1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

**A FRAMEWORK FOR CHARACTERIZING
AN OBJECT-ORIENTED DATABASE SYSTEM**

by

JIAN JIN, B.S., M. Eng.

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirement for the degree of**

Master of Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

February 14, 1992

copyright

1992, Jian Jin

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

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

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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

ISBN 0-315-75986-0

Canada

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

**A Framework for Characterizing An Object-Oriented Database System
submitted by Jian Jin, B.S., M.Eng.**

Thesis Supervisor

Director, School of Computer Science

Carleton University

Date: 14 Feb 92

Abstract

As object-oriented database system moving from research laboratory to commercial market, it is important to have a precise definition for it. In this thesis we provide characteristics of object-oriented data base systems at two levels. Our goal is two folded. First, by giving definitions for an object-oriented database system and an object-oriented data model we attempt to ameliorate the conceptual confusions in the field and clarifying some relevant issues. Second, by present two evolution paths of database systems and a unified view of database systems, we provide a platform for the development and implementation of future general-purpose database systems. Finally, we implement a restricted prototype of our object-oriented data model.

Acknowledgement

I would like to thank Prof. Ekow Otoo for his nice supervising during the development of this thesis.

This thesis was partially supported from EMR/NSERC grant.

Tabel of Contents

	Page
Acceptance Sheet	ii
Abstract	iii
Acknowledgements	iv
Table of Contents	v
Chapter 1 Introduction	1
1.1 The Need for Object-Oriented Database Systems	1
1.2 What Is Lacking	3
1.3 Objectives	5
1.4 Related Works	7
Chapter 2 Current Status and Systems Survey	10
Chapter 3 Definitions of Object-Oriented Database Systems	16
3.1 The Characteristics of An Object-Oriented Database System	16
3.2 Fundamental Principles of Object-Oriented Paradigm	17
3.3 Basic Features of Database Systems	23
3.4 Unique Features of Object-Oriented Database Systems	31
3.5 Advanced Features for New Applications	36
3.6 The Definitions	39
3.7 Differences from Previous Database systems	40
Chapter 4 An Object-Oriented Data Model	43
4.1 What is It	43
4.2 Structural Part	45
4.3 Operational Part	64

Chapter 5	An Unified View of Database Systems	81
5.1	Evolution of Database Systems	81
5.2	From Relations to Objects	82
5.3	From Network Model to Object-Oriented Model	85
5.4	Further Discussions	89
5.5	A Unified View	93
Chapter 6	The Simulation	95
6.1	System Architecture	95
6.2	Metadata Mechanism	97
6.3	Object Management	101
6.4	An Example	105
6.5	Remarks	111
Chapter 7	Conclusions	112
References		115
Appendix	C++ Source Code For The Simulation	123

Chapter 1. Introduction

1.1 THE NEED FOR OBJECT-ORIENTED DATABASE SYSTEMS

During the past several years the object-oriented paradigm has generated particular interest in the database community and the development of new generation of database systems, object-oriented database systems, has emerged. Roughly object-oriented database systems refer to database systems which support the object-oriented concepts, i.e., abstract data type, complex objects, encapsulation, etc. Generally the object-oriented database systems have been approached with two major intentions. One is to meet the requirements of new applications and another is to ameliorate the impedance mismatch problem between host languages and data definition languages (DDL) and data manipulation languages (DML).

1.1.1 New Applications

Traditional (relational) database systems are considered inappropriate for new database applications including computer-aided design/computer-aided manufacture(CAD/CAM)[Kemp 87] [Rowe 87], office information systems [Bane 87a] [Lyng 86] and computer aided software engineering (CASE) [Died 87]. Some disadvantages of conventional (relational) systems are listed as follows.

- Artificial representation of complex structured objects:* The relational data model fails to support the modelling of complex structured objects since it only allows the user to model objects by flat relations. Also because relations are all at the same level, other meaningful relationships are either lost or have to be maintained entirely by the application programs.

- No extensibility:* To allow user to model the complex objects naturally and efficiently, additional types should be definable by users. However the

data types in a conventional database system are fixed and it is not allowed to define new data types.

•*Missing operational semantics of complex objects:* Relational data modelling techniques fail to capture the operational semantics (i.e., set of operations applicable to an object) that are normally associated with complex objects. Instead, those semantics are completely specified by users in their application programs.

The limited modeling power leads to unnatural representation and poor performance. For example, the pattern of data access in CAD is different from that of commercial databases. Typically, a design object, such as an integrated circuit (IC), is composed of thousands of design elements. Design elements may have connections with external items, such as physical layout, component specifications or analytical data. When the designer looks at IC, all the related elements and many of the associated ones need to be quickly loaded from the database as a cluster. This combination of aggregate access and complex internal and external structure cannot be efficiently implemented in conventional database systems. However, in object-oriented systems, by providing rich data modeling constructs and raising the level of abstraction, users are able to deal with objects (and operations) that more closely resemble their counterparts in the real world. Thus with information of high data complexity and multiple inter-relationships, object-oriented database system may prove to have better performance than relational database systems. As pointed in [Stone 90], clustering and cache techniques for certain complex structures in object-oriented database systems can give them a performance two orders of magnitude greater than that of relational database systems.

1.1.2 Impedance Mismatch

A traditional database system supports a data model which is different from other parts of the programming environment. For example, a relational or CODASYL model interacts with the rest of the environment in a limited way, embedding DDL and DML in the host language, rather than through a

general purpose programming language. These characteristics are perceived as impediments to the productivity and system performance, since a great deal of explicit data retrieval and translation must be done to get data from one environment to the another. Moreover, it requires a programmer to be multilingual requiring knowledge of a host language, an embedded query language, and a data definition language. So it is desired that a database should be integrated with the rest of the programming environment as seamless as possible [Ford 88]. A truly seamless integration of computational and data storage environments requires that both the database system and the application use the same language and data model. This means that the same data types should be manipulated by the same operators. Object-oriented database systems try to ameliorate the impedance mismatch by extending the DML so that more of an application can be written in the DML. Although few current systems can express a complete application by themselves, with more of the application semantic information in database, the interface between database and general-purpose language is crossed less frequently.

1.2 WHAT IS LACKING

The explosion of the interest in object-oriented database system has led to various definitions and interpretations of this much-used and much-abused term, "object-oriented database". The definition of an object-oriented database system is subject to considerable debate and it will probably take a long time before the community will agree on a common definition. Rentsch [Rent 82] described the field of object-oriented programming languages as follows:

"...object oriented programming will be in the 1980's what structured programming was in 1970's. Everyone will be in the favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know what it is".

Although this statement is aimed at object-oriented programming

languages, its message is even more accurate for object-oriented database systems. As object-oriented database systems move from laboratory into the world of applications, there is great need to distinguish between large amount of database systems that call themselves "object-oriented". For example, some systems are obtained by extending object-oriented programming languages with persistent data while other developers add the facilities for handling complex objects on the top of relational systems. Also there are systems based on their own created object-oriented data models. As a consequence, it can be very difficult for a newcomer to understand and evaluate what is meant by claiming that a database system is "object-oriented".

There are several reasons for the confusions of the definition and interpretation of object-oriented database systems. First, there is no single standard object-oriented data model and there is no equivalent of relational algebra for object-oriented data model. Therefore, there are no standard guidelines for designing object-oriented database systems. Researchers from different background have taken different views, such as, Gemstone [Maie 86] adopts the object/message paradigm, Vbase [Andr 87] utilizes the abstract data type paradigm, IRIS [Fish 87] applies functional approach and POSTGRES [Ston 86] extends the relational data model to support nested objects. This is to be expected. The object-oriented approach stems from the programming language area. Many object-oriented concepts have been popularized by Smalltalk [Gold 83] efforts. However, we have no corresponding agreement as to what mechanism belongs to an object-oriented programming language, or what rules for encapsulation and inheritance should be adhered to.

Second, an existing object-oriented database system usually aims at one specific application. For example, the majority of the object-oriented database systems are not commercial products, but proprietary systems built into CAD systems. So they are characterized by their system architecture and external functionality. As a consequence, existing systems which are claimed to be object-oriented vary greatly from one to another.

Third, some fundamental object-oriented concepts are unique to object-oriented paradigm. They overlap with other data models. For instance,

King [King 89] has pointed out that certain systems that have been advertised as being object-oriented systems would fit in semantic modeling. In [Ullm 89] the object-oriented systems are those systems which support the object identity. Under this definition the network and hierarchical model are object-oriented too.

Finally, the evolution of object-oriented database systems has followed quite a different path from that of relational systems. In the mid-seventies, when relational systems attracted people's attention, a formal model had been already established. People were mainly developing implementation technology to realize the common underlying model. In contrast, the development of object-oriented database systems seems to be driven by the needs of new applications and their implementation are not based on a precise formal model. Moreover, various models are being developed, in parallel, or even following the implementation effort. Choosing the specification of the system and producing the technology to support its implementation are simultaneously pursued. This is the reason for the current diversity in object-oriented database systems.

1.3. OBJECTIVES

In this thesis, we attempt to contribute toward the characterization of an object-oriented database system. We catalog several different ways in which a database system can be viewed. Generally, there are three levels of abstraction used in describing database systems [Ullm 89] [Elma 89]. A fairly standard view point regarding the abstraction is shown in Fig.1.1.

External view is a high level abstraction of a database system. The essential features and functionalities of a database system are shown at this level. Also it provides the description of the part of the database that a particular user group is interested in and hides the rest of the database from that group. External view of a database system may be specified by a list of features and a high level data model.

Fig 1.1 Three levels of abstraction in a database system

Conceptual level is a global description of a database system. It concentrates on describing entities, data types, relationships and operations. A conceptual database is described in terms of a data model. For example, in the relational data model, data is seen as tables whose columns are headed by attributes and whose rows are tuples. The data model provides a unified and consistent way to represent the information in applications.

Physical level of a database system deals with the implementation issue which describes the complete details of data storage and access paths for the database. Also it specifies the implementation strategies for realizing features and functionalities of the database system. Since what we are interested in is the high-level characteristics of object-oriented database system, we emphasize the external view and the conceptual level. Ideally, the implementation technique should be chosen and developed after the specification of the system and formal model are established. Thus the discussion on high-level characteristics should not be interfered with the implementation issues. However, because any model eventually must be supported by certain implementation technique, we will still give some discussion on the implementation issues when it is relevant.

Since the object-oriented database technology is still in a premature stage, it is not our intention to propose a final definition. Also we do not claim that our model is a standard object-oriented data model. Our goal is to provide a platform for developing and implementing future object-oriented database systems. As a consequence, we present two definitions for object-oriented database systems. The first one is a basic definition and may be considered as a threshold specification for object-oriented database systems. The second definition which includes additional features that might not be supported by existing object-oriented database systems. While our object-oriented data model emphasizes and explores issues that are

ignored in the literature. Additionally, we believe that the object-oriented database technology does not only introduce another type of database systems but also is the direction of evolution of the database technology. It is our opinion that object-oriented database system is the result of evolution of the traditional database systems and may be considered as a unified framework for future database systems.

This thesis is organized around several themes: definitions of object-oriented database systems, an object-oriented data model, a unified view of database systems and a simulation. Our contribution has two aspects. First, by giving definitions for an object-oriented database system and an object-oriented data model we attempt to ameliorate the conceptual confusions in the field and clarifying some relevant issues. Second, by present a unified view of database systems, we provide a platform for the development and implementation of future database systems.

1.4. RELATED WORKS

The lack of formal definition of object-oriented database systems has attracted more and more attention in the database community and a number of research efforts have begun to address the problem. We briefly survey some related works.

Related to the definition of object-oriented database systems, two of the most significant papers are [Atki 89] and [Ston 90]. [Atki 89] authored by a distinguished group of object-oriented database system researchers, lists a series of rules that a database system can satisfy in order to be considered as an object-oriented database system. However, the rules are not the equivalence of Codd's rules for relational database systems. Stonebraker et al. [Ston 90] suggest a different set of tenets which are contrasted with [Atki 89]. The paper argues that new generation of database systems should support data, rule and object management with a complete toolkit, including integration of the DBMS and its query language into a multilingual environment. It decrees that new generation of database systems should be an extension of relational systems. The theme in this

paper is not particularly around the object-oriented database system. Our definitions in Chapter 3 are different from these two in the following aspects. First, instead of postulating a highly rigorous set of rules, we propose a broad set of requirements of object-oriented database system. Our requirements include some new features that are particularly needed by new applications. Also we propose a set of unique features for object-oriented database systems. Secondly, in addition to listing the requirements, relevant issues around these requirements are discussed too. Third, a comparison of object-oriented database system and previous systems is presented.

At the data model level, various approaches are adopted in trying to propose a formal object-oriented data model. However the data models proposed are usually based on some specific systems. Kim [Kim90] defined and justified what an object-oriented model should be, on the basis of a small set of central object-oriented concepts. The paper is based on the research results obtained from the ORION object-oriented database system [Kim 89a] prototyped in MCC. Lecluse et al. [Lecl 88] propose the data model foundation for an object-oriented database system. The originality of this model, called O₂, is a type system defined in the framework of a set and type data model. Hong and Maryanski [Hong 90] use a meta-model to represent object-oriented data model. The object-oriented data model presented in Chapter 4 consolidates and extends the ideas and approaches found in the literature. Our contribution is two folded. One is formalizing some basic concepts in a set of definitions. Another is introducing new ideas and approaches to cover some shortcomings of existing object-oriented data models.

[Beec 88] is related to our work in Chapter 5. In this paper, correspondence between relations and objects was observed. In order to convince the reader that object-oriented model is a result of evolution of conventional data models, the discussion in Chapter 5 also includes some models sitting between object-oriented model and relational model, such as, nested relations and semantic model. Moreover, the evolution path that started from network model is examined too.

This thesis is organized as following. First in Chapter 3 we present a set

of characteristics that can be considered as the specification of object-oriented database systems. An object-oriented database system must first of all be a database system, that is, it must provide the features that are found in a database system. We also require that some basic object-oriented features must be present in order for a database system to call itself object-oriented. The specification also includes some features unique to object-oriented database system as well as some functionalities that are not well supported in traditional database systems. Then in Chapter 4 an object-oriented data model at the conceptual level is described. A data model specifies the structure of the data in a database and a set of operations for retrieving and updating the data. A set of definitions are presented to formalize relevant basic concepts. New ideas and approaches are proposed to deal with some difficult issues and make the data model more powerful. In Chapter 5 we describe the evolution paths of database systems and reinterpret different concepts and ideas to provide a unified view of database systems. In Chapter 6 the simulation of the core of our data model is described. Some conclusions are given in Chapter 7.

Chapter 2. Current Status And Systems Survey

In this chapter, we describe several object-oriented database systems. Instead of completely describing them, we focus on their unique features and the differences among them. It is our intention to give a rough picture of what current object-oriented database systems look like and provide a base for our further discussion.

Iris [Fish 87]

The Iris database management system developed at Hewelett-Packard Laboratories is intended to meet the need of new database applications. In addition to the basic requirements for permanence of data, Iris provides new capabilities including rich data modeling constructs, inference, multimedia data (graphic images, voice, text, vectors, and matrices), long transactions, and versions. It is also designed to support multiple programming languages.

The Iris storage manager is a conventional relational storage subsystem. It provides associative access and update capabilities including transaction support. The query processor translates queries and functions into an extended relational algebra format that is interpreted against the stored database.

Object SQL (OSQL) which is an object-oriented extension to SQL is supported as interactive interface for querying. A graphical editor which allows user to explore the metadata structure as well as the stored data is also provided.

One unique feature of Iris is that it allows several formats of function implementation. Functions are defined on types and are applicable to the instances of types. A stored function is a table mapping input values to their corresponding result values. The derived functions may be specified in terms of other functions. The third kind of functions which are

implemented in some programming languages, such as C, is called foreign functions. Foreign functions are outside the control of the database system. Information about objects is modeled using predicate functions, instead of directly using attributes. The attribute concept is modeled by using functions whose values are derived from the predicates. This approach totally respects the concept of encapsulation.

Gemstone [Bret 89].

Gemstone is a commercial product developed by Servis Logic Corporation. It is based on Smalltalk with extensions of database features. It is disk-based and provides concurrency control, recovery, constraints, queries, index support and authorization. Gemstone incorporates object identity and encapsulation. The data abstraction is obtained by providing an external interface of objects as a set of messages.

Data definition and manipulation as well as queries are executed via a single database programming language, OPAL. Queries are formed over the instance variables of an object. This violates the principle of encapsulation since it should not be possible to write any external programs that depend on the representation of an object. Indices are also maintained on instance variables. However, since the presence of indices does not need to show through the interface it is not in contrast to the encapsulation. Indexing also may be installed on arbitrary collection of objects while most systems indexes on classes(types). Secondary storage management mechanism is provided for storing both large number of objects and objects that are large in size.

O2 [Banc 88]

The motivation behind the development of O2 database system is to prototype database systems of the 1990's. Researchers of Altair want to build a complete development environment for data intensive applications. Here we describe some unique properties of O2 system.

O2 makes a distinction between values and objects. Objects have identities and can be referenced by other objects, while values do not. A value can be embedded in an object or in another value but they have no outside references. In O2 a type describes a *minimal* behavior for an object. An object that belongs to a type must support all behavior defined in the type but might have more. Also classes and methods in O2 are not objects. This is usually different from most object-oriented systems in which everything is object.

Objects in O2 cannot be deleted explicitly, but references can be deleted. When there is no references on an object, the object is collected by garbage collector. This method is usually used in programming languages and is unusual for database systems. The advantage of this strategy is that the database is guaranteed to have no dangling pointers.

Vbase [Andr 87]

Vbase is a commercial product of Ontologic Inc. The data model of Vbase is influenced by abstract data type language CLU [Lisk 81]. The object behavior is elicited by a combination of properties and operations. The properties express the abstract state of an object and does not imply anything about the representation.

Strong type is evident through the system. This is in contrast to the Smalltalk-based systems. Vbase attempts to do all type checking at compile time but if it can not, type checking is deferred to run-time.

Vbase follows a layered approach to the system structure. A flexible architecture is achieved with a clear separation of specification and implementation. The language layer provides for the specification of objects. The abstraction layer provides essential modeling features of the object-oriented approach. The representation layer provides the semantics of the mapping to stored objects. The stored layer is responsible for object persistence.

Vbase provides *database variables* (names) for objects of any type. So objects with names can be accessed directly by their names. In most object-oriented systems an operation is always associated with a certain type(class). However, Vbase also supports operations that are not associated with any type.

Type definition language (TDL) is the data definition language for Vbase. TDL is used for creating new types as well as new objects of existing types, and for declaring database variables and constraints. The COP language, which is a strict superset of C, is used for two purposes. First, it is used to build methods that are executed within the database; second, it is used to build application modules that will access the database. Object SQL is another DML for Vbase.

OZ+ [Weis 89]

OZ+ developed at the University of Toronto is used as the underlying layer of office information system (OIS). It is designed to fuse the concepts of object-oriented programming languages with those of database systems and produce a programming environment that would be well-suited to the need of OIS implementors.

From the object-oriented system perspective, OZ+ supports complex objects. A complex object may organize the objects it contains in number of ways. In the *simple aggregation* (unordered set) the complex object is the parent of a set of sub-objects. A complex object also can be specified in *array type* (ordered set) where it is represented as an array of sub-objects or *tree type* where it is represented as a m-ary tree structure ordering of sub-objects. Sub-objects may themselves be complex objects.

Operations on objects in OZ+ are referred as rules which may be invoked by a message sent to it from a rule of another object. A rule invoked may return a value to the message sender. OZ+ adopts *blocking/ send/ receive/ reply* message protocol. In this protocol, the sending object is blocked until its message is both received and replied.

One interesting thing about OZ+ is that a message is itself an object. Message objects are referred as temporary objects, since the message objects exist only from the time they are invoked until the end of their execution. Self-trigger rules embedded in objects are used to model the active properties of objects. When the state of an object is changed, the triggers in the object are tested to determine whether any of them are executable.

OZ+ utilizes an existing relational storage subsystem. Therefore, algorithms are needed to encode classes to relations and objects into tuples, as well as to perform the inverse operations. Additional to the burden of data transformation, a negative aspect to this approach is that it does not provide the flexibility of clustering of related objects because the storage schema is strictly relational.

Orion [Bane 88]

Orion developed at MCC is also a Smalltalk derivative. Orion has concentrated on the advanced database features which are needed by new applications. Functions supported by Orion include versions and change notification, complex objects, dynamic schema evolution, multimedia data management, etc.

