootNotFoundException e) {
            // creates the root object if it doesn't exist
        }
    }
}
```

```
        categories = new Categories ();
        db.createRoot("CategoryOfCategories". categories);
        return categories;
    }
}

    // returns all categories in the database
static Categories getCategories (Database db) {
    try {
        categories = (Categories) db.getRoot("CategoryOfCategories");
        return categories;
    }
    catch (DatabaseRootNotFoundException e) {
        return initialize(db);
    }
}

    // creates a new category and returns it
public static Category createCategory(Database db, String
categoryName, String className) throws
CategoryNameAlreadyExistsException {
    Category ct = new Category (categoryName, className);
    getCategories(db).add(ct);
    return ct;
}

public static Category createCategory(Database db, String categoryName,
String className, String superCategoriesName) throws
CategoryNameAlreadyExistsException {
    Category ct = new Category (categoryName, className,
superCategoriesName);
    getCategories(db).add(ct);
    return ct;
}

    // finds a category by name and returns a handle to it
public static Category getCategory(Database db, String categoryName)
throws CategoryNotFoundException {
    return getCategories(db).get(categoryName);
}

    // removes a category from the database
public static void deleteCategory(Database db, String categoryName)
throws ConstraintViolationException,
ReferentialIntegrityViolationException, CategoryNotFoundException {
    getCategories(db).delete(categoryName);
}
}
```

The CategoryInterface interface specifies the methods that are to be implemented by the Category class.

```
package ocs.oc;

public interface CategoryInterface {

    // Accessory methods
    String getName();
    void setName(String name);
    String getClassName();
    Class getClassType() throws ObjectCategoryException;

    // adds an object to the category
    public boolean add (Object obj)
        throws WrongObjectTypeException, ConstraintViolationException;

    // adds a collection of objects to the category
    public boolean add (Collection c) throws WrongObjectTypeException,
        ConstraintViolationException;

    // removes an object from the category
    public boolean remove (Object obj) throws
        ConstraintViolationException;

    // removes from the category, all objects in c (Difference)
    public boolean remove (Collection c) throws
        ConstraintViolationException;

    // removes all objects from the category
    public void removeAll() throws ConstraintViolationException;

    //retains only elements in c (Intersection)
    public boolean retainAll (Collection c) throws
        ConstraintViolationException;

    // retrieves the first object that satisfies the predicate
    public Object get(String predicate);

    // retrieves all the objects that satisfy the predicate
    public Set select (String predicate);

    // returns an iterator for the category
    public Iterator iterator();
}
```



```
// methods to add and remove integrity constraints

// adds a constraint
public void addConstraint (String constName, String className, int
when)
    throws ConstraintNameAlreadyExistsException;

// removes a constraint
public void removeConstraint (String constName) throws
ConstraintNotFoundException;

// adds a key
public void addKey (String keyName, String[] attributes)
    throws KeyNameAlreadyExistsException;

// removes a key
public void removeKey (String keyName) throws KeyNotFoundException;

// adds a relationship between two categories
public void addRelationship (String relationshipName, int relType,
String attrName, String catName, String invAttrName) throws
RelationshipNameAlreadyExistsException,
RelationshipAlreadyExistsException;

/* removes a relationship from a category and from the related
category */
public void removeRelationship (String relationshipName) throws
RelationshipNotFoundException;

// adds a new trigger and a new associated constraint
public void addTrigger (String triggerName, String trClassName, String
cClassName, int when) throws TriggerNameAlreadyExistsException;

/* adds a new trigger and associates it with an existing
constraint */
public void addTrigger (String triggerName, String trClassName, String
constraintName) throws ConstraintNotFoundException,
TriggerNameAlreadyExistsException;

// removes a trigger
public void removeTrigger (String triggerName) throws
TriggerNotFoundException;

// checks if the category includes the object
public boolean contains(Object obj);

// returns the number of objects in the category
public int size();
}
```

The Key class specifies the attributes that make up a key object. It uses the methods of `java.lang.reflect` to dynamically create the predicate and evaluate the query.