The formal framework for schema changes consists of a set of properties of the schema. These properties are called *invariant*. The invariant must hold at every state of schema, so it is guaranteed that the change of schema does not leave the schema in an inconsistent state. A set of rules are established for preserving the invariant. When the schema invariant can be preserved in more than one way, the rules also guide the selection of one most meaningful way

Orion supports automatic type extents. That is for any class *C* the system maintains the set of all instances of *C*. GemStone does not support automatic extents and O2 supports it optional. As a consequence, there is a class *SetofC* for every class *C* in the class hierarchy. If *S* is subclass of *C*

then **SetOfS** is subclass of **SetOfC**. This creates a parallel hierarchy of **Set** classes. One related question is that since **SetOfC** is a class too then it seems reasonable to create class **SetOfSetOfC** , **SetOfSetOfSetOfC**, and so on. **Vbase** deals with typed set by a **Set[T]** parameterized type. **Gemstone** puts constraints on constructing typed set so that all members in the set belong to a given class.

Chapter 3. Definitions of Object-Oriented Database Systems

3.1 THE CHARACTERISTICS OF OBJECT-ORIENTED DATABASE SYSTEMS

Historically object-oriented database systems stem from the development of semantic data model [Hamm 81, Smit 77], and development of object-oriented programming languages. The Frame model [Mins 75] for knowledge representation in artificial intelligence, may also have influenced object-oriented database technology.

According to these roots two main approaches can be observed in the development of object-oriented database systems. Researchers from database community try to adjust conventional database technology to become object-oriented by supporting the concept of complex objects. At the lowest level, an object manager, which allows the system to handle complex objects, was built. Indexing and access methods installed for complex objects might be added to efficiently access objects. At the interface of the system, operations for objects are offered. As a result of these activities, the system provides a data model that allows users to represent one entity in application domain by exactly one object in database and offers appropriate operations to manipulate the structure of objects. However, concepts known from object-oriented programming languages such as encapsulation and message passing are disregarded. Since the objects stored have no behavior aspects, such systems are called structural object-oriented [Ditt 88a].

Another approach is followed by the programming language community. They extend object-oriented programming languages with database technology such as persistence and object sharing. The advantage of this approach is a unique data model for both, persistent and non-persistent data, i.e., the type of the data is not related to its persistent property [Atk 87]. The disadvantage is the missing realization of most other database concepts, such as indexing, concurrency control, recovery,

and declarative query facility.

Given the two approaches above, it seems reasonable to argue that a true object-oriented database system should be a combination of the essential concepts of these two. Thus, providing the basic features of a database system and following the object-oriented paradigm are considered as two major characteristics of an object-oriented database system. However, there might be some other features that are unique to an object-oriented database system because these features belong to neither database systems or object-oriented paradigm. Some of them, such as active objects and triggers, are the outcomes of combining database systems and object-oriented paradigm. Others, such as dynamic type creation and behavior management, are required by the system which is both a database system and an object-oriented system. As a new generation database system required by new applications, an object-oriented database system should support some advanced features including versions, schema evolution and multilingualism.

In the following sections we first present the fundamental principle of object-oriented paradigm in 3.2. Then in 3.3 we present a set of basic features of a database system. The unique features of an object-oriented database system and some advanced features for new applications are described in 3.4 and 3.5 respectively. We give two definitions of object-oriented database systems in 3.6. The basic definition may be used to determine whether or not a given system can be considered as an object-oriented database system. Since the extended definition is part of the platform of future database systems, there might be no existing object-oriented database systems that fit our definition. Finally, a comparison of object-oriented database system and previous database systems is given in 3.7.

3.2 FUNDAMENTAL PRINCIPLES OF OBJECT-ORIENTED SYSTEMS

3.2.1 Historical Roots

Object-oriented paradigm has evolved in three different disciplines: first in

programming language, then in artificial intelligence and database. The first appearance of the notion of “object” was in Simula, a language for programming simulation [Birt 73]. So Simula is generally regarded as the first object-oriented programming language. As surveyed in [Mica 88], since Simula, two paths have been taken to develop object-oriented programming languages. One is the development of new object-oriented languages, notably Smalltalk [Gold 83], Eiffel [Meye 85] and Trellis/Owl [Scha 86]. Another path is to extend conventional languages: C++ [Stro 86] and Objective C [Cox 84] as extension of C, Common LOOPS [Bobr 86] and Flavors [Moon 86] as extension of Lisp, Clascal [Schm 86] as an extension of Pascal, and so on. Since object-oriented concepts has been populated mainly through Smalltalk effort, both object class and class inheritance become most important concepts in object-oriented paradigm.

On the other hand, researchers in artificial intelligence have developed Frame [Mins 75] knowledge representation schema which has also influenced the representation of complex objects. In the database area, research activities in semantic data model have introduced similar object-oriented concepts that are embedded in programming language and knowledge representation. For example, in semantic data model the concept of class captures instance-of relationship, generalization captures IS-A relationships and aggregation captures IS-PART-OF relationships. Some well-known semantic data models include DAPLEX [Ship 81], E/R [Chen 76] and SDM [Hamm 81].

In the following sections we describe the fundamental concepts that form object-oriented paradigm. Our discussion is oriented around object-oriented database systems.

3.2.2 Objects

Object modeling provides a natural and straightforward way to represent the entities in real-world. In object-oriented paradigm, all conceptual or physical entities that appear in the application domain are modeled as objects. Every object is associated with an unique object identifier. Objects

are described by their internal states and behavior. The internal state of an object is usually captured in a collection of attributes (instance variables). The behavior of an object is specified in a set of operations (methods) which determine what the object will do when it receive a message.

3.2.3 Message Passing

In object-oriented paradigm objects communicate with one another by message passing. An object, called sender, sends a message to another object, called receiver. When the message is received by the receiver, a corresponding method is invoked and the result of the operation may be returned to the sender. Notice that during the execution of the invoked method, the receiver may send messages to other objects. Thus a message sending chain might occur.

Message passing approach has certain nice features [Agha 86]. First of all, information of whole system is distributed among the objects of the system. Each object has only the information about itself and the knowledge required for it to respond to messages from other objects. Second, all communication and interaction between objects are accomplished by message passing. No object can be operated, looked at or modified except by sending a request to the object to perform its methods. Third, by using class-inheritance mechanism a group of objects are able to share common knowledge or ability to avoid duplication. Finally, message passing is an appropriate way to think about parallel computation. Since many objects may be actively sending, receiving or processing messages at the same time this approach is inherently well suited to modeling parallel systems.

An object can send messages to any object as long as it has the reference, name or Oid, of the target object. In our discussion, we use C++ syntax to represent message sending event:

receiver. selector (arg₁, arg₂,...)

where **selector** is the name of the message, it is usually identical to the

name of corresponding methods. **Receiver** is the object to which the message is sent. The optional arguments. **Arg1**, **arg2**, ... are objects or they may be message sending expressions themselves.

3.2.4 Encapsulation

The concept of encapsulation comes from abstract data type systems. Encapsulation is essentially minimizing inter-dependencies between objects. From the software engineering point of view, this facilitates easy system modification because as long as the interface of an object remains unchanged, the implementation can be changed without affecting its use. On the other hand, the clients outside of an object need to know only the interface and do not need to concern the implementation or representation of the object.

In an object-oriented system the unit of modularity is object. So an object is the subject of encapsulation. An object has interface part and implementation part. The specification of operations that can be performed on the object forms the interface. The data structure, which represents the state of the object, and procedure, which implements the operations, form the implementation part. The object can only be accessed through the interface by invoking its methods, because its internal state can only be manipulated by its own methods. Thus the object's realization(values of its state and implementation of methods) are encapsulated.

3.2.5 Class and Type

Objects are classified into classes. Objects of a certain class are called instances of the class. The notion of class covers two aspects: an *object template* and an *object container* [Banc 88]. Object template means that the structure and behavior of an object is defined by its class. That is, a class specifies a set of methods and a set of instance variables for its instances. Thus all the instances that belong to the same class have the same structure and share the common methods. Object container means that all instances

of a class are collected in one container, called the extension of the class.

Some systems support the notion of type instead of class. The concept of type originally came from abstract data type in programming languages, where it is a tool to check program correctness. A type, in an object-oriented system, summarizes the common features of a set of objects with the same characteristics. Thus types reflect the template aspect of classes.

In most systems classes are themselves objects and they have a set of methods, called class methods, for their own use. For example, all classes should have a method, say `initialize`, which is used to create a new instance of the class.

Since using extension of class to group objects has certain restrictions (we discuss this in Chapter 4), we introduce another concept, object-set, as container of objects. Therefore in our discussion class only has the template aspect and it has the same meaning of type.

3.2.5 Inheritance and Delegation

Mechanisms for sharing knowledge and behavior between objects are among the most useful features of object-oriented idea. All object-oriented systems have the ability to share data, code and definitions, i.e., they provide some way to define a new object in terms of existing objects, borrow implementation as well as behavior description from previously defined objects. Researchers have been debating in the relative merits of various sharing mechanisms, such as, *class and inheritance* as opposed to *prototype and delegation* [Lieb 86].

Many object-oriented languages and almost all existing object-oriented databases implement the sharing ability through class & inheritance concepts. In an object-oriented system, objects of a certain group may have the same behavior in some aspects. Hence, when the system is implemented certain part of the codes can be shared or reused by these objects and this kind of sharing mechanism is called inheritance. Inheritance can simplify a large piece of software by exploiting the similarities between certain

objects. The key idea of the inheritance is to provide a simple and powerful mechanism for defining new classes that inherit properties from the existing classes. One interesting point is that since the subclass may be allowed to access attributes of its superclass, the encapsulation of the superclass is effectively violated [Synd 86]. *Single inheritance* means a subclass may inherit instance variables and methods of a single superclass, while *multiple inheritance* allows a subclass to inherit from more than one superclasses. Although multiple inheritance is not frequently required sometimes it can be quite clumsy to make do without it. With multiple inheritance we also need to handle the name conflicts [Bene 87a].

Although the class and inheritance approach is generally satisfactory, there are many applications such as office automation, CAD/CAM where the strict uniformity of objects in a class is undesirable. Some recent research has attempted to solve this problem by advocating models based on prototypes. In the prototype and delegation model, objects are not considered to belong to classes. Each object defines its own variables and methods. Any objects can behave as a prototype for the creation of a new object. Delegation was proposed as a direct alternative to inheritance. Formally delegation is a mechanism that allows objects to delegate responsibility for performing an operation, or finding a value, to one or more designated "ancestors". The new object can delegate operations and data values back to the originating object (its proxy) or locally can create new operations and data values. In fact, both delegation and inheritance strategies are used in human organization. Inheritance is used when a person refers to a book or asks someone else how to perform some tasks. Delegation is the familiar strategy of getting someone to do the work.

In addition to the flexible modeling capability, an advantage of the prototype-based approach over the class-based approach is that it permit dynamic and per-object determination of how to use the delegation protocol, i.e., the "object hierarchies" can be dynamically changed. Two concepts missing from prototype-based systems are class and class hierarchy. In the absence of the class concept, the name and the specification of the attributes must be replicated in every object. This seriously affects the storage space and run time efficiency. Also there is no

way to construct the schema. For these reasons, no database system is based on prototype model.

3.2.7 Polymorphism and Overloading

A polymorphism function is the one that can be applied uniformly to a variety of objects, that is, several operations share a name but have completely different behavior depending on the type of object. For example, operation `open` may apply to both files and windows. If the object is a file, `open` loads the file from a disk to the memory. If the object is a window, `open` displays it on the screen. In an object-oriented system this polymorphism is realized by defining `open` operation at a higher level (say, since classes `File` and `Window` are both subclasses of class `Object`, we define `open` as an operation of `Object`). However we redefine the implementation of `open` for each of class `File` and class `Window`. Thus, `open` has a single name and but denote two different implementations (overloading).

An obvious advantage of polymorphism is that the application programmer does not need to worry about the type of variables, because in some situations the type of a variable is unknown at compile-time. If a variable can be dynamically bound to objects of different classes, some form of run-time method look up must be performed (late binding).

Polymorphism also enhances the reusability and make the system more maintainable, since the current program not only works for objects of existing type, but also works for objects of newly defined types without modification.

3.3 BASIC FEATURES OF DATABASE SYSTEMS

In this section, we take a look at some basic features of database systems. Our discussion focuses on the issues which might arise when these features

are integrated into an object-oriented database system. Many of these issues are still open questions and further research efforts are needed to provide satisfactory solutions. Since most of these issues are related to the implementation techniques, they are out of our context. Thus our discussion is brief and questions are proposed without providing their solutions.

3.3.1 Persistence

As repository for stored data, a database should be persistent. By persistent we mean that the data in a database are still accessible after the process of creating the data has terminated. In programming languages, data live as long as they are referenced, otherwise, they may be removed by the garbage collector. This is no longer applicable in the presence of persistence. Also since memory is highly volatile because it is susceptible to processor failure and system shut down, the persistent data have to be stored on secondary storage.

In object-oriented database systems, persistence should be *orthogonal* and *implicit* [Atki 83]. Orthogonal means that all objects, whatever their type is, should be allowed to become persistent or volatile. Implicit means that the user should not have to explicitly make objects persistent. Location and manipulation of persistent objects should be similar to the manipulation of volatile objects. For example, it should be possible to move objects from persistent store to volatile store in the same way as to move objects from the stack to the heap. However, these two properties are not easy to achieve. One solution is to define the persistent property of an object by using operations which make the object persistent. This obviously violates the principle of implicitness. Another simple solution is to make every thing persistent. Even intermediate results as well as dummy variables become persistent. The drawback of this method, inefficiency, is obvious. A more sophisticated approach is to let the system designer specify which classes are to be persistent or only store the object which has unique name, as in O2 which allows a class to be defined as persistent or non-persistent. Also a non-persistent object can become persistent by giving it a unique

name. It is still a challenge to future research work to find more appropriate approaches.

3.3.2 Sharing and Recovery

Both concurrency and recovery mechanisms are tightly related to the concept of *transaction management*, since a transaction is usually viewed as both the unit of concurrency and the unit of recovery. Concurrent transactions have to be controlled so that the results of some serial execution of transactions has to be equivalent to the results of an interleaved execution of these transactions. For the recovery, in the case of system failure, all updates of a committed transaction will be reflected in the state of the database, and all updates of an aborted transaction will be purged from the database.

Let us look at some issues on applying transaction concepts on object-oriented database systems. Traditionally, the notion of a transaction is associated with four properties [Hard 83]: *atomicity, consistency, isolation, and durability* (ACID principles). This concept of transaction processing works well in conventional database applications where transactions are rather short. However, in some non-conventional applications, objects often need to be processed for a long period of time, such as in a design procedure. A long transaction needs a finer unit of recovery since it is not reasonable for a user to loose his work of days even weeks when the system crashes. So we need a new interpretation of atomicity. Moreover, some applications require work to be done cooperatively. If the isolation of transaction is strictly followed, the cooperative manner can not be supported. There are some approaches which overcome these restrictions of conventional transaction concept. By allowing transactions to exchange objects, to let other transactions see intermediate results [Klah 85], cooperative transactions are supported. By allowing user to define finer units of recovery, such as nested structural transaction [Moss 86], loosing large amount of works is prevented.

Encapsulation in object-oriented database systems only allows the data to

be accessed through their operations. Since these operations are stored as part of the database, the semantics of objects and operations can be considered to achieve greater concurrency [Skar 86].

3.3.3 Secondary Storage Management

Over the years highly sophisticated associative retrieval and access path techniques have been developed for relational database systems. These techniques are mainly based on limited atomic types of attributes and restricted type constructors(sets and tuples). The restricted environment provides a solid basis for developing efficient storage structures, access strategies and optimization techniques.

Object-oriented database systems provide considerably extended data modelling facilities and support a rich set of object types and data structure. In addition to the associative retrieval over a collection of objects, access to data within a complex object should also be provided since the objects may be rather complex as well as extremely large. This requires developing appropriate solutions on storage structure and access strategies.

Clustering: Clustering is a well-known strategy to speed up access. Clustering partitions objects in a database by means of their semantic properties. So related objects are placed in the same storage segment. First option is to store a complex object with its component objects. This puts related objects that are almost always accessed together into one package. Another option is storing all instances of a class together. This is used in queries where search of all objects of a type is required. Finally objects also can be clustered based on their property values. This method allows the objects which contain a particular property value to be placed into one storage segment.

In most clustering schema, it is only possible to place an object in one group. However, in some cases there is more than one reasonable way to cluster a given object. For example, some objects may be part of several objects (shared objects). So the clustering can only be realized partly. To

solve this problem, object replication is needed [Horn 87]. Although object replication incurs the penalty for update, it is very useful for objects that are either seldom updated or read only.

Index: Conventional database systems usually index on the structure of data. In object-oriented database systems, there is a conflict between encapsulation and maintaining auxiliary access paths.

If objects are to be indexed on their structure (their value of attributes), obviously encapsulation is violated. Moreover, there are many unsolved issues. One concern is how deep to index. Is it only possible to index on the immediate attributes of an object or also on attributes of its subcomponent objects, and so on? Also if a subcomponent object, as an independent object, changes its state in other context, how should the system keep track of this update? Here index maintenance is an extremely sophisticated task.

Another question arises if we consider inheritance. In relational system, the scope of access of a query against a single relation *R* is just tuples in *R*, while that of a query against a class *C* in object-oriented database is in general the instances of *C* and the instances of all subclass of *C*, and their subclass, etc. For example class *Person* has subclass *Student* which has its subclass *Graduated-Student*. Should an index on attribute *name* be maintained for each of these classes separately or should the index on *name* be installed for all instances of the class hierarchy rooted by *Person*? Of course, such decision depends on the operations on the objects of the classes. ORION, for example, supports both types of indexes [Kim 89].

Since object-oriented database systems compute objects by invoking their operations, maintaining index on the results of operations seems to be more natural for respecting encapsulation. But the problem is that we need to know which other operations on an object can cause the result of the indexed operation to change. Thus the implementor of a class must ensure that certain operations would not disturb indexing. This is contrary to what is provided in relational systems where index maintenance is guaranteed no matter how a relation is defined [Maie 89a].

To summarize, secondary storage management in object-oriented database systems is still an open issue. Satisfactory solutions for an efficient mapping from the object-oriented data model to secondary storage is not yet realized.

3.3.4 Query Facility

The main purpose of a query language is to allow users to ask queries to database easily. Generally there are some general criteria for query languages. (i) *High level*: users should be able to express non-trivial queries concisely. This requires that the query be declarative. (ii) *Application independent*: it should be work on any possible application database and not be designated to some special application area. (iii) *Efficient*: this requires that some form of query optimization be supported. We address problems that are related to the definition of a fully object-oriented query language which satisfies these three criteria.

Encapsulation: When a query language for an object-oriented database system is defined the first decision to be made is whether or not the query language respects the encapsulation principle. If the query language cannot violate encapsulation, then the only access to objects is through methods and the data is invisible. As a consequence, the schema in the sense of schema of a relational database system is not available because we don't know what the internal structure of an object is. Also the query ability is restricted to the facilities of methods provided, it does not allow the user to utilize data in all possible ways. Consider the employee object and assume that the database designer only defines two methods, hire and fire on the class `Employee`, then all information about employee objects are encapsulated and nothing can be retrieved. [Banc 89] argues that if the query language is only used to query the data in an ad hoc query mode (no updates), there is less reason to enforce this encapsulation. Thus the query language should be able to violate the encapsulation in the ad hoc query mode. However when the query language is used as an embedded DML or as a programming language, the encapsulation must be followed, otherwise

we will lose all the benefits of object-oriented paradigm.

We believe that it is awkward that in a system the encapsulation principle is not applied in a single standard. Our data model in Chapter 4 completely respects the encapsulation, because we think that it is the database designer's responsibility to build adequate methods to reveal the state of the objects. Moreover, if the existing methods for a class are not adequate for the application users may add new methods, since users are capable to define new methods on objects.

Methods: In traditional database systems, query predicates are formed by three components: *attribute name*, which is the name of attribute, *operator*, which is either a comparison operator (<, =, >) or a set operator (contain, not-contain), and *value*, which is an instance of some data type. For example in "name=John", name is an attribute, = is an operator and John is a value. In object-oriented systems, user can define any kind of methods on a class. So these methods can be used in a predicate in arbitrary way, that is, methods can replace the positions of attribute name, operator and value in conventional query predicate. Consider the following example.

Example 3.1

Query 1: " Find the students who are taking more than four courses "

aStudentSet. Select (student. courseLoading > 4)

Where courseLoading is a method that counts the number of courses a student is taking in current term. Here the method is used as attribute name in relational queries.

Query 2: " Find the students whose name contain string 'son' "

aStudentSet. Select (student. name. contain("son"))

We assume that a student's name is a string. The method contain defined on class String tests whether a given string contain a certain sub-string. Here the method plays the role of the operator.

Query 3: " Find the students whose age are greater than the average age of

all students”

aStudentSet. Select (student. age > aStudentSet. averageAge)

Where **averageAge** is a method defined on set of students. It computes the average age of all students in the set. Here the method is used as a value in relational queries.

Although the integration of methods into the query language provides extremely powerful ability, it also causes some problems. Since an object-oriented database system in its initial state is an empty system, so a query language can not totally depend on the user defined methods and has to offer some basic operators. Moreover a query language must be very handy, it is not enough just to provide a rich set of methods to the user. It is also desired to provide the user with mechanisms to handle these methods, i.e., get their names, their meanings and syntax of usage. An object-oriented database system should provide a query interface which goes far beyond the scope of conventional query languages.

Query optimization: The ability to optimize queries in a relational system largely derives from the fact that the relational model is very simple and uniform. Contrast to this, queries on object-oriented databases are based on the message passing paradigm. Messages cause the execution of methods which are written in a computational complete programming language. This gives the object-oriented model greater expressive power than relational algebra, but also restricts the query optimization because the implementation of methods are encapsulated and the internal properties of methods are unknown from outside. This makes it difficult to apply general query optimization because it is hard to estimate the cost of query expressions. Korth [Kort 88] claims that it is impossible to achieve the same degree of optimization as in a relational system. The limitation of optimization is often cited as a major drawback of object-oriented approach. However, a number of proposals for query optimization in object-oriented database have been published. Shaw [Shaw 89] defined a query algebra that support abstract data type and object identity. Based on the algebra, some predicates transformation can be used in optimization. Query optimization remains a big challenge for research in object-oriented

database systems.

3.3.5 Data Model

A database system provides some level of data abstraction by hiding details of data storage that are not needed by most database users. A data model is the main tool for providing this abstraction. A data model specifies how information can be represented and manipulated within the formal framework of a database system. A data model mainly consists of two parts: structural part which describes the representation of real-world entities and the relationships between the entities, and the operational part which describes how the data are manipulated. As we mentioned before, a standard data model for object-oriented database systems has not been established yet. We address this issue in Chapter 4 which is totally devoted to present a conceptual object-oriented data model.

3.3.6 Metadata (Database Schema)

A database system maintains a central schema for database. A schema is a catalogue of the types defined in a database. Representation structure of stored data is specified by database schema and different application programs and users share this meta-information. Meta-information, the description information for the database, is often itself represented in the same structure as the stored data. Hence it is accessible through the query mechanism. That is a database system should provide uniform treatment of data and meta-data. In the Chapter 5 we will describe in detail how the metadata for an object-oriented database system is realized.

3.4 UNIQUE FEATURES OF OBJECT-ORIENTED DATABASE SYSTEMS

There are some features that are unique to an object-oriented database

system because these features belong to neither database systems nor object-oriented paradigm. These features, including active objects and triggers, dynamic type creation and behavior management, are consequences of combining database systems and object-oriented paradigm.

3.4.1 Behavior Management

Since the description of an object contains the data part (its values of attributes) and the behavior part (its methods), an object-oriented database system should not only provide the data management facility but also some behavior management mechanism too. The ability of managing the behavior is particularly desirable in process-intensive application areas, such as CAD, where lots of complicated design processes are involved. It is encouraged that current data management systems incrementally evolve to future behavior management systems. Behavior management includes manipulating behavior definitions of objects and generating composite behavior based on the existing behavior definitions. Unfortunately, very few existing systems have paid attention to this aspect. Beer [Beer 89] has suggested that arbitrary functions may be considered as values. This provides a good basis for behavior management because functions can be postulated just as we do with other kinds of values.

Managing behaviors in an object-oriented database system should provide a set of constructors to functions. So the users may apply these constructors on existing functions to generate new functions. A set of constructors may include composition, broadcast-to, sequencing and so on.

Composition constructor is used to combine several existing functions to build a new function which may be directly used later. Suppose we have method `advisor` defined on class `Student`, which returns the academic supervisor of a student. A student's supervisor belongs to class `Faculty`, which has a method `phoneNo` that returns the telephone number of a faculty member. If someone needs the phone number of the advisor of student `S`, he may apply two methods, `advisor` and `phoneNo`, one after

another:

s. advisor. phoneNo

Composition constructor can be used to integrate these two methods to create a new methods of class **Student**, **advPhoneNo**.

(Student, advisor). composition (Faculty, phoneNo) as advPhoneNo

The constructor may have a general format as following.

(class₁, method₁). composition

(class₂, method₂)...(class_n, method_n) as newMethod

Notice that the new method created is a method of class₁. Also the domain class of the result returned by method_i must be class_{i+1}. This actually is the condition that those n methods may be executed consecutively. Normally a composite method is a temperate one and only available in current query session. However, it also may be stored explicitly so that it is permanently stored in database and can be used just as same as other methods.

Since objects are processed by sending them messages, it is often desirable to organize several messages in a certain way so that they are sent to the receiver object properly. Behavior constructor can combine the existing methods in a conditional, iterating, sequential, or concurrent way. This provides a powerful tool to create methods as combinations out of a large number of existing methods. It also provides opportunities for dataflow execution in a parallel architecture [Bans 87]. We describe several such constructors.

anObjectSet. broadcast aMethod

The broadcast constructor is the most elementary expression of parallelism on a set. Each object in **anObjectSet** can be applied method **aMethod** in parallel. For example, if method **oldMan** gives the name and age of a employee if he is older than 50. Then the result of

aEmployeeSet.broadcast oldMan

contains the names and ages of the employees who are older than 50 in aEmployeeSet.

Condition constructors is also a very essential function constructor.

x if p method1 else method2

where *p* is a predicate, *methods1* and *method2* are methods defined on object *x*. If *p* is true, *method1* is applied on *x*, otherwise *method2* is applied on *x*.

Arbitrary complex functions may be built by applying these constructors on the existing methods. The new functions may be stored in database and a user can invoke them for further use.

3.4.2 Dynamic Type Creation

We believe that an object-oriented database system should support dynamic type creation. The main reasons are:

- to compute the relationships between the objects which are not predefined in the database schema and
- to make the object algebra (query language) as a closed algebra. Since the input to a query are objects (or set of objects) that belong to some classes, its output should also be objects (or set of objects) that have structure and behavioral properties as other objects.

Providing dynamic type creation, in a narrower sense, is the intention to keep operations like **project** and **join** used in relational systems. Relational systems capture the relationship between relations by embedding the same column in two relations. Based on this common column the relationship is computed by join operations. Since the 1NF restriction is dropped in object-oriented model and objects may represent arbitrary

complex structure, relationships may be captured in those complex structures which are defined in the schema definition. We call these kinds of relationships predefined relationships. By knowing the Oids, an object may refer to other objects directly. Thus, far fewer join-like operations are necessary in object-oriented model. However it is not realistic to expect that the database designer could foresee all possible relationships needed by all applications that share a database. For example it is useful to join road map of a country with the geographic map to produce a more complete map that contain both information. So we believe that the manipulations which create new objects are necessary. Project-like operations are useful too in the object-oriented model. It may be used to produce only necessary information as output of a query and hide some information (such as salary of employee object).

In the relational model, the input of operations are relations and the output is another relation. The relational algebra is closed and the result of a relational operation may be further processed by other operations. Since the join-like and project-like operations in object-oriented model may produce objects that do not belong to any existing classes, the situation is not as simple as in relational model. The relevant questions are

- (i) what is the type of the result objects?
- (ii) where is this new type located in the IS-A class hierarchy?
- (iii) what is the relationship between input and output? i.e., are the output objects new objects or existing objects ?

A major drawback of most of the current query languages for object-oriented databases is they do not maintain the closure property [Alas 90]. In these languages, the output of a query usually is a relation (tuple-set) which only contain the structure part and the behavior part is missing. Consequently, the result of a query cannot be further processed by the same set of language operators. To solve this problem, one solution is to make some restriction on the internal representation of objects. Carey and Bancilhon [CAR88, BAN87] use nested relations (tuples-form) as their logical views of objects. Although these languages are closed, i.e.

operations operate on nested-relations and produce nested relations, as we discuss in 4.3, this approach ignored behavior aspect of new created type. Our solution for these questions are presented in next chapter.

3.4.3 Active Objects, Trigger and Event

Certain mechanisms, such as triggers and rules, enable database systems to automatically perform defined actions in response to certain events. A trigger is a monitor placed on a data item and can be considered as a rule which consists of event, condition and action [Hsu 88]. An event specifies when a trigger has to be executed. Events usually are bound to database operations, transactions, clock events, etc. A condition is a predicate that is evaluated whenever the event occurs, if the condition become true, the action is executed. An action is a program to be executed when an event occurs and the condition is satisfied.

In object-oriented database system, the concept of active object is closely related to triggers. Most objects in databases are passive in the sense that they must receive message before performing any action. If an object can execute action without receiving message it is an active object. In OZ+ the self-triggering rules and events mechanism is implemented.

3.5 ADVANCED FEATURES FOR NEW APPLICATIONS

There are some advanced features for support of applications which object-oriented database systems are designed to serve. Features which usually appear in non-conventional database systems include schema evolution, versions, and long transaction management. Although these features may not be unique to object-oriented database systems, they are supported by object-oriented database systems in a better way then conventional database systems. Also they are closely related with the characteristics of object-oriented database systems. For example, Orion already provides some of these features.

3.5.1 Schema Evolution

At the conceptual level, a database system usually is described by the conceptual schema. In an object-oriented database system, a schema is described by the definition of classes and the structure of the class lattice, i.e. sub/super relationship. Due to the encapsulation principle, applications can only perform the operations on objects, that is, only the behavior specification of the class definition should be included in the schema. The data structure and implementation of methods belong to the internal level. Thus a schema modification is concerned with changes to the definition of the behavior of a class, i.e., adding, deleting and modifying the interface (not the body) of a method, changes with the structure of the class lattice, i.e., creating and deleting a class, and altering the inheritance relationship among the classes.

However, most current systems (Orion, Gemstone) put the complete class definition into the database schema, not only the interface of classes but also their implementation (data part). We think this mixes the internal level and the conceptual level of a database system. The precise definition of the notion of schema in object-oriented database systems is not yet achieved. Nevertheless, following the notion of schema which is commonly used in literature, schema changes may be divided into three categories [Ngny 89]:

- changing class definition: instances variables and methods.
- adding, deleting and renaming a class in the class lattice.
- modifying the class lattice by altering the relationship between classes.

A comprehensive discussion of these issues can be found in [Bene 87a], [Ngug 89] and [Skar 86a].

3.5.2 Versions

There is a general consensus that version control is one of the most

important functions in various data-intensive application domains, such as design applications. Users in such environments often need to generate and keep multiple versions of an object, and select one that satisfies their requirement. Thus keeping track of the dependencies of several versions of objects should be the task of database systems rather than the application programs.

However, there is no common concept of what versions are. Different application areas need different version concepts. In a design environment, versions may be used to record the life-cycle of a design object, to document the process of development, to represent different solutions to a problem, or to express different views of an object.

The main issue related with versions is change notification. In object-oriented database, an object may recursively refer to other objects and an object may be referred by any number of other objects. So when the versions of an object are changed or a new version of the object is created, the objects that have referred to it may become invalid, and thus need to be notified of the change.

Proposed version models include [Chou 86], [Ditt 86b], [Klah 86].

3.5.3 Multilingualism

The concept of multilingualism arises from application program interface (API) through which a user program will communicate with the database. Most current object-oriented database systems are tightly connected to a particular programming language. There is no agreement on a single high level language and applications are written in a variety of programming language. So the multilingual database systems are desired. Also a multilingual database system allows different external application systems to access and share the data resource of a database. Some research activities have been carried on this issue [Skar 86b], [Banc 88], [O'Bri 88].

To make a database system multilingual, various formats of a persistent object must be supported. Each format corresponds to a programming

languages. Therefore each requires a compiler extension unique to the language and a run time system particular to the high-level language (HLL) [Ston 90]. All these run time systems will connect to a common database system. Now the question is what is the format of the query language to this common database system? It should be SQL-like or similar to a particular object-oriented programming language. This is still an open question.

In some sense, the requirement of multilingual is contrary to the intention of reducing the "impedance mismatch". To reduce the mismatch, a function should yield same result if it is executed in a user space on transient data or inside the database on persistent data. The only way this can happen is for the execution model of the database system to be identical to that of the specific programming language. On the other hand, if we require that object-oriented database be accessed by applications written in a variety of languages, it is impossible that the type system of the database system will match all the type systems of several programming languages.

3.6 DEFINITIONS

To summarize the discussion in previous sections, our definition for object-oriented database systems contain four parts of high level characteristics:

- (i) Basic database system features, which include persistency, sharing and recovery, secondary storage management, query facility, data model and metadata.
- (ii) Fundamental principles of object-oriented paradigm, which include complex objects, encapsulation, classes, inheritance(or delegation), and polymorphism and overloading.
- (iii) Unique features, which include behavior management, dynamic type creation and active objects.
- (iv) Advanced features, which include schema evolution, long transaction

management, version control and multilingualism.

As a result, we have the following definitions. The first one which is a basic definition may be used to judge if a given system is an object-oriented database system. The second definition is an extended one and we consider it as the framework of future database systems.

Definition 3.1 Basic definition: An object-oriented database system should have the basic features of database systems of (i), and follow the fundamental principles of object-oriented paradigm of (ii).

Definition 3.2 Extended definition: In addition to satisfying the requirements of the basic definition, (i) and (ii), an object-oriented database system should possess the unique features of (iii) and the advanced features of (iv).

3.7 DIFFERENCES FROM PREVIOUS DATABASE SYSTEMS

From our definitions the distinction between traditional and object-oriented database systems is clear. Relational database systems do not follow the object-oriented paradigm in 3.2. Although the network and hierarchical systems partially support the complex objects and object identity, they do not support encapsulation, class, inheritance and polymorphism. Thus, there is a clear distinction between traditional database systems (relational, network and hierarchical) and object-oriented database systems.

However, there are some kinds of systems or models that are much closer to object-oriented database systems than traditional database systems do. These systems or models include *semantic model*, *knowledge base system*, *nested relations* and *extensible systems*. The evolution of the object-oriented database system is heavily influenced by those technologies. Therefore, there is a certain amount of overlap between them. It is not always easy to see the borderline between these fields and object-oriented approach. As an application of our definitions, we attempt to characterize the distinction between these technologies and object-oriented approach.

Semantic Modeling: As we see it, a semantic model is probably the one that is closest to the object-oriented system. Due to the simplicity of relational modeling, any semantics that could not be expressed in terms of relations have to be embedded in the application code. Semantic models are proposed to react to this drawback of the relational model. They provide high level modeling constructs so that more semantic information about the data may be represented and stored in the database.

Semantic data models provide object identity and class hierarchy concepts. However they don't include the concepts of data abstraction and user defined operations (methods). Lack of data abstraction and user defined operations (methods), objects in semantic models have only structural part but no behavioral part. That why sometimes semantic models are also called structural object-oriented (as opposed to the behavioral object-oriented) [Ditt 88a]. These two features distinguish object-oriented systems from semantic models.

Knowledge Base Systems: Knowledge base system is another term that has no universally accepted meaning. This is partially because the word "knowledge" is a tricky notion to define formally. Ullman [Ullm 89] defines a knowledge base system as a system that

- provides what a database system provides and
- provides a single, declarative language to serve the roles played by both DML and host language in a DBMS.

This is obviously not enough to distinguish it from an object-oriented database system. In our opinion, although knowledge base systems sometimes also include the systems that support "objects" in object-oriented sense, the term always implies supporting for logical rules and inferences.

Nested Relations: Nested relations attempt to increase the modeling power of relational model by dropping the requirement that relations be first normal form. This idea is also referred as unnormalized relations or NF squared (non first normal form) relations. In the sense that they are meant to address some shortcomings of relational model, their objectives

are similar to those of object-oriented systems. On one hand, lacking of encapsulation and class hierarchy, they only support part of the features that found in object-oriented systems. On the other hand, the works on NF squared relations tend to be more mathematically rigorous than the work on the object-oriented systems. Proposals for an algebra and a calculus [Guti 89], and even a normal form [Ozso 89] for NF squared relations have been emerged. Therefore we may look nested relations as something between relational systems and object-oriented systems.

Extensible Systems: The motivation behind extensible systems is the fact that it will never be possible to provide built-in features that will support everything that will be used in future. Extensible systems attempt to find approaches for building a database system which can be easily extended to accommodate new functionality [Lind 87]. Such systems are usually built by assembling components from a library of building blocks. Of course, object-oriented systems allow users to define their own data types and functions. In object-oriented systems inheritance is used to realize extensibility. Extensibility also can be provide in any other kinds of database systems. So object-oriented systems provide a special approach to accomplish the extensibility.

Chapter 4 An Object-Oriented Data Model

4.1 WHAT IS IT?

There are several interpretations of the term "data model" in the world of databases. It can be used to denote the schema or description for the information in a particular application database. For example, we can call the schema of a bank account database as "bank account data model". It also can be concretely interpreted as the data definition facility, i.e., the data definition language, of a particular database system. For example, the data definition language of ORACLE specifies "ORACLE data model". More broadly, the data model means the mathematical algebra and logic shared by a category of similar database systems. It provides the general principle in which information about application is represented and manipulated. For example, "relational data model" presents the common core of all relational database systems. In relational data model, application information is captured in relations and individual entities are represented as tuples. Relational model also provides a set of operations on relations. All of these are based on the solid mathematical theory, relational algebra. Obviously, it is the last interpretation of data model that we are interested in when talking about the object-oriented data model. However such formal data model for object-oriented database has not been achieved yet. For a given object-oriented database system, it has its own data model. For example, the object-oriented data model of Vbase is described by its data definition language, DDL. However, the common mathematical structure across all object-oriented database systems has not emerged yet.

People have reached a consensus on the relational, network and hierarchical data models. In traditional database systems, a data model is a combination of *data structures*, *operations* on these structures and *constraints* on the state of these structures [Tsic 82]. The data structure part includes some primitive data types, record structures such as, *tuple* for relational and *logic record*(node) for hierarchical and network, and structure constructors such as *set* for relational, *tree* for hierarchical and

DBTG set for network.

There are several features of conventional data models which we may notice. First, conventional data models consist of only a few general *parameterized* data types (relations or record types). Stating that a data type is parameterized means that structures can be defined over different record types while the record types can be defined over different primitive data types. During the database schema definition, the parameterized data types are specified. When we define a relation the attributes of the relation and the domains of attributes are specified. Second, there are very few general operations provided. Operations are the same regardless of the semantics of the data involved. Third, the structure of the data automatically determine the operations on the data and new operations can not be added. When we define a relation we automatically get the operations such as creating a new tuple of the relation, setting and getting the value of attributes of the relation. As a result, the data types (structure and associated operations) are determined by their structures. This fact indicates that the conventional data models are not abstract type systems. Since the conventional data models only consists of a fixed set of parameterized data type, they are even not true type systems [Maie 89b].

Object-oriented database systems allow users to define new data types (classes). The data structures are used to represent the internal states of objects. Operations on objects are defined explicitly and are not implied in the structures of the objects. All objects do not share the same operations but they have their own operations. Therefore, the question is what features characterize the data model of object-oriented database systems? Most existing object-oriented data models are based on the type system of a particular programming language such as C++ or Common Lisp. Since the type system of a particular language won't suit all database systems, a general language-independent standard of data model is hard to achieve.

Since what is a data model in the context of object-oriented database systems is still an open question, we use the term of "data model" in the classical sense, that is, our data model consists of two parts, a structural part and an operational part. In this chapter by capturing the common core of current systems and adding some of our own ideas we present an object-

oriented data model. On the structural aspect we propose our definitions for some essential concepts, such as, objects and values, states and attributes, and object identity. These concepts are much overloaded and sometimes are misleading in literature. We attempt to clear up this situation by proposing precise definitions for these concepts. For some more sophisticated concepts such as relationship and set of objects, we furnish them with clear semantics and powerful capability to cover the drawbacks of the approaches proposed in literature. On the operational aspect we deal with three general types of database operations, retrieval operations, updating operations and meta operations. Our goal is to provide such operations for object-oriented data models that not only take full advantage of the object-oriented paradigm, such as, encapsulation and rich modeling capability, but also retain the nice features of traditional data models, such as set-oriented and the closed property in relational model.