```
package ocs.oc;

import java.util.*;
import java.lang.reflect.*;
import com.odi.*;
import com.odi.util.query.*;

public class Key {

    private String name;
    private String[] attributes;

    // the constructor accepts the key's name and its attributes
    Key(String name, String[] attributes) {
        this.name = name;
        this.attributes = attributes;
    }

    // utility methods
    String getName() { return this.name; }
    String[] getAttributes() { return this.attributes; }

    // checks for duplicate key
    void checkKey(Category c, Object obj, int when) throws
    ConstraintViolationException, ObjectCategoryException {
        String predicate = "";
        java.lang.reflect.Field fld;
        Object atrValue;
        FreeVariables vars = new FreeVariables();
        FreeVariableBindings bindings = new FreeVariableBindings();
        for (int i=0; i<attributes.length-1; i++) {
            predicate = predicate + "(" + attributes[i] + " == " +
            "fv_" + attributes[i] + ") && ";
            try {
                fld = (c.getClassType()).getField(attributes[i]);
                vars.put("fv_" + attributes[i], fld.getType());
            }
            catch (NoSuchFieldException e) {
                throw new ObjectCategoryException("Unexpected exception in key: " +
                toString() + "\n" + e.toString());
            }
        }
        try {
```

```

        atrValue = fld.get(obj);
        bindings.put("fv_"+attributes[i], atrValue);
    }
    catch (IllegalAccessException e) {
        throw new ObjectCategoryException("Unexpected exception in key: " +
toString() + "\n" + e.toString());
    }
}

predicate = predicate + "(" + attributes[attributes.length-1] + " ==
" + "fv_" + attributes[attributes.length-1] + ")";
try {
    fld = (c.getClassType()).getField(attributes[attributes.length-1]);
    vars.put("fv_" + attributes[attributes.length-1], fld.getType());
}
catch (NoSuchFieldException e) {
    throw new ObjectCategoryException("Unexpected exception in key: " +
toString() + "\n" + e.toString());
}

try {
    atrValue = fld.get(obj);
    bindings.put("fv_" + attributes[attributes.length-1], atrValue);
}
catch (IllegalAccessException e) {
    throw new ObjectCategoryException("Unexpected exception in key: " +
toString() + "\n" + e.toString());
}

// check if there is a duplicate
Query q = new Query(c.getClassType(), predicate, vars);
Set result = q.select(c.getAll(), bindings);
boolean isUnique = true;
switch(when) {
    case ObjectCategory.Before_Insert:
        if (result.size() > 0)
            isUnique = false;
        break;
    case ObjectCategory.After_Modify:
        if (result.size() > 1)
            isUnique = false;
        break;
}
if (!isUnique) {
    String msg = "Key violation Exception in key: " + toString() + "\n
violated by: " + obj.toString();
    throw new ConstraintViolationException (msg);
}
}
}

```

A Constraint object specifies the constraint's name, the name of the class that defines it, and when to check. The Constraint class dynamically instantiates an object of type "constClassName", which implements the ConstraintInterface interface, and invokes the object's *validate* method.

```
package ocs.oc;

import com.odi.*;

public class Constraint {

    private String name;
    private String constClassName;
    private int when;

    // the constructor
    Constraint(String name, String className, int when) {
        this.name = name;
        this.constClassName = className;
        this.when = when;
    }

    // utility methods
    String getName() { return this.name; }
    int getWhen() { return this.when; }
    String getClassName() { return this.constClassName; }

    /* instantiates an object of type "constClassName" and invokes
       its validate method */
    void check(Object obj, int when) throws ConstraintViolationException,
    ObjectCategoryException {
        Class cc;
        ConstraintInterface ci;
        try{
            cc = Class.forName(this.getClassName());
        }
        catch (ClassNotFoundException e) {
            String msg = "Unexpected exception in constraint: " + toString() +
            "\n" + e.toString();
            throw new ObjectCategoryException (msg);
        }
        try {
            ci = (ConstraintInterface) cc.newInstance();
        }
    }
}
```

```
        catch (IllegalAccessException e) {
            String msg = "Unexpected exception in constraint: " + toString() +
"\n" + e.toString();
            throw new ObjectCategoryException(msg);
        }
        catch (InstantiationException e) {
            String msg = "Unexpected exception in constraint: " + toString() +
"\n" + e.toString();
            throw new ObjectCategoryException(msg);
        }
        ci.validate(obj, when);
    }
}
```

The ConstraintInterface defines the only method that is to be implemented by user defined Constraint classes.

```
package ocs.oc;

public interface ConstraintInterface {

    // the method accepts and validates the object
    public void validate(Object obj, int when) throws
ConstraintViolationException;
}
```

A Relationship object defines the attributes through which two categories are related. It also indicates the type of the relationship to be created.