4.2 STRUCTURAL PART

4.2.1 Objects and Values

Intuitively, the concept "object" contains everything. Every data entity in an object-oriented database is an object. However, in a narrower sense, objects are used to abstractly describe the application entities. Such as student and department, etc. What are values? Look at number "eight". Why it is considered as a value while a student instance is considered as an object? Possible reasons are listed as follows:

- i) Values are printable while abstract objects are not.
- ii) The information about a value is carried by itself, while the information about an object is described by other objects and values.
- iii) All values have names in name space, while objects normally don't have names.
- iv) Values are built into the system: they need not to be defined, objects need to be defined in the system.

Distinctions i) and ii) are from logic point of view. Values such as the number eight, string "abcd" are printable, while a student or a car are not. However, note that fundamentally we never print the object eight, "eight" or "8" is just its name. We also can simply print the name of a student or license number of a car. Distinction ii) states that values describe other things while objects are the things being described. This is also no absolute distinction: values may also be the things being described. Such as in a mathematician's database, numbers are the subject being described. We believe that the logical meaning of values and objects may differ from one context to another. It is hard to establish the precise definition of the difference between the two concepts.

Distinctions iii) and iv) are rather technical and they are system dependent. Many object-oriented database systems allow users to name arbitrary objects and built-in types in different systems are different too. For example, a string is built-in types in many languages. But some systems such as, C language, consider it as concatenation of characters. Our argument is that although iii) and iv) are system dependent, for a specific system they catch a clear distinction between values and objects.

An object-oriented database system provides a set of primitive types, and the instances of those primitive types are called primitive objects. If we apply iii) and iv) on primitive objects, we may find that the primitive objects match the property of value very well. They are built in the system. Each instance of a primitive type has a name. They are available whenever users need to use them and they don't need to be created before they can be used. If we call the types that are defined from the primitive types or other user-defined types as derived-type, we have our definition of values and objects as:

Definition 4.1 Instances of a primitive type are values while instances of a derived type are objects.

Although this definition is system dependent, for a given system, the borderline between value and objects is clear. Values with their names are hard wired into the system. Whereas abstract objects (and their name, if

they are given names) have to be defined.

Based on **Definition 4.1**, we treat objects and values differently. An object can be referred to by any number of other objects. Object **A** refers to object **B** means that some surrogate of **B** is stored in **A**, so **A** can send messages to **B** to invoke **B**'s methods. So once an object changes its state, all the objects which refer to it will be affected. On other hand, a value can not be referred by other objects but is duplicately stored in the objects that need it.

4.2.2 States and Internal Representation of Objects

Like most other models, in our model, the internal structure of an object is described by a set of attributes. Semantically, we can classify attributes into two categories. *Descriptive attributes* are these attributes that describe the properties or overall features of objects. Examples of descriptive attributes are attribute **color** in object **myCar** and attribute **age** in object **john**. *Reference attributes* are attributes that contain the references to other objects. For instance, attribute **chairman** of **mathDept** and attribute **supervisor** of **studentA** are examples of reference attributes. Reference attributes provide the means of relating objects to each other. Relationships between objects are computed through those references by methods.

An interesting observation is that descriptive attributes usually are values and reference attributes are abstract objects. This is not surprising. Since, as we just discussed, describing other things or being described makes significant distinction from abstract objects and values. Thus we have following definition for these two kinds of attributes:

Definition 4.2 Descriptive attributes of an object are the attributes whose domains are primitive types whereas reference attributes are those whose domains are derived types.

Information about an object is reflected by the state of the object. We believe that the concept of state is fundamental to an object-oriented model.

The concept of the state of an object is distinct from the object itself. Although state of an object may change from time to time, the object logically is still the same one. Also an object is an abstract concept whereas its state may contain value information. The concept of state explains two types of equality, equality of objects and equality of their states.

Although the semantic meaning of states is clear, to our knowledge, there is no proper definition for this concept in the context of object-oriented databases. Most systems consider the states as the value of attributes. For instance, in O2 the state of an object is just a tuple, formed by types and values of the attributes. In Orion, the notation of state is dispensed with and attributes defined directly on objects are used instead. In these systems the state and internal representation of objects are somehow mixed. We argue that the two are different concepts. The types and values of object's attributes is only the internal representation of the object. While the state of objects may be observed by users.

For each class (type), representation (attributes) is used to store the state of its instances. Since only the methods of an object can manipulate its representation, the representation of the object is encapsulated. Sometimes changes of the representation do not reflect changes of the state. For example, a triangle is represented by its three corners. These three points are stored in an ordered-set. Changing the order of these three points would not affect the state of the triangle. Moreover, some states of objects can be the results of certain methods and they have no corresponding attributes. Those states are also referred as derived attributes. For example, there is no attribute which store the area of a triangle, but the area can be calculated from the value of other attributes by a method. But users don't know whether a result is derived from other attributes or it is the value of a attribute. They are both viewed as the results returned by methods.

Also since an object is described by its structure and behavior, the state of the objects should include the behavior aspect too. We define the concepts of state as follows.

Definition 4.3 The state of an object is the set of results returned by all methods defined on it.

First, this definition separates the internal part of an object which is inaccessible from outside from the visible part. Second, the results of methods depends on both the value of attributes and definition of the methods. The state under our definition captures both the structure and the behavior aspects of objects. Third, it explains the nature of derived attributes, since results of all methods contain those derived attributes too.

4.2.3 Object Identity and Equality

One of essential differences between object-oriented data model and relational data model is the absence of identity concept in the relational model. Because a relation is a mathematical set, there must be a set of attributes in the relation that constitutes a key. By comparing the values of the key, users may know whether two tuples are the same or not.

Every object in an object-oriented database is associated with a unique object identifier (Oid). In object-oriented database systems Oids are very important for several reasons. First, Oids provide a permanent handle for objects whose state and behavior may change from time to time. Second, since the state of a complex object includes other objects, with their own state, the state of an object can be described as a set of Oids of objects that are included in the state of the object. Without Oids it is very hard to represent the state of a complex object. Third, since an object's contents are encapsulated, they can not be expected to provide a means for identification.

With Oids, equality of objects may be handled easily. Two objects are the same if and only if their identity are the same. Notice that the Oids must belong to some primitive types, such as integer. So Oids are values and do not have their own Oids and whether two Oids are the same can be checked by operations defined on primitive types.

4.2.4 Set of Objects

Distinguish Class and Object-Set: In order to provide set-oriented operations, queries should be against the collections of objects instead of a single object. So the database systems need to provide some mechanism to group a number of objects together. We call a collections of object as an *object-set*.

Some systems consider a class to be a synonymous with an object-set, which contains all instances of the class. As soon as an object of a class is created, it is automatically put into the object-set that contains all instances of the class. Users don't need to specify which object-set it belongs and it may not be removed from the class unless it is destroyed. This approach can take advantage of inheritance hierarchy, since the relationships among those object-sets are the same as the relationships among the classes. Also it enable users to automatically reach any existing objects of interest, because all instances are included in their classes. Class=set concept also make it easier to conceptualize a query [Kim 89]. Acturally, this approach groups objects in the same way as in relational model, where relations=set.

We find there are some restrictions for this approach. For example, documents for several applications may have the same specification, so they belong to the same class **Document**. It does not always make sense to use **Document** as query target, since it contain documents for all applications. An application may only interests its own instances of **Document**. We think it is more realistic to ask each application to maintain and manipulate its own set of documents. Second, since every object is included in its class, so the database can not store objects that do not belong to any object-set. For example, for class **Employee**, there is no way to model "someone is hired but has not started yet" or "someone is on a temporary leave", since they are not employee currently. For the latter case, we have to delete the employee object from database first then add it into database when the employee is back. Finally the heterogeneous object-set is not supported.

Therefore, we support the view in which the concept of object-sets are independent from the class. In our model, arbitrary object-sets may be

formed to meet different requirements. From user's point of view, this provide a simple way to access the information they are interested in. For example, we have a database which stores employees for several companies. The employees of each company form an object-set. The users from a company may query on the object-set that only contains the employees of his company. Also certain security may be implemented. The president of a company may create an object-set **candidates** that contains the employees from whom the vice-president will be selected. Although **candidates** is a subset of all employee in the company which may be accessed by other users, other users still can never discover the president's "candidates for the vice-president", because they don't know the handle of **candidates**. From implementation point of view, creating arbitrary object-sets usually means that all instances of a class are broken into several smaller object-sets. Thus index structures are much smaller in this case. Finally it provide a way to support heterogeneous object-set. As a consequence of separating the concepts of class and object-set, classes in our model are no longer used as a container of objects but to define a template for its instances.

Object-Set as First-Class Object: Now we describe how object-sets fit in our data model. From traditional point of view, set of objects is an aggregating constructor that groups objects together. It is the system that provides this constructor. Also appropriate operations on the set of objects are provided by system too. With these operations, sets of objects may be accessed and manipulated. Since the set constructor and operations are system provided, users are unable to modify or extend them. Moreover, a set of objects is treated differently as other objects. As a consequence, something that is not an object exists in an object-oriented database.

To follow the object-oriented paradigm completely, in our model, we treat object-sets as objects rather than a metalinguistic constructor. We consider object-sets uniformly as other objects. Since object-sets are objects too, their behavior are specified by methods defined on them. Basic operations include inserting(deleting) objects into(from) an object-set, copying entire object-set and the operations that support set-oriented query, such as **select**, **project** and **join**. We will formally present these

operations in the operational part of our data model. As these basic operations, sophisticated functionality of object-sets may be implemented as their methods too. For instances, if a member object is being manipulated, then the object-set will hold further messages that are sent to the object until the updating is finished. Also an object-set may determine how the messages it received are propagated to its members, sequentially or in parallel. This make object-oriented database system a big difference from traditional database systems. While data in traditional databases are passive and all the operations are issued from outside, object-oriented database system lets you include much of the operations(methods) in the database itself. Therefore, some traditional DBMS functionalities become the behavior property of stored objects in databases. Treating object-sets as objects provides users with opportunity to change and extend the functions of object-sets by defining additional methods. This also increases the extensibility of the system.

Since object-sets are objects they must belong to some classes, i.e., object-sets are defined by the classes they belong to. The nature of those classes largely depend on what kind of object-sets are allowed in a model. Most current systems only support the homogeneous object-sets, that is, all the member objects in an object-set belong to the same class. Thus the class of an object-set is related to the class of its member objects. In ENCORE data model [Zhon 86], the class of object-sets are considered as parameterized type, $S[T]$. If the member objects in an object-set are belong to class **Student**, then the object-set that contains student objects belong to class $S[\text{Student}]$.

To our knowledge, no current systems support heterogeneous object-sets. Also the relationships between these object-sets classes has not been well explored in current literature too.

Classes Of Object-Sets: In our model, arbitrary kinds of object-sets, including heterogeneous object-sets, are allowed. We first classify various object-sets into several categories. For each category, classes are described to define the object-sets of this category. Also the relationships between these classes are explored.

The most general kind of object-sets in our model is heterogeneous object-sets. Objects contained in a heterogeneous object-set may belong to any class. We may use one class, denoted as **HOSet**, to define those object-sets, i.e., all heterogeneous object-sets are instances of **HOSet**. Since a heterogeneous object-set may contain objects of any classes, it has no knowledge about the classes of its members. There are some operations that treat an object-set as a whole part. In this situation, the information of member objects are not needed. Such operations include copying an entire object-set, deleting an entire object-sets and inserting an object to an object-set, etc. These operations may be defined on **HOSet**. Other operations on an object-set need to propagate the message to its member objects. We can process the query, "Find the persons whose ages are older then 16", on an object-set which contains instances of class **Person**:

```
aPersonSet. select (x. age > 16).
```

Here, we need to know that objects in **aPersonSet** are instances of class **Person** and methods **name** and **age** are defined on **Person**. An instance of **HOSet** is unable to be applied by this kind of operations, because it does not know the classes of it members and what methods are defined on the members. The above select operation is not available for an object-set that contains instances of **Person**, **Course** and **Department**.

Second category of object-sets in our model is heterogeneous object-sets with some restrictions. We use **RHOSet** (Restricted Heterogeneous Object-sets) to denotes this kind of object-sets. The object-sets in this category may contain the instances of class **C** or the instances of the subclasses of **C**. For the object-sets which contain instances of the classes that are in the class hierarchy rooted at class **C**, we use class **C-RHOSet** to define them. For example, if class **Person** has two subclass **Faculty** and **Student**, which, in turn, has subclass **GradStudent**, then class **Person-RHOSet** defines the object-sets that contain objects that belong to one of the following classes, **Person**, **Faculty**, **Student** and **GradStudent**. Object-sets of **C-RHOSet** may propagate certain message to its members. The messages propagated must be defined on the root class, **C**. Since any method defined on class **C** is also inherited by its subclasses, all the objects in an object-set

of **C-RHOset** can accept the messages defined on **C**. Previous select operation is available for **Person-RHOset**. Because **name** and **age** method are defined on class **Person**. Thus the following statement is valid.

aPerson-HOSet select (x. age >16).

Also we notice that in the class lattice, there is a sub-lattices rooted **C** for each class **C**. Thus each class **C** has its corresponding class **C-RHOset**.

Third category of object-set is the homogeneous object-sets, that is, the object-sets only contain instances of a single class. We denote this kind of object-sets as **HomOset**. The object-sets that contain instances of class **C** is defined by class **C-HomOset**. Since an instance of **C-HomOset** has all the knowledge about its member's class, it can propagate any message that defined on **C** into its member. Also as in **RHOset**, for each class **C** there is corresponding class **C-HomOset**.

We describe the generalization/specialization relationships between those object-set classes. Since class **HROset** defines the heterogeneous object-sets without any restriction. It is the most general object-set class. Any other object-set classes are its subclasses.

For any class **C** in the class hierarchy, as we pointed, has two corresponding object-set classes, **C-RHOset** and **C-HomOset**. Obviously, the later is a special case of the former. So we treat **C-HomOset** as a direct subclass of **C-RHOset**. Moreover, if **C** has a subclass **S**, then logically the objects contained in object-sets of **S-RHOset** are instances of **C** too. We put **S-RHOset** as a direct subclass of **C-HomOset**. **S-RHOset** has a direct subclass **S-HomOset**. Following example shows how these object-set classes related.

Example 4.1

If we have class **A**, **B**, **C** and **D** which form a class hierarchy as shown in Fig. 4.1 (a). The four classes introduce nine object-set classes which form a class hierarchy as shown in the Fig 4.1 (b).

Fig. 4.1 An example of object-set class hierarchy

We need to notice two things in our discussion. First, most systems provide a class which is the root of class hierarchy, that is all classes in the system are its subclasses. In our example, class A is the root class. In this situation HOset is actually equal to A-RHOset. Also a leaf class in the class hierarchy does not have any subclasses. Since classes B and D in our example has no subclasses, B-RHOset and D-RHOset are equal to B-HomOset and D-HomOset respectively. From this observation, we have Fig.4.1 (c). For the total number of object-set classes in a system we have follow formula:

$$\begin{aligned} & \text{Number (object-set Classes)} \\ &= \text{Number (classes)} * 2 - \text{number (leaf classes)} - 1. \end{aligned}$$

In Chapter 5 we will describe our implementation of object management facility in the object-set-style.

4.2.5 Relationships

A data model can represent relationships between the entities. Relationship are one of the most fundamental parts of any data model. From one point of view, it is what distinguishes database systems from file systems.

A relationship is a correspondence between entities. For example, **workFor** relationship relates **joe** to **computerDept**. This relationship has some implied directionality. That is, given **joe** we can use **workFor** to determine **computerDept**, but not in other direction. We define **joe** and **computerDept** as the *subject* and the *object* of **workFor** relationship respectively. To move from **computerDept** to **joe**, we need another relationship **hire** that relates **computerDept** to **joe**. We call **hire** is the *inverse relationship* of **workFor**. In our model for the efficient execution of queries we require that both a relationship and its inverse are exist.

Look at another example, "Joe works for Computer Department since June 1". Here, there is an extra information "June 1". Notice that "June 1" is neither solely related to **joe** or solely related to **Computer-Dept**. "June 1" is determined by the combination of participating entities in the relationship, and not by any single one. It is some information about the relationship itself. We define this kind of information as *association information* because it is associated with both the subject and the object of the relationship. Some relationships have association information while others have not.

Traditionally, relationships also may be classified into 1-1 relationship which relates one subject to one object, 1-N relationship which relates one subject to more than one objects and N-N relationship which relates a set of subjects to a set of objects. We believe that this classification of relationship needs to be modified in our data model. Since we consider that a set of objects is an object too, the concept of 1-N and N-N relationship is not needed in our model. This is illustrated in Fig. 4.2., where traditional 1-N relationship (Fig. 4.2 (a)) become the 1-1 relationship that relate an object

to an object-set (Fig. 4.2 (c)), whereas N-N relationship (Fig. 4.2 (b)) became again 1-1 relationships that relate two object-sets (Fig. 4.2(d)).

Fig. 4.2 Graphic Representation of Relationships

Relationships in Our Model: In the relational model, the relationships between entities are represented as relations too. This provides a uniform treatment for both entities and relationships. As pointed out in [Maie 90], in an object-oriented model, a relationship may also be named and users can express queries on those relationships. Thus the relationships are objects themselves. We do not take this point of view in our model. The reasons are stated in the subsequent sections.

From the modeling point of view, entity and relationship are two different concepts. The real world consists only of entities, such as people, cars and books, etc. A relationship is a correspondence between entities. So relationships should be distinguished from entities at the conceptual level. A good example is that in E-R model they are treated as two different concepts. From users' point of view, since a relationship always attaches

some entities it is desirable that users consider a relationship as part of the information about the entities that form the relationship instead of as an independent data item. Thus users need not deal with complicated relationships among the entities, i.e., they should not feel the existence of relationships. For example, the parents of a student consider the grade of their child on some courses as the information about the student while the teacher who teaches a course may consider the grades of the students in the class as the information about the course. They both don't need to realize that a grade relationship exists between the student and course, and don't need to query explicitly on the relationship.

Also it might not be possible to instantiate a relationship apart from the objects that form the relationship. If users are allowed to directly deal with relationships and relationships are treated as normal objects, then the database might be introduced in an inconsistent state by creating a relationship that has no objects to attach. To prevent such situation, additional constrain mechanism is need to be implemented. Obviously, this is a burden for system implementation.

For the above reasons, the relationships are not explicitly made visible to the outside world.

Relationship Without Association Information: In the hierarchical and network models, most relationships are realized by physical links. For example, to represent `workFor` relationship, `joe` contains a pointer which points to the physical address of `computerDept`. In relational model, some relationships are implied in the content of attributes. If two relational tables have a same column, then the `join` operation can use the common column to compute the relationship between the relations.

In an object-oriented data model, the relationships without association information may be implied in the definitions of classes and object instantiation. So no additional effort is needed to represent this kind of relationships. For example, when the class `Employee` was defined an attribute was specified to contain a reference to an instance of class `Department`. At the time that an instance of `Employee`, `joe` was instantiated, a reference to `computerDept`, an instance of

Department, was stored in Joe's dept. attributes. Up to this point the relationship workFor from joe to computerDept is established. Also the relationships may be computed by a methods defined on Employee or Department. If the query is "Find the name of the department that Joe works for", the query invokes the method of Employee which sends a message to computerDept. This message, in turn invokes a method of Department and the name of the computerDept is returned. In fact the communication chain between objects may be arbitrary long. To find Joe's department's chairman's name, the computerDept may send the message to its chairman to get his name

The hire relationship from Department to Employee is a traditional 1-N relationship, because a department usually has more than one employee. To implement a traditional 1-N relationship most current systems store a set of references to the objects of the relationship in the subject of the relationship. Thus, the attribute, staffs of computerDept, would contain all references to its employee objects. In our model, we consider a set as an object instead of a constructor. So we store the reference to the set instead of the references to all individual objects.

In the computerDept we store a single reference to the object-set S which contains all the employee objects in the computer department. So a traditional 1-N relationship becomes a 1-1 relationship in our model.