```
package ocs.oc;

import com.odi.*;

public class Relationship {

    private String name;
    private int type;
    private String attributeName;
    private String relatedCategoryName;
    private String inverseAttributeName;

    // inverse relationship type
```

```
static int inverseType(int type) {
    switch (type) {
        case ObjectCategory.One2One:
            return ObjectCategory.One2One;
        case ObjectCategory.One2Many:
            return ObjectCategory.Many2One;
        case ObjectCategory.Many2One:
            return ObjectCategory.One2Many;
        case ObjectCategory.Many2Many:
            return ObjectCategory.Many2Many;
    }
}

// the constructor
Relationship(String name, int type, String atrName, String catName,
String invAtrName) {
    this.name = name;
    this.type = type;
    this.attributeName = atrName;
    this.relatedCategoryName = catName;
    this.inverseAttributeName = invAtrName;
}

// utility methods
String getName() { return this.name; }
int getType() { return this.type; }
String getAttributeName() {return this.attributeName; }
String getRelatedCategoryName() {return this.relatedCategoryName; }
String getInverseAttributeName() {return this.inverseAttributeName; }

/* the method checks the type of relationship, instantiates an
object of the RelationshipInterface type, and invokes its
validate method. */
void validateInverse(Object obj, int when) throws
ConstraintViolationException {
    RelationshipInterface ri;
    switch (type) {
        case ObjectCategory.One2One:
            ri = new Relationship121(obj, when, this);
            break;
        case ObjectCategory.One2Many:
            ri = new Relationship12M(obj, when, this);
            break;
        case ObjectCategory.Many2One:
            ri = new RelationshipM21(obj, when, this);
            break;
        case ObjectCategory.Many2Many:
            ri = new RelationshipM2M(obj, when, this);
            break;
    }
}
```

```
        ri.validateInverse();
    }
}
```

The RelationshipInterface interface defines the only method that is implemented by the different implementations.

```
package ocs.oc;

interface RelationshipInterface {

    // the method checks the referential integrity
    void validate() throws ConstraintViolationException;
}
```

A Trigger object specifies the trigger's name, the user-defined class that implements it and the Constraint object it is associated with. The class dynamically creates an instance of "className" and invokes its fire method.

```
package ocs.oc;

import java.lang.reflect.*;
import com.odi.*;

public class Trigger {

    private String name;
    private String className;
    private Constraint constraint;

    // the constructor
    Trigger(String name, String className, Constraint constraint) {
        this.name = name;
        this.className = className;
        this.constraint = constraint;
    }

    // utility methods
    String getName() { return this.name; }
    String getClassName() { return this.className; }
    Constraint getConstraint() { return this.constraint; }
}
```

```

        /* the method instantiates an object of type "className" and
        fires it */
void fire(Object obj) throws ObjectCategoryException {
    Class tc;
    TriggerInterface trigger;
    try {
        tc = Class.forName(getClassName());
    }
    catch (ClassNotFoundException e) {
        throw new ObjectCategoryException ("Unexpected exception: Class " +
getClassName() + " not found");
    }
    try {
        trigger = (TriggerInterface) tc.newInstance();
    }
    catch (IllegalAccessException e) {
        String msg = "Unexpected exception in trigger: " + name + "\n" +
e.toString();
        throw new ObjectCategoryException(msg);
    }
    catch (InstantiationException e) {
        String msg = "Unexpected exception in trigger: " + name + "\n" +
e.toString();
        throw new ObjectCategoryException(msg);
    }
    trigger.fire(obj);
}
}

```

The TriggerInterface interface defines the only method that is implemented by user-defined triggers.

```

package ocs.oc;

public interface TriggerInterface{

    // an entry method
    public void fire(Object o) throws ConstraintViolationException;
}

```


The OCPersistent class defines the attributes and methods the system needs for managing persistent objects.

```
package ocs.oc;

import java.util.*;
import com.odi.*;
import com.odi.util.*;
import com.odi.util.query.*;

public abstract class OCPersistent{

    /* the attribute contains references to the categories the object is
    member of */

    private OSVector membership;

    // initializes the OSVector
    public OCPersistent() {
        this.membership = new OSVector();
    }

    OSVector getMembership () {
        return this.membership;
    }

    void addMembership (Category ct) {
        getMembership().addElement(ct);
    }

    void removeMembership (Category ct) {
        getMembership().removeElement(ct);
    }