The main advantage about our approach is that the system has well controlled structure, because when an object needs to communicate to a set of objects it hands the control over to the object-set and it never directly contacts the individual objects. In most current models, to calculate the average salary of the employees in computerDept, a method is defined on Department to do the calculation because computerDept contains all the references to its employee objects and individual employee object is accessed by computerDept directly. If there is an object CarletonUniv, which belongs to to class University, needs to do the same thing, another method has to be defined in University too. Because Department and University are two classes with no inheritance relationship between them, two methods are needed to be implemented to

accomplish the same task. Obviously, this is not efficient. In our model, since only the reference to *S* is stored in *computerDept*, When the *computerDept* needs to calculate the average salary of its employees, it may send a message to invoke some method defined on *Employee-RHOSet* instead of computing by its own method. The *CarletonUniv* may do the task in the same way. Thus, only one method defined on *Employee-HomOSet* is needed.

Relationships With Association Information: As we discussed, some relationships have association information with them. Since association information is related to both the subject and the object of the relationship, the user should be able to access it from both the subject and the object. How does a relationship with association information be represented in object-oriented model?

A straight forward approach is to store the association information in both the subject and the object of the relationship. The problem of this approach is obvious. In addition to the data redundancy, it also makes updating not convenient. When the association information is changed, both subject and object need to be updated. Modification on only one may introduce an inconsistent state. Another approach is the analogue of relational model. In relational model, the relationships with association information are represented as relations too. Users may query relationship without going through the relations that form the relationship. So in an object-oriented model, relationship might be represented as normal objects. Although this approach does not have the problems of the first approach, it has some drawbacks as we discussed before. So we need to modify this approach so that users may not directly access these relationship and even do not feel their existences.

The idea is simple, the data items that represent the relationships with association information are hidden from users, i.e., they are not shown at the conceptual level of a database. However, this data item must be accessible by the subject and the object of the relationship, so users may obtain the association information by querying the object or the subject. In our model, these data items are in the form of objects too, they can only be accessed by certain objects that are the subjects or objects of the

relationship and can not be accessed by users or other objects. Because of this properties we call them *internal objects* and the class that defines the internal objects is called *internal class*. Actually some systems, as C++, already support this concept. In C++, when a class is defined, its methods may be declared as private and some existing classes may be declared as friend classes. Thus, only the instances of friend classes may invoke those private methods by message sending.

We explain this approach through the following example. There are students $S_1, S_2, \dots, S_n$ and courses $C_1, C_2, \dots, C_m$ and they belong to classes **Student** and **Course** respectively. One association information about the relationship between students and courses is *grade*, i.e., S_i takes C_j with grade g_{ij} . This relationship is captured in another class **Stu-Crs** which has three attributes, **student**, **course** and **grade**. Obviously there are at most nm instances of class **Stu-Crs**. Class **Stu-Crs** and its nm instances are invisible from users and they only can be accessed by instances of **Student** or instances of **Course**.

We call the internal object which represents relationship with association information an *association map*. Generally an association map contain two parts of information, the entities which form the relationship and association information of the relationship. Therefore, between the subject and the object of the relationship there is no direct links. In some sense creating an association map between the entities which form the relationship follows the same principle as normalization in relational model. Thus it eliminate the potential inconsistent state caused by updating.

4.2.5 Class and Object Graph

Hierarchical structure (by hierarchy we mean a partial order among the entities) arises from the inherited relationships between ancestors and descendants. Thus the nature of a hierarchy depends on the exact meaning of inheritance. Inheritance has many forms depending on what we wish to inherit when and how the inheritance takes place. We think the most important thing about it is to distinguish the inheritance of implementation

(code) from the inheritance of behavior (specification). We discuss there two kinds of inheritance and their corresponding hierarchies in the following section.

Class hierarchy: It is clearly desirable that the data model should be capable of capturing *is-a* relationship. So in our model we require that the behavior of a subclass must be *compatible* with the behavior of its superclass. Unfortunately, there is still no agreement on the formal specification of object's behavior. It is an unsolved problem to formally describe the behavior of an object. For an in-depth discussion, see [Wegn 88] and [Tour 86]. In our discussion, by saying a class is compatible with its superclass we only mean that in any context, an instance of superclass can be replaced by an instance of its subclass without causing syntactic error. When we define a new subclass C of class P, following rules must be followed:

- (i) Additional attributes and methods can be added to C;
- (ii) Any methods inherited from P can not be removed from C;
- (iii) A method m_P in P can be replaced by a method m_C in C, following the *contravariant parameter rule*: (C and P are instances of C and P, a_i and b_i are parameters of methods m_P and m_C respectively)

IF $a = p.m_P(a_1, \dots, a_n)$ and $b = s.m_C(b_1, \dots, b_n)$
THEN b_i must belong to the same class or *superclass* of a_i , and
 $(i=1, \dots, n)$
 b must belong to the same class or *subclass* of a .

Note that these rules make it possible that the type of an expression be determined at compile time. For object-oriented database systems this feature is important because there is no need to insert expensive run time checks in the compiled code. Also since a subclass is logically compatible with its superclass, our model permits formulating queries over an object-set that consists of all instances of a class and its subclasses.

Implementation Hierarchy: Implementation hierarchy provides a mechanism that supports the software engineering principle of code reuse.

This hierarchy allows the attributes and methods of a class to be inherited by another class. So we may reuse what is similar between old class and new class. In most current systems, there is only one hierarchy, i.e., class hierarchy and implementation hierarchy are mixed. However, although class hierarchy and implementation hierarchy are closely related it is not necessary that the two are exactly same.

Consider a stack and a double-end queue (deque). Deque may be viewed as a subclass of stack because deque is a stack with additional data and operations. From implementation point of view, since deque need two list pointers while stack only need one, it is simpler to have code for stack inherit from the code of deque rather than in the other direction. The implementation hierarchy is hidden from the user since it only reflects the implementation strategy of the system.

Class Graph and Object Graph: At the schema level, we can view an object-oriented database as a collection of classes inter-related through various relationships. Thus the schema of the database may be represented as a directed graph where vertices denote classes and edges denote relationships.

One of these relationship is sub/superclass relationship that forms class hierarchy which we discussed before. In our graphical notation, a pair of sub/superclass is denoted as an edge from subclass to superclass. Other relationships have different interpretations in existing systems. In many systems, the relationships between classes are based on the structural definition of the classes. If the domain class of an attribute in class A is class B, then an edge from A to B is present in the graph. The graph formed by all these edges sometimes is called composition hierarchy, because it captures the composition information of class structure. The main restriction of this approach is that behavior aspect (definition of the methods) are ignored. Also it is contrary to the encapsulation principle, since the structural (attributes) definitions of classes should not appear in the schema.

We propose a class graph based on the message sending concepts. If the objects of class A may send a message to the object of class B then a arrow

from A to B is used to represent this fact. We show the schema of a database containing information about companies, employees and automobiles (see [Kim 89c]) in Fig. 4.3. The ovals denote classes and the small circles represent primitive classes. The edges denote message sending paths. The inheritance hierarchy is represented by those dark arrows from subclasses to superclasses.

At instance level, a database can be viewed as a collection of objects, grouped together in classes, and inter-related through message sending paths. The graphical representation of a database at instance level is called *object graph*.

Fig. 4.3 An example of class graph.

4.3 OPERATIONAL PART

4.3.1 Operations On Object-Oriented Database

The operational part of a data model usually includes a set of operations for specifying updates and retrievals on database. Before we present the

operational part of our model, we address some important points which are unique to object-oriented database systems.

First, in conventional database systems, data have always been viewed as passive. Operations performed on data are issued from outside. Through DBMS, users manipulate the data and cause the database to be modified from one consistent state to the next. The data and the code manipulating data are separated. In an object-oriented database, an object is no longer only passive data, it has its own behavior. Users may define any methods on an object. Thus an object-oriented database system should not only be considered as a repository for persistent objects but also provides a workspace of executing applications because the knowledge about the applications is embedded in the methods of objects. Therefore unlike traditional database view in which running applications are explicitly outside the database, objects should be actively executed within database.

Second, in traditional database systems, users may directly access the data in database, while in an object-oriented database objects are encapsulated and their internal value may not be directly accessible. However, associative operations are based on the state of objects. For example, a query predicate against objects' state often be used as search criteria of the query. While respecting to the encapsulation principle, the state of an object is always obtained as the result of the methods defined on the object.

Third, in any data manipulation expression, there must be some *named objects* serving as the starting points for searching desired objects. So a database system must support names for data items in some way. In conventional database systems only restricted types of data items can be named. As in relational systems, relations are associated with persistent names. Tuples and the values of attributes have no names. That is why relational queries are always against relations. The network model supports names for record types, i.e., each record type has a name. It also supports names for individual record occurrence in a restricted way. Various currencies are used to refer to certain record occurrences. For instance, currency RCU(current-run-unit) points to the most recent accessed record occurrence. However, currencies are transitory and do not persistent past

the end of a user session. We define named objects as persistent variables: whose value retain within the database system from one session to the next. In our model we permit a persistent variable may be declared on any type. Since objects are grouped into object-sets, object-sets are usually used as start points of query process. Therefore, most object-sets have names. Moreover, some individual objects especially those are frequently referenced also can be named too. For example, the chairman of a department, although is (treated as) an object of class **Employee** and is contained in some object-sets, we might want a persistent variable that holds this instance of **Employee**. So users can directly access this object, rather than search it in the object-set that contain it.

4.3.3 Update Operations

Update operations are those operations that do not return any information but change the state of the database. Basic update operations include creating new objects, destroying existing objects, inserting an object in an object-set, removing an object from an object-set, and modifying the state of individual objects.

Create: Operation for creating a new object of class **C** is a method, **create**, which is applied on class **C**. Methods applied on classes are sometimes called class methods while normal methods which applied on the instances are called instance methods.

C. create (initial-values)

What the above statement does is to set the attributes defined on **C** with values provided in initial-values. However, this does not mean that users need to know the internal structure of the object. What information required to create an object of **C** is specified in the specification of method **create** of **C**. Thus initial-values is only a parameter of method **create** and it does not reveal any information about the internal structure of the objects of **C**.

Destroy: There are two ways to destroy an existing object. Some systems

provide explicit operations for destroying an object. In these systems, when an object need to be removed from database, the destroy operation is explicitly applied on that object. One problem of this approach is that if an object is destroyed then objects that refer to this object may contain a dangle reference. To avoid this problem, some systems do not have destroy operations but provide garbage collection mechanism. When an object has no references to it, it is removed from the system by the garbage collector. We think that using one way or another is only an implementation issue as long as a system provides something to remove objects from database.

Insert: Insert operation can be executed by sending an `insert` message to an object-set. Since in our model an object may belong to more than one object-sets or may not belong to any object-set. What exactly the `insert` method does is depending on whether the object to be insert already exists in database or not. If we assume that the object to be inserted already exist and has a name, the following statement adds an object to an object-set.

`anObjectSet. insert (anObject)`

On the other hand, if a user wants to insert a new object, then the arguments should also include the information about the class and initial state of the new object, like in the `create` operation. The following statement first create an new object of `C` then insert it into `anObjectSet`.

`anObjectSet. insert (C, initial-value)`

Insert method is also responsible for checking the integrity constraints and making sure that after adding the object, the state of the database is still consistent.

Remove: To remove an object from an object-set, the `remove` message is sent to the object-set. Also there are two situations. If the object to be deleted is a named object then we use the following sentence:

`anObjectSet. remove (anObject).`

Otherwise, the argument must provide the information which can be used to specify the desired object.

anObjectSet. delete (predicates).

Thus all objects in **anObjectSet** that satisfy the predicates expression are removed from the object-set. The predicates are constructed on the objects' state which is returned by methods defined on the objects.

Modify: To modify an individual object, some modify methods may be applied to change the state of a named object or objects that satisfy a given predicate expression. The following statements deal with these two situations respectively.

anObject. aModMethod (new-values, predicate expression)

anObjectSet. aModMethod (new-values, predicate expression)

Notice that there may be many methods defined in a class to change the state of objects of the class.

4.3.3 Retrieval Operations

Query in Object-Oriented model: Retrieval operations is provided for users to query the desired information from a database. The kind of retrieval operations provided by a system varies from one data model to another. For example, in hierarchical and network model, because records are linked by pointers, the retrieval operations are executed in navigation model. A typical navigational operation is **get-next**. In relational model, the tuples with same attributes form a relation which can be treated as a set. So the retrieval operations contain some set-oriented operations such as **select**, **project** and **join**. There are significant differences between the retrieval operations of relational model and that of hierarchical and network model. In relational systems, operations are applied on set of tuples and a query is specified in a single expression which specifies what we want to retrieve but not how to retrieve it, while in hierarchical and network systems, operations are applied on a single records, also we need to be aware of the position of currency indicators.

These differences mainly are the consequences of the different ways in which data items are related each other. In network and hierarchical models, all the linkages are at individual record level, i.e., pointers connect record occurrences from one to another. This naturally leads to navigational query model. In relational model, there are no links between tuples while linkages are all at the relation level. Two relations are related if they have some common attributes (column). Since relations are sets, linkages at relation level leads to set-oriented query.

By knowing Oid of an object, other objects may directly refer to the object. The fact that Oids are used to access objects provides the connection at instance level. That is why queries in the early object-oriented systems [Maie 86] were solely of the navigation form: a method (or message expression) was applied on individual object and returned an individual object. Query "Find Joe's department chairman" may be expressed as follow:

`joe.workFor.chair`

Assume that method `workFor` applied on `joe` returns `computerDept` and method `chair` applying on `computer-dept` returns `frank`, the query follows the navigational path: `joe -- computerDept -- frank`.

Queries of this kind are not high level, because it is up to the user to specify how to navigate through the database. In order to compete against relational systems which support declarative query language, set-oriented and associative query, an object-oriented model should provide efficient and reliable manipulations of large amount of data. Thus, more recently the focus has been to define associative queries that return sets of objects [Bene 88] [Shaw 89] and not individual objects.

Also, some systems, notably Orion, restrict queries to merely selecting subsets of the objects that are stored in the database. We recognize that although many relationships requested by queries will exist in the existing objects, existing objects in database may not necessarily reflect all possible relationships. In other words, although the purpose of queries is to retrieve information from a database, desired information seldom exists as a whole,

it has to be constructed or tailored. Therefore, relational operations, join and project, still have a place in an object-oriented model.

Our final point is related to the results of queries. In relational model, the answer to a query is a relation. So in object-oriented model, one could think a query returns an object. If we only allow queries to retrieve existing objects, then queries filter a set of objects according to some predicates and the answer is a set containing some or none of existing objects. If we allow queries to create new objects the question arises as what class they belong to? The difficulty of this question lies in the fact that those new objects may have no existing class. Several solutions are possible.

The result of a query belongs to a “universal class” and whose methods allow displaying and printing objects in certain way. The problem of this solution is that a result may not be further processed since there is no method to do so. This is due to the fact that a method implementation is dependent of the structure part (attributes definition) of the class that it is associated with. The universal class accepts arbitrary objects as its instances. So it has no particular structure. As a result, it is impossible to define specific methods to manipulate its objects.

In ENCORE data model [Zdon 86], results of queries are tuples that belong to type **Tuple**. Type **Tuple** is a parameterized type:

Tuple [**<(A₁, T₁), ... (A_n, T_n)>**].

Type **Tuple** can be considered as a meta type and it defines a create operation that takes types **T₁, ..., T_n** as input and return a new type as its output. By default, the created new type has methods **get-attribute-value** and **set-attributes-value** for each attribute **A_i**. For example, when a project operation is applied on a set of employee objects on **name** and **birthday**, the result of the operation is a set of new objects. These new objects belong to type **Tuple** [**<(name:string), (birthday:date)>**] which has four methods. Two for getting values of attributes **name** and **birthday** respectively. Another two for setting value of attributes **name** and **birthday** respectively. Thus, the solution is improved in the sense that results of query may be further processed, because they belong to some

classes and a set of methods is defined on it.

We find there are still some drawbacks in this approach. First, the new class created for new objects in the result of query is an independent class because it has no superclasses and subclasses, i.e., there is no way to locate the new class in the class hierarchy. Second, some behavior information is lost in the new objects. Because Tuple [`<(name:String), (birthday:Date)>`] is a new class and it has no relationship with class Employee, so all the methods defined on Employee are no longer available in Tuple [`<(name:String), birthday:Date>`]. For an instance, a method `age` may be defined on the class Employee. `Age` calculates an employee's age by using his birthday which is an attributes of Employee. Since the result of project operation contain the birthday information, it is desired that the `age` method be available in class Tuple [`<(name:String),(birthday:Date)>`]. In this approach, the behavioral aspect of new objects is seriously restricted.

The best solution is to dynamically create a class for the query results and locate the class within the class hierarchy. So the new class can inherit the methods defined on existing class in the class hierarchy. As a summary of above discussion, the retrieval operations in our model attempt to achieve the following goals:

- While fully respecting object-oriented concepts, such as encapsulation and inheritance, our model supports the formulation of set-oriented queries that can include both value-base (associative) and reference-based (navigational) retrieval approaches.

- In addition to querying existing objects in database, our model also allows the creation new relationship and new objects. Thus join-like and project-like operations are supported.

- Our retrieval operation is a closed one in the sense that results are objects and they belong to some classes. New classes may be created to define the query results so that the results may be further manipulated by the methods defined in their class. Moreover, a new class is located within the class hierarchy so that the new class can inherit from old classes and the

behavior aspect of the new objects may fully captured.

Simple Queries: In the relational model, because only relations have names and tuples have not, relations, sets of tuples, are privileged structure. Consequently, queries are always against relations. In object-oriented model matters are different since object-sets are object themselves. So all data items are on the same level. Thus, although queries against object-sets are necessary, one must be able to query directly on individual objects as long as objects have their names. a query such as “ What is Joe’s major ?” may answered by the following statement (`major` is a method defined on `Joe`):

`joe. major.`

Set-Oriented Operations: Set-oriented operations use object-sets as input and produce an object-set as output. Since in our model, object-sets are objects and they belong to their respective classes, all the set-oriented operations are defined as methods of object-set classes. Also since these operations may be applied on any type of object-sets, we defined them as methods of the most general object-set class, `HOSet`. Three operations are presented as follows:

`object-set1.union (object-set2)`

`object-set1.intersect (object-set2)`

`object-set1.difference (object-set2).`

All of these three operations use Oids as their testing for objects equality. By testing the Oids, duplicate objects are eliminated from the result of `union`; common objects are computed by `intersect` and objects that in `object-set2` are removed from `object-set1` by `difference`.

In most object-oriented models, `object-set1` and `object-set2` must contain the objects of the same class. Since our model support the heterogeneous object-sets, this requirement is not necessary. `Object-set1` and `object-set2` may belong to different object-set classes. Moreover, they may be heterogeneous object-sets themselves.

The class of the output depends on the class of the receiver (**object-set1**) and parameter(**object-set2**). If two input object-sets belong to the same object-set class, then the output is a new object-set which belong to this class too. If either the receiver or the parameter is an instance of **HOSet**, then the output belongs to **HOSet**. If two inputs are both instances of some **RHOSet** or **HomOSet** classes and two classes have no super/sub relationship, the result of union is an instance of **HOSet**, the result of intersect is an empty set and the result of difference is the receiver.

4.3.4. Relational Operations in Our Model

One interesting challenge is to define a retrieval model for object-oriented database systems which is equivalent to relational operations, and at same time honoring all fundamental principles of object-oriented paradigm. We describe **select**, **project** and **join** operations in object-oriented database systems. Since in relational model these operations are applied on sets of tuples (relations), in our model we define them as methods of object-set classes.

Select: Select has a message sending format shown below.

anObjectSet. select (x in aClassSet and P(x)=true)

The semantic of above expression is straightforward. However there are several issues that need to be noticed. Because an object may refer to other objects by sending message, the predicates **P(x)** is not only over the state of **x** but also against the states of objects that may be referred by **x**, and so on. The length of the message sending chain is unlimited. Looking at the following example,

aVehicleSet. select
(v. color="blue" and v.manufacture.name ='Ford Motor Co').

The class **Vehicle** has a method **manufacture** whose return class is class **MotorCompany**. The query predicates not only use the state of **Vehicle** object, its color, but also use the state of **MotorCompany**

object, the name of the motor company. The query predicates follow the message sending pattern. In `V.manufacture.name`, `v` is the receiver of message `manufacture`, and result of this message which is an instance of class `MotorCompany` is, in turn, the receiver of message `name`.

For the select operations, we note that the input of selection may belong to any object-set class, and the result is obtained by picking up objects which satisfy predicates in the input. The result of the select operation belongs to the same class of the input and is a subset of the input. There is no new class created by select operation.

Project: In relational model, project is applied on a relation to get a subset of attributes of the original relations. The result of a project operation is a relation too, which can be applied by other operations. Also the duplicates that result from dropping distinguishing attributes are eliminated. In order to perform a project operation, we need to know what attributes a relation has. This information is included in the database schema. In object-oriented model, the representation (attributes) of objects is encapsulated, what attributes does a class have is unknown from outside. So there is no way to perform the operation that projects on a subset of attributes. Moreover, in the relational model, the state of a tuple is shown in the value of its attributes directly, while in object-oriented model, the state of an object is shown in the results of its methods, and the real value of its attribute is encapsulated.

First issue we need to deal with is how to define the project in object-oriented model so that it behaves similar to relational projection while it still respects the encapsulation principle. Most current object-oriented systems provide project-like operation which project directly on the attributes of the objects. As we pointed before, the attributes belong to internal structure part of objects and is not included in the schema information. Projecting directly on attributes simply violates the encapsulation. Another problem with this approach is that since project operation only carve up the structure of the objects in input the behavior of the objects in the output is not specified. As consequence, there will be no methods defined for them. Our project operation requires a list of methods instead of a list of attributes as its parameter. To simplify our discussion,

we assume the input is a restricted homogeneous object-set.

an-C-RHOSet. PROJECT ($m_1, m_2, \dots, m_n$)

where ($m_1, m_2, \dots, m_n$) is a subset of all methods of C . These methods define the behavior aspect of the objects in the output. For each object of class C in the input object-set, there is a corresponding object in the output object-set. Since the methods which are defined on the objects of C but not included in the project parameter are removed from the objects in the output, the objects in the output do not belong to class C . They are new objects which may not belong to any existing class. Thus a new class C' needs to be created to define those object in the output. Obviously, C' is a superclass of C , as less methods are defined on C' .

Where is the class C' located in the class hierarchy? Suppose the immediate superclass of C is S . If all the methods defined on S is included in the ($m_1, \dots, m_n$) then C' is defined as a immediate subclass of S and a superclass of C , i.e., C' is located between C and S in the class hierarchy. Otherwise, the superclass of S is checked and so on, until we find a class S' such that all its methods are in ($m_1, \dots, m_n$), then C' become an immediate subclass of S' .

What is the internal structure of C' ?, i.e., what attributes does C' have? The most simple solution is to keep all the attributes defined on C . However, since some methods of C are removed from C' , the attributes that are used by those methods are not useful in C' . It is not efficient to keep those attributes that will never be used. Ideally, C' only keeps the attributes that are manipulated by its methods ($m_1, \dots, m_n$). To do this, we need to find what attributes are used by each method of ($m_1, \dots, m_n$). This means that the body of each method needs to be checked. Obviously, it is too expensive if it not impossible. Therefore, we propose an intermediary approach. After C' is located in the class hierarchy, one existing class E will become immediate subclass of C' . We define internal structure of E as internal structure of C' , i.e., E and C' have the same attributes definition. Our approach may be described by the following example.

Example 4.2.

Suppose we have several classes defined as follow:

```
class C:  
  attributes {a1,a2,a3,a4}  
  methods   {m1,m2,m3,m4}
```

```
class E:  
  attributes {a1,a2,a3}  
  methods   {m1,m2,m3}
```

```
class S':  
  attributes {a1,a2}  
  methods   {m1}
```

Their inheritance relationships are shown in Fig. 4.2 (a). After the following project executed,

a-C-RHOSet. project (m1,m2),

class **C'** is created to define the objects in the result. **C'** has two methods projected from **C**. So it is a superclass of **C**. However, the direct superclass of **C**, class **E** has methods {m1,m2,m3} which are not all included in **C'**. **C'** can not be put between **C** and **E**. When we check the direct superclass of **E**, **S'**, we find that **C'** include all **S'**'s methods. So we put **C'** between **E** and **S'**.

(a) class hierarchy before **C'** is created (b). class hierarchy after **C'** is created

Fig. 4.2 An example of the result of project.

As result, C's methods are specified in the project operation, {m1,m2} and its attributes {a1, a2, a3} is defined as same as in its immediate subclass E.

```
class C':
  attributes      {a1,a2,a3}
  methods         {m1,m2}
```

Finally, we need to point that another new object-set class, C'-RHOSet, defines the output as its instance.

Join: In relational model, most join operations often serve to recompose entities that were decomposed for data normalization. Join is based on the comparison of two related attributes in two relations. In object-oriented model, there is no need for many of these joins, because objects can be referred to directly by other objects through message sending.

Example 4.3.

we have two relations shown in Fig. 4.3.

Employee				Department			
Name	Age	Salary	WorkFor	DeptNo	Manager	Location	Goal

Fig. 4.3 Two relations.

To answer following query, "Find the names of department's managers", relational model need a join operation.

PROJECT (NAME){Employee JOIN(ENO=MANAGER) Department}.

In object-oriented model, because object computerDept may refer to its manger, an instance of class Employee by sending message. The join is not needed.

```
aDepartment-HOSet do:{D | D. manager. name}
```

where **do** is an iterative method of HOSet.

Although most relationships between objects are specified by direct reference in advance. Join operation is useful for computing the relationships between objects that were not prespecified.

aC₁-RHOSet. JOIN (aC₂-RHOSet, joinCondition)

The result of a join is an object-set which contains new objects obtained from pairs of objects, one from aC₁-RHOSet and another from aC₂-RHOSet. These pairs of objects meet the requirement specified in the predicate.

Example 4.4.

In relational model, to answer the query “Find students living in a town in which some universities are located”, the join of the two corresponding relations, Student and University, on predicate “University.loc = Student. addr” is required. In our model, this query may be answered by following statement:

**aStdudent-RHOSet. join (aUniversity-RHOSet ,
(x.addr = y.loc)&(x in aStudent-RHOSet)&(y in aUniversity-RHOSet))**

where aStudent-RHOSet that contains instances of Student is an object-set of Student-RHOSet and aUniversity-RHOSet that contains instances of University is an object-set of University-RHOSet.

Since the join operation create new objects like project operation, we need to define what these new objects in the result of the operation are. Basically we have two choices depending on the internal structure and behavior of the new objects. An object in the result might be a pair of objects, one from each input object-sets, like the result of set-theoretic product. This approach is used in [Shaw 89], where a joined object consists of two input objects. In relational model, when a relation of n-tuples is joined with a relation of m-tuples the result is relation of (n+m)-tuples, in order to retain the closure property of the operation. This argument no longer holds in object-oriented model since we have a richer structure so

that a pair of objects is still an object. However, a set-of-pairs semantic has the disadvantage that joins would not be associative, that is, $(R \text{ join } S) \text{ join } T$ would be different from $R \text{ join } (S \text{ join } T)$. This would exclude some significant optimization means in a way similar to relational algebra.

In our model, we propose a flatten semantic, where an object in the result is obtained by “flattening” the pair and combining the attributes and methods of two input objects. In the student-university example, after join a new class **Stu-Univ** is created to define the objects in the result. Class **Stu-Univ** is specified by the attributes and methods defined in the class **Student** and the attributes and methods defined in class **University**. Since class **Stu-Univ** combines the structure and behavior of classes **Student** and **University**, it inherits the definitions of **Student** and **University** and is a subclass of both of them. Hence the multiple inheritance is needed.

Similar to project operation, another new object-set class **Stu-Univ-RHOset** is created. The result of the join operation is an instance of **Stu-Univ-RHOset**.

4.3.4. Meta Operations

So far we have described operations applied on the objects of existing classes. Since our model support the creation of new classes, operations at the meta level are required. In our model, meta data and data are treated uniformly, that is, classes are objects themselves and are defined by a meta class, **MetaClass**. All classes are instances of **MetaClass**. A class method of **MetaClass**, **createClass**, is defined for creating a new class. A new class may be created by following statement.

MetaClass. createClass (newClassSpecification)

The specification of the new class is provided as the parameter of **createClass**. The specification includes attribute definitions, method definitions for the new class as well as the position of the new class in the class hierarchy.

In previous section both **project** and **join** operations need to create new classes. During these operations, a message **creatClass** with parameter is sent to **MetaClass** to create a new class.

Chapter 5. A Unified View of Database Systems

5.1 EVOLUTION OF DATABASE SYSTEMS

The roots of database technology lie in file management, particularly in the index-sequential file organization. In the early seventies, two main data models arose: the hierarchical model and the CODASYL network model. The network model was developed by Data Base Task Groups of the CODASYL Systems Committee and was heavily influenced by the COBOL programming language. Two notions were introduced in the network model: *records* and *sets*. Records basically are the aggregations of attributes that describe the entities to be modeled. Sets are used to model the *member-of* relationships, 1:n relationships among the records. By using linkage records, the n:n relationships can be modeled as well. Within a set, the owner record and the member records are connected by pointers. To answer a query, the user navigates through the records following these pointers. The concept of *currency* supplies the necessary support. The hierarchical data model was largely the result of a IBM product, IMS. In the hierarchical data model, the data is represented as records whose attributes may contain set-valued attributes. In this way any hierarchical model can be considered as a special case of network model, thus it is ignored in our discussion.

Both models were soon overcome by the relational model which is based on a formal theory, relational calculus and relational algebra. The most striking feature of the relational model is its simplicity. The only notion introduced in the relational model is *relation* which can be thought of as a table. Each row of a table represents an entity to be modeled and columns of the table denote the attributes of the entities in the relation. On those relations a set of operations is defined to retrieve and update the data in the database. The first relational database system, system R was implemented at IBM in San Jose and a declarative query language, SQL, was introduced which has become a standard for relational query language.

Currently, relational systems with SQL are a quasi-standard for commercially available database systems.

However, starting with the relational model, the research and development of database systems seem to diverge in different directions, namely the nested relation model, semantic model and object-oriented model. These three models were proposed to support the non-traditional database applications. Nested relational model [SCHE 86] is obtained by giving up the first-normal-form condition in the flat relational model, i.e., allowing relations as the values of attributes of a relation. Adding the generalization/ specialization concept into semantic data model seems to increase the divergence. Finally the object-oriented model introduces another set of concepts from the object-oriented paradigm, such as encapsulation and behavior of objects.

It appears very difficult to obtain a unified model which contains the existing models as special cases. However in this chapter, we try to show that the seemingly different concepts are not really incompatible. Although the definition and the terminology for these models are different, we try to explain them in a consistent way. More specifically, we try to establish our object-oriented data model as a more general data model that contains the relational and network models as its special cases. In order to explain this we describe two evolution paths: from relational model to object-oriented model and from CODASYL network model to again object-oriented model. As a result, we can consider the object-oriented model as a synthesis of its predecessors.

5.2 FROM RELATIONAL MODEL TO OBJECT-ORIENTED MODEL

In this section, we describe the evolution starting with relations and in the following section we describe the evolution where the network model is used as the starting point.

The only type constructor in the relational model is relation. Relations are sets of tuples and tuples are made of a set of atomic types. We can consider the relation constructor as a tuple constructor followed by a set

constructor. However, unlike semantic model, the tuple constructor and the set constructor in the relational model are not considered as two separate constructors. They are always applied together and in that order. Further, the first normal form condition requires that the tuple-components (that is, the attributes of a relation) be atomic data types. Therefore, the relation type constructor can only be applied once on a relation.

Example 5.1

DeptRel=Relation	EmpRel=Relation
deptNo: integer	empNo: integer
deptName: string	name: string
chairman: empNo	salary: real
staff: empNo	workFor: deptNo

In the definition of DeptRel, in order to be a INF relation, it would have to contain empNo, the key of EmpRel, as a foreign key. Thus, all values of the attributes are atomic data types.

The nested relational model [Sche 86] is obtained from the flat relational model by discarding the INF condition, i.e., allowing relations as values of attributes. However, in the nested relational model, only certain structures can be the value of attributes. The values of attributes are still restricted as either atomic types or another relation. So as in the flat relational model, set and tuple constructors are always applied together and strictly alternate.

Example 5.2

DeptNRel=nested relation	EmpNRel=NestedRelation
deptNo: integer	empNo: integer
deptName: string	name: string
chairman: empNo	salary: real
staff: EmpRel	workFor: deptNo

Since nested relations allow relation-valued attributes, the value of attribute staff is another relation, EmpNRel, instead of just a foreign

key.

If we further drop the restriction that set and tuple constructors have to strictly alternate and allow the recursive schema definition, we obtain the so called the extended nested relations.

Fig. 5.2 Constructors Graph

Example 5.3

DeptENRel=ExtNestdRelation

EmpENRel=ExtdNestedRelation

deptNo: integer
deptName: string
chairman: EmpTup
staff: EmpENRel

empNo: integer
name: string
salary: real
workFor: DeptTup

From nested relations to extended nested relations, there are two points we need to notice. First the set and tuple constructors are not necessary strictly alternative. For instance, the value of attribute chairman in

DeptENRel is neither an atomic value nor another relation but a *tuple*. Thus in the definition of DeptENRel, two tuple constructors are applied consecutively. Second, when looking at the definition for relational or nested relation schema. We notice that in order to be well defined, these relations have to be non-recursive, i.e., the nested relations form a hierarchical structure. However, in Example 5.3, we define EmpENRel to contain an attribute workFor of type DeptTup. This would certainly be nice as to provide a fast access from an employee to his/her department. Then the type definition is recursive. We call the attribute such as workFor as a *reference attribute*. In functional model, such as IRIS and OODAPEX, in addition to attributes, functions are introduced for this purpose. In other contexts, this would be called "pointers", "references", "object-identifier", etc. Supporting such recursive relationship extends the hierarchy of relations to a general network of relations. This is an essential movement. Fig. 5.2 explains the constructors in relations, nested relations and extended nested relations in the graphic notation from [Hull 87].

The generalization/specialization is an important concept found in both the semantic data model and the object-oriented model. Defining a new type (class) as a subtype (subclass) of an existing type is a convenient way of reusing type definition. It also provides a way to model the IS-A relationships among the entities in applications. As a result, the subtype inherits the description of the supertypes and may define more methods and attributes on it instances. Up to now, we have a database definition that can be called structural object-oriented. However, object-oriented model also contains the behavioral abstraction mechanism, methods and encapsulation. We will discuss this separately in 5.4.

5.3 FROM NETWORK MODEL TO OBJECT-ORIENTED MODEL

5.3.1 Structural Comparison

We describe second evolutionary path starting with the network model. As matter of fact the strong relationship between the network model and object-oriented model has been already pointed out, e.g. [Ullm 88]. Here

we describe the network model in such way that it may be considered as a special case of the object-oriented model.

First of all, network model is much closer to object-oriented model in the sense that the concept of object identity is already introduced in the terms of database key. So they are both called *identify-based* system.

Records are used for the description of entities and may be themselves complex structures, that is, the attributes may consist of other lower level attributes. This allows a network records to model hierarchical objects. However, each record is defined independently, i.e., a record type can not be referenced in the definition of other records type. A 1:n relationship is captured in a DBTG-set, which contains a owner record and a number of member records.

Now if we reinterpret the definition of network schema, it looks pretty like the definition of objects. Records correspond to objects. An attribute of a record, may be interpreted as an attribute of atomic type of the object. If a record participates in a set as the owner, its members may considered as a set-valued attribute of the object. If a record participates in a set as a member, then its owner (if defined) may be interpreted as an attribute whose domain is another type, type of the owner.

Example 5.4

Suppose we have two records defined as follow.

Record	A:	Record	B:
	A1:		B1:
	A2:		B2:
Set	AB:	Owner	A
		Set	AB

We can reinterpret these two records in object-oriented terminology. In object model, A is an object which has two attributes of atomic data type, A1, A2 and one set-value attribute AB. B is an object too which has two atomic attributes B1, B2 and a single-valued non-atomic attribute A.

From above perspective, network model is a special case of object-oriented model where the values of attributes for an object are either single-valued primitive objects (normal attributes) or set-valued non-primitive objects (members) or single-valued non-primitive objects (owner).

Of course, an IS-A relationship between record types is missed, so the notion of inheritance is not supported. Also, direct recursive DBTG set definition is not allowed in the network model, i.e., a record type can not be defined as both owner type and member type in the same DBTG set. Therefore, from the structural point of view, the object-oriented model has the following additional features:

- The values of attributes may be arbitrary combination of set-valued and single-value to primitive and non-primitive objects.
- Direct recursion is allowed.
- Generalization/specialization is supported.
- Directly model n:m relationship.

5.3.2 Operational Comparison

Let us look at in a network database system to see how the operations are defined. All the operations on the network database such as retrieving, creating, and modifying of records are performed through the currency indicators. Currency indicators are predefined variables with fixed names, and they are pointers pointing to record instances. For example, the current-of-run-unit (CRU) is a variable of a type which is the union of all record types and it points to the most recently visited record. Also a current-of-set currency holds the most recently visited record within a set, either the owner record or one of the member record, while a current-of-record currency points to the most recent record of a record type.

The operations provided by a network system include **STORE** for

adding a new record occurrence into database, **ERASE** for deleting a record occurrence from database, **FIND/GET** for retrieving a record and **CONNECT** or **DISCONNECT** for inserting or removing a record into or from a **DBTG** set.

We briefly describe the behavior of these operations. A new record is created outside of database, then stored by **STORE** operation. The result of **FIND** operation is a single record which is pointed by **CRU**. When we want to retrieve a record from database, we have to first apply several **FINDs** in order to set **CRU** to the desired record. Then a **GET** operation put the record in the user working area with the representation that user understand. We need to notice two things here. First, **FIND/GET** only retrieve a single record. There are no set-oriented operations. If we want to see more records, we must repeat the **FIND/GET** procedure for every instance. Second, **FINDs** cannot be nested.

Compare to the operations in object-oriented model, we find that some essential features are similar in these two models. Network records are accessed by predefined functions, mostly for navigational access. The navigation paths follow the various predefined links (pointers) that connect records of the same type, or member records to their owner record, or member records within a set. So users are able to navigate either from one record to another of the same record type, or from a record of one type to a record of another type. In object-oriented model, objects are related in two ways. First, objects may be grouped by object-set mechanism and the iterator over an object-set is provided for iterating the objects in the object-set. The object-set may be either homogeneous, where all the objects in the object-set belong to the same type, or heterogeneous, where the objects in the object-set may belong to different types. Second, objects are also connected by their identities. The identity of an object may be stored in another object, so the latter can refer to the former one. This provides user with another vehicle to move one object to another. Moreover, in network model, **CONNECT** or **DISCONNECT** are used to add or remove records into or from an object-set. As in the object-oriented model, **INSERT** and **REMOVE** perform similar functions.

It is obvious that object-oriented model has introduced significant

improvements in that:

- Instead of using limited predefined pointer variables. An arbitrary number of object variables with names defined by user can be used.
- Set-oriented operations are provided.
- Functions returning a single object can be nested.

5.4 FURTHER DISCUSSIONS

In the previous sections, we described two evolution paths of object-oriented data model. Our goal is to establish our object-oriented data model as a unified model for existing data models. However, we have avoided discussing some concepts which are obviously not compatible between different models. These issues include value-based vs. identity-based systems, attributes vs. functions. In this section we try to explain how these incompatible concepts can coexist in an object-oriented model. Also some concepts seem to be unique to the object-oriented model, such as, behavior abstraction, encapsulation, state of objects. We attempt to find their correspondences in conventional models.

5.4.1 Value-Based and Identity-Based Systems

The relational data model is value-based, as opposed to network model, which could be called identity-based. The concepts arise from the mechanism that a data model adopts for relating objects to each other. A value-based system expresses a relationship between objects by embedding the same (or related) value in two or more objects. An identity-based model can relate two or more objects independently from their embedded values.

As we described before, in the hierarchical and network models, the only way to access the records in the database is by means of CRU which holds the identity of the most recent visited record. (In the network model,

2

of/de

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

PIONEERS IN METHYLENE BLUE TESTING SINCE 1974

METED
LABORATORIES

1000 COUNTY ROAD 8, BURNING WOOD, OH 44027 USA
TEL 612 428 7967 FAX 612 428 7967 TLA 510000000

a record's identity is called database key). So the database may be navigated by moving CRU. If a user wants to access a record, he must first set CRU to the desired record, then a subsequent GET operation dereferences CRU pointer and provides the user with the record. The retrievals are based on the identities of records. We call such system an identity-based system.