    int membershipNumber() {
        return getMembership().size();
    }

    /* the user calls this method after the object's contents are modified.
    It allows OCS to check for integrity constraints that are defined over
    the categories the object is member of */
    public final void update() throws ConstraintViolationException {
        OSVector m = getMembership();
        for (int i=0; i<m.size(); i++) {
            Category c = (Category) m.elementAt(i);
            c.checkObject(this, ObjectCategory.Before_Modify);
        } }
}
```

References

- [1] Andleigh, P. R. and Gretzinger, R. M. *Distributed Object-Oriented Data-Systems Design*. PTR Prentice-Hall, Inc., 1992.
- [2] Atkinson, M., DeWitt, D., Maier, D., Bancilhon, F., Dittrich, K. and Zdonik, S. *The Object-Oriented Database System Manifesto*. In proceedings to the First International Conference on Deductive and Object-Oriented Databases (DOOS89). Kim, W., Nicolas, J.-M. and Nishio S. (Editors). Kyoto, Japan, 1989.
- [3] Bancilhon, F. Delobel C. and Kanellakis P. (Eds.). *Building an Object-Oriented Database Systems. The Story of O₂*. Morgan Kaufmann Publishers, Inc. San Mateo, CA, 1992.
- [4] Bertino, E., Catania, B. and Shidlovsky, B. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers, 1997.
- [5] Cattell, R. G. G. *Object Data Management. Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, 1994.

-
- [6] Cattell, R. G. G. and Barry, K. D. (Eds.) *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., San Francisco, California, USA, 1997.
- [7] Claude, D., Christopher, L. and Philipps R. *Databases: From Relational to Object-Oriented Systems*. International Thomson Computer Press, 1995.
- [8] Date, C. J. *An Introduction to Database Systems, 6th edition*. Addison-Wesley, 1995.
- [9] Date, C. J. and Darwen H. *Foundations for Object Relational Databases. The Third Manifesto*. Addison-Wesley Longman, Inc., 1991.
- [10] Date, C. J. and Darwen, H. *A Guide To The SQL Standard, Third Edition*. Addison-Wesley Publishing Company, 1994.
- [11] Dittrich, R. K. *Object-Oriented Data Model Concepts*. Universitat Zurich, Institut fur Informatik, Zurich, Switzerland. In proceedings to Advances in Object-Oriented Database Systems. NASO Ansi Series, Spring-Verlag, 1992.

-
- [12] Dogac, A. *Advances in Object-Oriented Database Systems*. NASO Ansi Series, Spring-Verlag, 1992.
- [13] Ellis, A. M. and Stroustrup, B. AT&T Laboratories. *The Annotated C++ Reference Manual*. ANSI Base Document, Addison-Wesley Publishing Company, 1990.
- [14] Elmasri, R. and Navathe, S. B. *Fundamentals of Database Systems, second edition*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, USA. 1994.
- [15] Elmasri, R., Weeldreyer, J. and Hevner, A. *The Category Concept: An Extension to the Entity-Relationship Model*. Data and Knowledge Engineering 1, 1985.
- [16] Fowler, M. and Scott, K. *UML Distilled Second Edition. A Brief Guide to the Standard Object Modeling Language: Object Technology Series*, Addison-Wesley, 2000.
- [17] Jaworski, J. *Java 1.2 Unleashed, The Comprehensive Solution*. SAMS, USA, 1998.

-
- [18] Johnson, J. L. *Database, Models, Languages, Design*. Oxford University Press, Oxford, NY, 1997.
- [19] Joshi, I. D., Runolfson, R. and Chandak, R. *The Comprehensive Guide to the JDBC SQL API*. Ventana Communications Group, Inc. 1997
- [20] Khanna, L. *EER-Schema Mapping Tools for Interoperable Database Representations*. 1997.
- [21] Khoshafian, S. *Object-Oriented Databases*. Wiley Professional Computing, 1993.
- [22] Korth, F. H. and Silberschatz, A. University of Texas at Austin. *Database System Concepts*. McGraw-Hill Book Company, 1986.
- [23] *ObjectStore Java API User Guide*. Release 6.0. ObjectDesign Inc., March, 1999.
- [24] Petros, J. *Modeling and Implementation in an Object-Oriented Database Management System*. 1993.

-
- [25] Rao, R. B. *Object-Oriented Databases. Technology, Applications and Products*. McGraw-Hill, Inc., 1994.
- [26] Skarra, A. H. and Zdonik, B. S. *Type Evolution in Object-Oriented Database*. In *Research Foundations in Object-Oriented and Semantic Database Systems*. Cardenas, A. F. and McLeod, D. (Eds.), Prentice Hall, 1990.
- [27] Sridharan, P. *Advanced Java Networking*. Prentice Hall PTR, Upper Sadle River, New Jersey, 1997.
- [28] Tsichritzis, C. D. and Lochovsky, H. F. University of Toronto. *Database Management Systems*. Academic Press, New York, New York, 1977.
- [29] Tsichritzis, D. and Lochovsky, F. *Data Models*. Prentice-Hall, 1982.
- [30] Ullman, J. Stanford University. *Principles of Database Systems*. Computer Science Press. Rockville, Maryland, 1982.

-
- [31] Ullman, J. Stanford University. *Principles of Database and Knowledge-base Systems, Volume I*. Computer Science Press, Rockville, Maryland, 1988.
- [32] Yang, C. C. *Relational Databases*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1986.
- [33] Yannakoudakis, J. E. and Chang, P. C. *Standard Relational and Network Database Languages*. Springer-Verlag, 1988.