In the relational model, there is no concept of identity and objects are identified by their values. A tuple is identified by a subset of its attributes, called its key. This approach leads to simpler computational structure, because users may use a value-based expression to specify the object he want. Then the DBMS locates the object by the attribute values given by the user in the query expression. However, the value-based approach has some limitations. First, theoretically, there is no way to distinguish two objects with the same values for all attributes. That is why in value-based model, every object must have a unique key value. Second, if the key value of an object changed, then other objects that refer to it by its key value would no longer refer to it. For example, if a student changes his `studentNo` which is used as the key of the relation `Student`, then changes have to be made too in the relations where the `studentNo` is used as foreign key. Otherwise, the database would be inconsistent.

Object-oriented data model on one hand, is closer to the network model, in the sense that it has the ability to make references through an object identity which remains invariant across all possible modification of object's value. The identity-based models have no such problems as in the value-based model. Objects may be identified by their identities. So two objects are distinguishable even when they have the same values for all their attributes, because they have different identities. Also since the identity of an object is immutable, there are no operations that can ever change the identities. Therefore, an object's attribute values may be changed without effecting the objects that refer to it.

Although object-oriented databases are based on the notion of identity, it is interesting to note that the value-based concept still has a place in the object-oriented model. Note that the way in which objects are related may or may not be based on the identity. References to an object by using its identity is much like pointers in conventional models, such as network

model. When object O₁ needs to refer to object O₂, we may store the identity of O₂ in O₁. Since access to O₁ must be through the methods defined on it, computing the association between O₁ and O₂ must be done by a method. This method uses the O₂'s identity to access O₂. But this is not the the only way to connect O₁ and O₂. We may also store such values that a method of O₁ may use these values to allocate O₂. For example, a student's advisor may be referenced by storing advisor's identity in the student object. However, we may also store some values about the advisor, such as his SIN or employee number. The method may use this value to locate the advisor object. Obviously, the values stored in the student object play the role of key in the value-based model. In this way, the object-oriented model provide a framework for unifying value-based and identity-based model.

One related question is that can we consider a primary key in a relational database as the same as an object identity? The answer is no. Primary keys are only unique within a single relation while object identities are unique across the whole database. The unique identity also make it possible for an object-oriented model to manipulate heterogeneous object-sets.

5.4.2 Function, Behavior and Value

In relational and network models, only a few general operations are defined. These operations may be applied on objects of any types. There are no functions defined on individual record type or relation. Therefore, in relational model attribute names may be used directly in predicates of query, i.e., the values of attributes are directly accessible. In object-oriented model, arbitrary functions may be defined on objects and each object type has its own set of functions (methods). The internal state(the value of attributes) of an object can only be accessed through the methods defined on the object. Thus the values of attributes are protected and the attribute names can not be directly used in query expressions. Although these two frameworks look very different on the surface. Our argument is that by appropriate reinterpretation, two frameworks are based on the

same essential principle.

First of all, we may consider attributes of a relation as unary functions on tuples of the relation. A relation may be seen as a table which is formed by tuples (rows) and attributes (column). For a tuple, given a attribute name, the value of this attribute found from the table according to the column where the attribute is located.

On the other hand, in object-oriented model, the state of an object may be obtained by applying a set of methods, some of these methods just simply return the values of corresponding attributes. For example, `p.getName` returns the value of attribute `name` of object `p`. However, the representations of stored `name` and returned `name` may be different. Other methods may return derived attributes which are computed from other attributes. A person's `age` may be computed from attribute `birthDay` by method `age`. These methods are unary functions, because they only have one argument, that is, the receiver object. We can reinterpret each of these functions as a binary relation. For example, the function `name` defined on class `Person` is a unary relation whose tuples are all the instances of `Person` and the first column is the identities and the second column is the `name` attribute of `Person`.

Now we make some discussion on the multi-argument functions. Multi-argument functions are such functions that have more than one argument. Consider `courses(student, subject)` which is defined on class `Student`. It returns the courses in `subject` taken by `student`. `courses(peter, physics)` returns the physics courses taken by `peter`. This function may be represented in the following way. It is a table in which for each student `S`, there is a list of subjects, and for each subject there is a list of course taken by `S`. Such a table is essentially a relation with relation valued attributes, i.e., a nested relation.

To summarize, unary functions may be represented as a binary relation while multi-arguments functions may be represented as nested relations. We can represent the behavior functions as values. Since the behavior of an object is specified by its methods, we provide a way to convert behavior of objects in object-oriented model to attribute value in relational model. In

addition to providing a consistent way to look at attributes and functions, values and behavior, our reinterpretation of function also make it possible to manipulate methods as values. This is similar to the concept of *data function* introduced in the language COL [ABIT88].

5.4.3 Encapsulation and View

Encapsulation is also a concept that is unique to object-oriented model. By adding encapsulation facility the values of attributes of an object are hidden from outside. Users only see the results returned by the methods applied on the object. These results are obtained by the manipulation of the values stored in the attributes of the object. We found it is very similar to the concept of view in relational database, where views are virtual relations defined on the real relations. Thus we consider the encapsulation facility as just a refinement of the view facility.

Suppose we have an object-oriented database, the definition of the structural part of all classes form the *structural schema* which includes the attribute specification for each class. Then the methods for each class are defined. These methods manipulate the attributes and return the result that may be seen outside. We call the definitions of all the methods as *behavior schema*. Since methods are defined by the formulas in terms of attributes, we say that the structural schema implements the behavior schema. Moreover, since the structural schema is hidden from users by behavior schema, we say that the behavior schema encapsulates the structural schema. Thus users only see the behavior schema and they may believe that this is the real database. From this point of view, the schema of traditional database is the structural schema of an object-oriented database.

5.5 A UNIFIED VIEW

We have shown that our proposals for an object-oriented data model can be considered as an evolution from the relational and network models.

By giving-up the first-normal-form condition and allowing relations as values of attributes, the flat relational model become nested relational model. Moreover, if tuples are allowed as values of attributes we receive the general model of complex objects. These models are able to describe arbitrary hierarchical structure. Evolution from hierarchical structure to network structure is achieved by introducing reference attributes (pointers to other objects). This also makes recursive schema definition possible. An additional essential ingredient of object-oriented model is the support of generalization/ specialization relationship among the objects.

A similar evolution can be observed when the classical network model is used as the starting point. Structurally, the network model already directly supports the complex objects and it supports the concept of object identity too. In that respect the network model is closer to the object-oriented model. On the operational aspect, records are accessed through a predefined variable, CRU, and since records are connected by pointers the relationship between objects may be computed by chasing those pointers. This may be treated as special operations in object-oriented model.

Finally, we made same discussions on the concepts that look very different on the surface. Issues include value-based vs. identity-based, attributes vs. states, behavior, methods vs. values. We reinterpret these concepts and provide some degree of compatibility between those concepts. As a result, our object-oriented data model is obtained by integrating relational and network data model ingredients. We are convinced that an object-oriented data model can play a unifying role and can be regarded as a new high level data model standard for future database systems.

As a result of these discussion we see how our object-oriented model can be obtained as a synthesis of well-established concepts, namely, (i) set-oriented, declarative query from relations and nested relations, (ii) definition for complex objects and Oids from the network model, (iii) behavior abstraction and encapsulation from abstract and functional mode' and (iv) inheritance and IS-A relationship from the semantic model. The practical aspect of such synthesis is that classical data models are contained in object-oriented data model as special cases.

Chapter 6 The Simulation

6.1. SYSTEM ARCHITECTURE

In this chapter we present a simulation of the core part of the data model described in Chapter 4. The main purpose of our simulation is to demonstrate the basic features of the data model. Our simulation includes three parts, the metadata mechanism, object management facility and stored objects. We discuss the rationale behind our implementation and the alternatives we considered. This chapter is organized as following. First we give the architecture of our simulation which consists of three components. Then we describe each of these three parts with some details. Finally an example of database application in university environment is presented.

Our prototype is written in object-oriented programming language C++. C++ supports most features of object-oriented concepts such as, encapsulation, complex objects and inheritance (it supports multiple inheritance). As a strongly-typed language, C++ supports the static type-checking. In C++ all data items, objects, variables even pointers and references, are associated with type information. The type information associated with objects are used for run-time binding of a message to the proper method, whereas types of pointers and references might be used for enhancing efficiency since the type of an object that pointed (referenced) by a pointer (reference) can be checked without accessing the object itself. Variable types are used for type-checking at compiling time. Moreover, C++ has flexible encapsulation mechanism. For a class, its members, both structure members (attributes) and functional members (methods), may be declared either private, public or protected. This well-controlled accessibility is beneficial for database applications. Finally, instead of treating everything as object C++ provides us a nice base to distinguish values and objects. Our prototype consists of three component modules shown as Fig.6.1.

Fig. 6.1 System architecture

Fig. 6.2. The class hierarchy in the simulation

Metadata module provides the mechanism to store the descriptive information of the data stored in database. Object management module supports basic object management functions, such as organizing and manipulating the stored objects as well as query processing. Stored data contain all instance objects in an application. These objects are grouped in object-sets. All these three modules are implemented in class format. The class hierarchy is given in Fig.6.2.

6.2. METADATA MECHANISM

Generally the function of metadata (system, catalog) of a database system is to store the descriptions of database that DBMS maintains. The metadata includes a description of the database schema on conceptual, internal, and external level, and any mapping between the schemas at different levels. It also store the information need by specific DBMS modules, such as, rules for query optimization and information on security and authorization. In our simulation we only implement the kernel part of metadata, that is, the description of the stored data in database.

What information should be stored as the metadata of an object-oriented database? Since in our object-oriented model, application entities are modeled as objects which are defined by their classes, the stored data are the instances of classes. Therefore the meta information of stored data is the descriptions of those classes.

The basic information that must be stored as meta data of a object-oriented database is the description of classes including class names, attributes and methods as well as superclasses / subclasses relationships. For an attribute of a class, important information are attribute name, attribute domain which is another class, and its owner. For a message, useful information includes message name, receiver domain of the message, and the domain of its return-value.

A high-level Enhanced Entity-Relationship (EER) diagram that describes the main part metadata for an object-oriented database is shown in Fig. 6.3. In Fig. 6.3, we also show the authorization information. Although it is not included in our implementation, it is crucial information for any practical systems.

Fig. 6.3 The E-R diagram of the metadata

As we mentioned before, metadata should have the same representation as stored data. We present metadata in object-oriented format too. So we can use our object management module for querying, updating and maintaining the metadata.

As shown in Fig. 6.3, entity type "class" is implemented as an abstract

class, called **MetaClass**. **MetaClass** is used as baseclass for deriving its two disjoint subclasses, **UserClass**, and **PrimiClass**. Here is the definition of **MetaClass** in C++ syntax. We omitted some details such as methods for some routine works, such as **printOn()** which print the object in the printable format, **isA()** which returns the class of receiver object, etc.

```
class MetaClass: public Object
{
    String&          className;
    MessageSet&      methods;
    ObjectSet&:      allInstances;
public:
    Metaclass(String&, Set of Message&); // the constructor
    getClassName();
    getMethods();
    getAllInstances();
}
```

Class **PrimClass** has no more members than its superclass, **MetaClass**. The only difference is that **PrimClass** is not an abstract class, so it has instances. In C++ there are two primitive type **Integer** and **Character**, although **String** can be treated as array of character, for convenience, we treat **String** as primitive type too. We represent these three primitive type as three instances of **PrimClass**.

Another class derived from **MetaClass** is **UserClass** whose instances store the meta information of classes other than primitive class. Additional to the members inherited from **MetaClass**, **UserClass** also has members **superclass**, **subclasses** and **attributes** to hold additional information about a user defined class. Its definition is shown as follow:

```
class UserClass: public MetaClass
{
    MetaClassSet& superclass
```

```

    MetaClassSet& subclasses
    AttributeSet&  attributes
public:
    getSuperclass();
    getSupclass();
    getAttributes();

```

Class **Attribute** describes the attributes appearing in all classes. The information needed to store includes attributes name, attribute domain and owner. **Attribute** is defined as follows:

```

Class Attributes: public Object
    String&          attributesName;
    Metaclass&       owner;
    Metaclass&       domain;
public:
    getAttributesName();
    getOwner();
    getDomain();
}

```

Similarly, class **Message** is defined as:

```

Class Message: public object
{
    String&          selector;
    Metaclass&       receiver;
    Metaclass&       resultDomain;
public:
    getSelector();
    getReceiver();
    getResultDomain();
}

```

See Appendix, *Metadata Class Definition and Methods Implementation for Metadata Classes*, for complete definition and implementation for the metadata module.

6.3 OBJECTS MANAGEMENT

Since our prototype is just a simulation, we won't deal with persistency issue. Therefore, our system is memory-resident and all objects are allocated in memory. Our work focuses on objects organization, object manipulation and query preprocessing.

6.3.1 Organizing Objects in Object-Sets

All objects in a database are grouped into object-sets. As we pointed in our data model, these object-sets are modeled as objects too, they are defined by their classes. In our system, for each class **C** there is a corresponding class **C-RHOSet** whose instances contain the objects that belong to classes in the subhierarchy rooted by **C** and a corresponding class **C-HomOSet** whose instances contain the objects that belong to class **C**.

We introduce the concept of ownership. As in real life, an object-set starts out empty and is filled with objects. Once objects are placed in an object-set, they are owned by the object-set. If this were not the case, one could add an object to an object-set then destroy it without the object-set knowing that the object had been destroyed. This would leave the object-set in a very confused state. Thus when an object-set owns an object you must not destroy the object without first removing that object from the object-set, or you can ask the object-set to destroy the object for you.

All the object-set classes are derived from the class, **ObjectSet** while **ObjectSet** is derived from class **Set** which mainly does follows:

(i) provide a way of gathering objects together and operating on them. Related methods include:

```
add(Object&);           //add an object to a set.
destroy(Object&);       //remove an object from a set and destroy it.
detach(Object&);        //remove an object from a set without destroying it.
```

(ii) provide a method of iterating through the objects in set.

forEach(actionPtr, parameters); //parameters be passed to the iterating
//function pointed by actionPtr.

Iterating function is applied on every object in the set.

(iii) define a method for displaying the objects in the set in a formatted way.

PrintOn(OutputStream&);

We describe more object manipulation functions of **ObjectSet** in following section.

6.3.2 Object Manipulation

In addition to the properties inherited from **Set**, **ObjectSet** needs more capabilities to manipulate objects stored in database. Here are the functions we add into class **ObjectSet**.

Membership Checking: Since an object-set only contains objects of certain classes, before an object can be added into an object-set we have to check which class is the one that the object belongs to. We override the method **add()** of class **Set** in every object-set class. (Method **isA()** returns the class which the object belongs to.)

Insertion: Method **insert** adds an exiting object into an object-set.

insert (anObject)

Deletion:

delete(conditionPtr, parameters)

Methods **detach** and **destroy** of class **Set** can be used to remove an object from an object-set. However, to use these methods we have to have a reference of the object to be deleted, since the methods need the reference as parameters. However, in most cases of database applications, most objects don't have names. So it is desired to delete objects which meet certain conditions. Method **delete** does this by checking each object in an

object-set, whenever an object meets the condition pointed by conditionPtr, it is removed from the object-set.

Modification:

modify(conditionPtr,modifyPtr)

Method **modify** iterates over an object-set. If an object satisfies the condition specified by the function pointed by conditionPtr then the state of the object is modified by function pointed by modifyPtr. Since the state of objects are encapsulated and can not be accessed directly, modify function always invoke the methods of the object to modify the state.

6.3.3 Query processing

One big advantage of relational model over older models is the capability of allowing users to express queries in a declarative form without concerning for physical organization of data. This is accomplished by introducing set-oriented operations in query processing.

Indeed, a major criticism of current object-oriented database systems is that the query processing executed in the way of pointer chasing and it takes us back to the days of navigational systems. Object-oriented languages, like C++, typically do not provide set-oriented processing capability. One goal of our system is to provide set-oriented processing capability similar to the those found in relational query languages.

Current object-oriented systems are “reference” oriented, that is, the relationship between two objects is established by embedding the identity of one object in another. When we design a database schema it is impossible to envision all relationships between the objects. So with lack of capability to compute arbitrary “join” operation in relational model, some unpredifined relationships can not be queried in object-oriented database. We also correct this deficiency by providing “join” operation.

Select operation: For each user-defined class C, there is a corresponding

object-set class **C-HomOSet**. The method **select** of **C-HomOset** returns an object-set which contains objects that satisfy given condition. Receiver of **select** may be any instance of **C-HomOSet** and the result is also an instance of **C-HomOSet**. Method **select** has the form:

```
aC-HomOSet& select (conditionPtr parameters)
```

The iterating function **condition** pointed by **conditionPtr** has prototype:

```
int condition (Object&, void *)
```

The iterating function checks each object in the receiver. If an object satisfies certain conditions the function returns 1 else it returns 0.

Example 6.1

```
int lengthCheck (Object& o, void*)
{ if (((String&)o).length()==aLength)
    return 1;
    else return 0;
};
condFuncType lengthCheckPtr=lengthCheck;
aString-HomOset.select(lengthCheckPtr, aPointer);
```

The above query returns a subset of **aSetOfString** and all strings in the result set have length **aLength**.

Project operation: Project use a function which specifies a subset of all methods defined on the objects in the receiver to create new objects of a new class.

```
project (prjectFunction, paramter*)
```

Since the dynamic class creation is not implemented in our simulation, objects in the result of a project operation belong to the root class, **Object**, and the result object-set is an instance of class **ObjectSet**.

Join operation: Join operation is used to create relationships between

objects from two object-sets. The operation creates new tuples to hold the generated relationships.

`objectSet1 join (objectSet2, joinCondition)`

The implementation of `join` in C++ is a double loop which examines all pairs of (o_1, o_2) such that o_1 in `objectSet1` and o_2 in `objectSet2`. Since the class creation is not implemented, for a pair if `joinCondition` is satisfied then the pair is considered as a new object which belong to the root class, `Object`. The complete definition and implementation of object-set classes are show in Appendix, *Object-Set Classes Definition and Methods Implementation for Object-Set Classes*.

6.4 AN EXAMPLE

In this section we give a simple database example in university environment.

6.4.1 Enhanced E-R Diagram

Consider a university database that keeps track of students, their majors and registrations, as well as the university course offerings. The database also keeps trace of the faculty and department information. The EER diagram of this schema is shown in Fig. 6.4.

For each person in the database, it is required to keep information on the person's name, social insurance number, address, sex and birth data. Two subclasses of the `Person` entity type are `Faculty` and `Student`. Additional attributes of `Faculty` include rank (assistant, associate, professor, etc.), office phone, and salary. Each student has an additional attribute student number. An academic department has the attributes department name, telephone, and main office. The attributes of `Course` are course number, course name.

Let us look at the relationships among these entity types. We related

each faculty member to an academic department which the faculty member works for by relationship **belong**. The reverse relationship of **belong** is **hasFMembers**. Relationship **major** relates a student to his or her major department, and relationship **advisedBy** relates to the faculty member who is his or her supervisor. The reverse relationships of these two relationships are **hasStudents** and **supervise** respectively. Additionally there is a many-to-many relationship, **std-crs**, between **Student** and **Course**. Notice two of the relationships have association information, i.e., **start-data** in **adviseBy/supervise** and **grade** in **std-crs**.

Fig. 6.4 The EER diagram

6.4.2 Transformation from EER to Object-Oriented Format

Now we convert the EER into object-oriented format. Although the conversion is oriented around our example, we think it has wider applicability.

Suppose we have three primitive classes **Integer**, **Character** and **String**. We use these three primitive classes as primitive constructors, i.e., they are leaves in the class composition hierarchy. Also some enumerated types are used as primitive types, such as **rank{assistant-prof, associate-prof, prof}**, **sex{male, female}** etc.. Actually, in C++ enumerated type is **Integer**.

For each entity type in the EER diagram in Fig. 6.4, we introduce a class to represent it. So the super/sub relationships in EER are preserved. For each attribute in an entity type there is an attribute in corresponding class. According to the discussion in our data model, we represent relationships that have no association information by embedding the references of objects in the attributes of another object. For example, one-to-many relationship **belong** is represented by embedding the reference of a **Dept** object in attribute **worksFor** of the **Faculty** objects that work for the department. On the other hand since relationships **advisedBy/supervise** and **std-crs** have extra information, we introduce additional classes to represent them. Class **Advise** has three attributes, **student**, **supervisor**, **startDate**, and class **Std-Crs** has attributes, **student**, **course**, and **grade**. The class graph for this example is shown in Fig. 6.5.

Notice that in Fig. 6.5, in order to distinguish the internal classes from the normal classes we use rectangles instead of ovals to represent internal classes **Advise** and **Stu-Crs** and a dash arrow represents the linkages between an internal class and its friend classes. For each class, its corresponding object-set class is introduced for containing the instances of the class. Thus we have object-set classes, **StudentSet**, **DeptSet**, **FacultySet**, **CourseSet** and **Crs_StuSet**. See Appendix, *Application Classes Definition and Methods Implementation for Application Class*, for

complete definition and implementation for these classes.

Fig. 6.5. The class graph of the example

6.4.3 Creation of Application Data and Query Examples

After all classes are defined, we create instances of these classes as stored data in our simulation. Our data consists of two categories of objects, the instances of metadata classes and the instances of application classes.

Metadata Part: As we described, the metadata is described by five classes, **Attribute**, **Message**, **MetaClass**, **PrimClass** and **UserClass**, where **PrimClass** and **UserClass** are two subclasses of **MetaClass**. A function, see Appendix, *Function for Creating Metadata*, is written to generate the instances of those metadata classes. Since **MetaClass** is an abstract class, it has no instances. Three instances of **PrimClass** are created to store the meta information of classes, **Integer**, **Character** and **String** and ten instances of **UserClass** are created to store the meta information of user defined classes, **Object**, **Person**, **Student**, **Faculty**, **Dept**, **Course** and their corresponding

object-set classes. The meta information of attributes and methods of all the classes are stored as instances of class **Attribute** and **Message** respectively.

Application Data Part: Similar to the metadata part, a function, see Appendix, *Function for Creating Application Data*, is written to generate the instances of application classes. Three instances of **Dept**, five instances of **Faculty**, ten instances of **Student**, six instances of **Course** and twenty-two instances of **Crs_Stu** are created by the function. Also an instance of each object-set class is created. Since these object-set instances are used as the start points of queries, each of them is given a name, **oSetS** for the instance of **StudentSet**, **oSetD** for the instance of **DeptSet**, **oSetF** for the instance of **FacultySet**, **oSetC** for the instance of **CourseSet** and **crsStuAssoc** for the instance of **Crs_StuSet**.

Query Examples: We give some examples of queries that are against the database we created. Since in our simulation only object-set objects have names while individual objects don't have names, all queries are against object-set instances, i.e., object-set objects are the receivers of the query statements.

We start with a simple query.

Query 1: "Find students whose advisor is lives in Houston"

```
oSetS.select(advHoustonPtr, aPointer)
```

```
where
```

```
Student& advHouston (Student& s, void*)
```

```
{ if s.getAdvisor().getAddress().contain("Houston")  
    return s;
```

```
};
```

```
condFuncType addrHoustonPtr=addrHouston;
```

Four student instances are contained in the result of the above query: Benjamin Bayer, Katherine Ashly, Dick Davison and Charles Cooper.

A project operation is needed in the following query.

Query 2: "Find Names and their departments of all faculty members"

```
oSetF.project (nameDeptPtr, void*)
```

where

```
ObjectSet& nameDept(Faculty& f, void*)  
{ ObjectSet& result=*(new ObjectSet());  
  result.add(f.getPName());  
  result.add(f.getBelongs());  
  return result;  
funcType nameDeptPtr=nameDept;
```

The answer of this query is : { {"John Smith", "Mathematics"}, {"Frank Wong", "Mathematics"}, {"Jennifer Wallance", "Computer Science"}, {"Alicia Zelaya", "Computer Science"}, {"Ramesh Jabbar", "Physics"} }. As we mentioned before, since the dynamic type creation is not implemented in our simulation. New objects produced by query processing belong to the root class, **Object**, i.e., the result of the query contains five instances of **Object**.

A join operation is needed in the following query.

Query 3: "Find the students who lives at the same street as some faculty members live"

```
oSetS.join (oSetF, sameStreetPtr)
```

where,

```
ObjectSet& sameStreet (Student& s, Faculty& f, void*)  
{ ObjectSet& result=*(new Object());  
  if f.getAddress().street()==s.getAddress().street()  
  {result.add(s);  
   result.add(f);  
   return result;}  
}
```


The response for the above query is: { {"Frank Wong", "Katherine Ashly"}, {"Ramesh Jabbar", "Joyce English"} }. Similar to the project operation, the elements objects in the result belong to class Object.

6.5 REMARKS

We point out some features in our simulation. Most of these features are related to our implementation language, C++. Unlike Smalltalk, Turbo C++ does not provide the metadata classes, so users are unable to access the meta information of the classes. Therefore, we build the metadata classes to store the meta information of the classes, i.e., schema of the database. Since these metadata classes are user defined classes, when the schema is changed, such as, adding a new class or modifying an existing class definition, the metadata is not updated dynamically. It is updated manually by an extra set of statements.

In C ++, the extension of a class, its all instances, is not automatically maintained. So we create the object-set module to group objects into object-sets. When a new object is created, it may be inserted into one or more than one object-sets, or it may not be inserted into any object-sets. it is users' responsibility to maintain the object-sets. Heterogeneous class is implemented in our simulation as class ObjectSet. HomOSet classes are implemented too while RHOSet classes are not implemented.

Due to the time constrain, our simulation only implements the core part of the data model described in Chapter 4. The prototype is memory-resident and persistence is not implemented. Also we do not have a separate query language whereas the queries are expressed in C++. However, the simulation does demonstrate the fundamental features of our data model, such as, object-sets, special treatment of relationships, metadata and set-oriented operations

Chapter 7. Conclusions

As object-oriented database system moving from research laboratory to commercial market, it is important to have a precise definition for it. In this thesis we provide characteristics of object-oriented data base systems at two levels. Our contribution has two aspects: (i) ameliorating the conceptual confusions in the field and clarifying some relevant issues; (ii) providing a platform for the development and implementation of future database systems.

The lack of formal definition of object-oriented database systems has attracted more and more attention in the database community and a number of research efforts have begun to address the problem. In Chapter 3 we proposed two definitions for an object-oriented database system at high level characteristics. The first one is the basic definition which may be considered as a threshold to determine whether a given system is an object-oriented database system. Various existing systems may be justified according to this definition. Moreover, we analyzed several kinds of database systems or data models that are closely related to our subject. We pointed out the similarity and differences between those systems or models and object-oriented database system. The second one is the extended definition. In addition to the requirements in the basic definition, it also includes features that are unique to object-oriented database system and features that required by new applications. The extended definition is a part of our platform for future database systems.

Chapter 4 is dedicated to the establishing an object-oriented data model. Since the exact semantic of the term “data model” in the context of object-oriented database system has not been defined yet, our discussion on the data model was made in the classical sense, that is, our data model consists of two parts, structural part and operational part. On the structural aspect we propose our definitions for some essential but fussy concepts, such as, objects and values, states and attributes, and object identity. For some more

sophisticated concepts such as relationship and set of objects, we furnish them with clear semantics and powerful capability to cover the drawbacks of the approaches proposed in literature. On the operational aspects we deal with three general types of database operations, retrieval operations, updating operations and meta operations. Our goal is to provide such operations for object-oriented data model that not only take full advantage of the object-oriented paradigm, such as, encapsulation and rich modeling capability, but also retain the nice features of traditional data model, such as set-oriented and closed property in relational model.

Another part of our platform for future database systems is a unified view of various database systems. We presented two database system evolution paths: from relational to object-oriented and from network to object-oriented. Also we reinterpreted different concepts in various systems so that they become compatible at certain abstract level. As the result of our discussion, we have two conclusions: (i) object-oriented database system is the evolution of traditional database systems; (ii) traditional data models may be considered as special cases of the object-oriented data model we present.

Finally, in Chapter 6 we present a simulation of the core part of the data model described in Chapter 4. The main purpose of our simulation is to demonstrate the basic features of the data model. Our simulation includes three parts, the metadata mechanism, object management facility and stored objects. We discuss the rationale behind our implementation and the alternatives we considered.

The object-oriented database technology is still in a premature stage. From its definition and data model to implementation techniques, there are numerous open questions. It is still a challenge to future research work to address these issues. Therefore, it is not our intention to propose a final definition. Also we do not claim that our model is a standard object-oriented data model. Our goal is two folded. First, by giving definitions for an object-oriented database system and an object-oriented data model we attempt to ameliorate the conceptual confusions in the field and clarifying some relevant issues. Second, by present two evolution paths of database systems and a unified view of database systems, we provide a

platform for the development and implementation of future general-purpose database systems.

Reference

- [Agha 86] G. Agha, *ACTORS: a Model of concurrent computation in Distributed systems*. The MIT Press, Cambridge, MA 1986.
- [Alas 90] A. Alashqur, et. al., "A Rule-Based Language fro Deductive Object-Oriented Databases", *Proc. of 6th International Conference on Data Engineering*, Los Angeles, CA, Feb. 1990.
- [Andr 87] J. Andrews and C. Harris, "Combining Language and Database Advances in a Object-Oriented Development Environment," *Proc. of OOPSLA*, Oct. 1987, pp. 430-440.
- [Atki 90] M. Atkinson et. al., "The Object-Oriented Database System Manifesto", in *Deductive and Object-Oriented Database*, Elsevere Science Publishers, Amsterdam, Netherlands, 1990.
- [Atki 87] M. Atkinson and O.P. Buneman, "Types and Persistence in Database Programming Languages", *ACM Computing Surveys* 19, 2, June 1987, pp. 105-190.
- [Atki 83] M. Atkinson et. al., "An approach to persistent programming", *Computer Journal*, Vol.26, no.4, 1983, pp 360-365.
- [Banc 88] F. Bancilhon et. al., "The Design And Implementation of O2, An Object-Oriented Database System", *proc. of 2nd Intl. Workshop on Object-Oriented Database System*, 1988.
- [Banc 87] F. Bancilhon et. al., "FAD, a Powerful and single Database Language", *Proc. of 13th VLDB Conference*, Brighton, 1987, pp. 97-105.
- [Banc 89] F. Bancilhon et. al., "A Query Language for the O2 Object-Oriented Database System", *Proc. 2nd Intl. Workshop on Database Programming Languages*, Glenden Beach, Oregon, June, 1989.
- [Beec 88] D. Beech, "A Foundation for Evolution from relational to Object Database", *Proc. of Intl. conf. on Extending Database Technology*, March 1988, pp. 251-267.
- [Beer 89] C. Beeri, "Formal Models for Object-Oriented Databases", in *Deductive and Object-Oriented Database*, Elsevere Science Publishers, Amsterdam, Netherlands, 1990.

- [Bene 87a] J. Benerjee et. al., "Data Model Issues for Object-Oriented Applications", *ACM Trans. on Office Information Systems*, Vol.5,N0.1, Jan. 1987, pp. 3-26.
- [Bene 87 b] J. Benerjee et. al., "Semantics and Implementations of Schema Evolution in Object-Oriented Databases", *Proc. of ACM SIGMOD*, 1987.
- [Bene 88] J. Benerjee et. al., "Queries in Object-Oriented Database Systems", *Proc. of Forth Int. Conf. on Data Eng.*, Feb. 1988, pp. 31-38.
- [Birt 73] G. Birtwistle et. al., *Simula Begin*, Auerbach Press, Philadelphia, 1973.
- [Bobr 86] D. Bobrow et. al., "CommonLoops: Merging Common Lisp and Object-Oriented Programming", in *Proc. 1st intl. conf. OOPSLA*, Protland, OR, Oct. 1986.
- [Bret 89] R. Brettl, et. al., "The Gemstone Data Management System", in *Object-Oriented Concepts, Database and Applications*, ed. W. Kim and F.H. Lochovsky, Addison-Wesley, Reading, MA, 1988.
- [Care 88] M. Carey, et. al., "A Data Model and Query Language For EXODUS", *Proc. of ACM-SIGMOD Conference 1988*, pp. 413-423.
- [Chen 76] P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Trans. database System*, Vol.1, No.1, Jan. 1976, PP. 9-36.
- [Chen 90] A. Chen et. al., "Storage and Access Structures to Support a Semantic Data Model", in *Readings in Object-Oriented Database Systems*, ed. S. Zdonik and d.Maier, Morgan Kaufmann Publishers, San Mateo, California, 1990.
- [Chou 86] H. Chou and W. Kim, "A Unified Framework for Version Control in a CAD Environment", *Proc. 12th VLDB*, 1986.
- [Cox 84] B. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology", *IEEE Software*, Jan. 1984.
- [Data 86] P. Datam, et. al., "A DBMS prototype to support Extended NFNF Relations: An integrated View of Flat Tables and Hierarchies". *Proc. ACM-SIGMOD Conference on Management of Data*, Washington DC, 1986.
- [Died 87] J. Diederich and J. Milton, "ODDESSY: An Object-Oriented Database Design

System", *Proc. of Int. Conf. on Data Engineering*, May 1987, pp 235-244.

[Ditt 88] K. Dittrich, "Preface", in *Proc. 2nd International Workshop on Object-Oriented Database Systems*, Bad Munster am Stein-Ebernburg, FRG, Sept, 1988.

[Ditt 88a] K. Dittrich and R. Lorie " Version Support for Engineering Database Systems", *TOSE*, Vol. 14, No. 4, 1988

[Elma 89] R. Elmasri and S. Navathe, *Fundamentals of Database systems*, The Benjamin /Cumming Publishing Company, Redwood City, CA 94065, 1989.

[Fish 87] D. Fishman et. al., "IRIS: An Object-Oriented Database Management System", *ACM Trans. on Office Information Systems*, Vol. 5, No.1, Jan. 1987, pp. 49-60.

[Ford 88] S. Ford et, al. "ZEITGEIST: Database Support for Object-Oriented Programming", *Proc. of 2nd Int. Workshop on Object-Oriented Database System*, Sept. 1988, pp.23-42

[Gold 83] A. Goldberg and D. Robsom, *Smalltalk-80: The language und its implementation*. Addison-Wesley, Reading, MA 1983.

[Guti 89] R. Guting and R. Zicar, "An Introduction to the nested Sequences of Tuples Data Model and Algebra", *Nested Relations and Complex Object in Databases*, Lecture Notes in Computer Science, #361, Spring-Verlag, 1989.

[Hamm 81] M. Hammer, D. MecLeod, "Data Description with SDM: a Semantic Data Model ", *TODS*, Vol. 6, No.3, Sept., 1981.

[Hard 83] T. Harder and A. Teuter, "Principles of Transaction Oriented Database Recovery", *ACM Computing Serveys*, Vol. 15, No. 2, June 1983.

[Hong 90] S. Hong and F. Maryanski, "Using a Meta-Model to Represent Object-Oriented Data Model", *Proc. of Sixth Int. Conf. on Data Eng.* May 1990, pp. 424-433.

[Horn 87] M. Hornick and S. Zdonik, "A shared, Segmented Memory System for an Object-oriented database", *ACM Trans. on Office Information Systems*. vol.5. no.1. pp.70-85, 1987.

[Hsu 88] M. Hsu et. al. ," An Execution Model for Active Data Base Management Systems", *Proc. 3rd Intl. Conf. on Data and Knowledge Bases: Improving Usability and*

Responsiveness, 1988

[Hull 87] R. Hull and R. King, "Semantic Database Modeling: Survey, Application and Research Issues." *ACM Computing Surveys*, Vol. 19, No. 3, September 1987.

[Katz 86] H. Katz et. al., "Managing Changes in a Computer Aided Design Data Base", *Proc. 13th VLDB*, 1987.

[Kemp 87] A. Kemper, P. Lockemann and M. Wallrath, "An Object-Oriented Database System for Engineering Applications", *Proc. of ACM SIGMOD Annual Conf.*, Dec. 1987, pp. 299-301.

[Khos 90] S. Khoshafiar M. Franklin and M. Carey, "Storage Management for Persistent Complex Objects", *Information Systems* Vol.15, No3. pp. 303-320, 1990.

[Kim 89a] W. Kim et. al., "Features of ORION Object-Oriented Database System", in *Object-Oriented Concepts, Database and Applications*, ed. W. Kim and F.H. Lochovsky, Addison-Wesley, Reading, MA, 1988.

[Kim 89b] W. Kim et. al., "Indexing techniques for object-oriented databases", in *Object-Oriented Concepts, Database and Applications*, ed. W. Kim and F.H. Lochovsky, Addison-Wesley, Reading, MA, 1988.

[Kim 89c] W. Kim, "a Model of Queries GfFor Object-Oriented Databases", *Proc. of International Conference on VLDB*, 15, 1989, pp. 423-432.

[Kim 90] W. Kim, "Object-oriented Database: Definition and Research Directions", *IEEE Trans. on Knowledge and Data Engineering*, Vol.2, No. 3, Sept. 1990, pp.327-341.

[King 88] R. King, "My Cat is Objected-Oriented", in *Object-Oriented Concepts, Database and Applications*, ed. W. Kim and F.H. Lochovsky, pp 23-30, Addison-Wesley, Reading, MA, 1988.

[Klah 85] P. Klahold et. al., "A Transaction Model Supporting Complex Applications is Integrated Information System", *proc. 3rd. SIGMOD*, 1985.

[Klah 86] P. Klahold et. al., "A General Model for Version Management in Database", *Proc. 12th VLDB*, 1986.

[Kort 88] H. Korth, "Optimization of Object-Oriented Queries", *Proc. 2nd AOODBS*,

1988.

[Lecl 88] C. Lecluse et. al., "O2, an Object-oriented Database Model", *Proc. SIGMOD*, 1988.

[Lieb 86] H. Lieberman," Using Prototype Object to Implement Share Behavior in Object-Oriented System", *Proc. of OOPSLA*, Sept. 1986, pp 214- 223.

[Lind 87] B. Lindsay et. al., "A Data Management extension Architecture", *Proc. ACM SIGMOD Intl. Conf. Management Data*, May 1987.

[Lisk 81] B. Liskov et. al., "CLU reference Manual", *Lecture Note in Computer Science*, Springer-Verlag, 1981.

[Lyng 86] P. Lyngback and W. Kent, " A Data Modeling Methodology for the Design and Implementation of Information Systems", *Proc. of Int. workshop on Object-Oriented Database Syst.*, Sept.1986, pp. 6-17.

[Maie 86] D. Maier and J. Stein, "Development of An Object-Oriented DBMS." *Proc. of OPSLA '86*, Sept.,1986, pp. 472-482.

[Maie 89] D. Maier, "Making Database Systems fase enough for CAD Applicaions", in *Object-Oriented Concepts, Database and Applications*, ed. W. Kim and F.H. Lochovsky, Addison-Wesley, Reading, MA,1988.

[Maie 89b] D. Maier, " Why Isn't There an Object-Oriented Data Model?", *Proc. IFIP 11th World Computer Conference*, Aug. 1989.

[Meye 85] B. Meyer, "Eiffel: A Language for Software Engineering", *Tech Report, Dept. Comp. Sci., Univ. of California at Santa Barbara*, Nov. 1985.

[Mica 88] J. Micallef," Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", *J. Object-Oriented Programming*, vol.1 no.1, Apr. 1988,pp.12-36.

[Mins75] M. Minsky, "A Framework for Representing Knowledge", in *Psychology of Computer Version*, MIT Press, Cambridge MA, 1975.

[Moon 86] D. Moon, "Object-oriented Programming with Flavors", *Proc. 1st OOPSLA*, 1986.

PAGINATION ERROR.

ERREUR DE PAGINATION.

TEXT COMPLETE.

LE TEXTE EST COMPLET.

NATIONAL LIBRARY OF CANADA.

BIBLIOTHEQUE NATIONALE DU CANADA.

CANADIAN THESES SERVICE.

SERVICE DES THESES CANADIENNES.

Appendix C++ Source Code of The Simulation

Metadata Classes Definition

```
#ifndef __METACLAS_H
#define __METACLAS_H
#endif

#ifndef __IOSTREAM_H
#include <iostream.h>
#endif

#ifndef __CLSTYPES_H
#include <clstypes.h>
#endif

#ifndef __STRNG_H
#include <strng.h>
#endif

#ifndef __OBJECTSET_H
#include <objectset.h>
#endif

//define ObjectSet classes

class MetaClassSet: public ObjectSet
{
public:
MetaClassSet( sizeType setSize = DEFAULT_SET_SIZE ) :
    ObjectSet( setSize ) {}

    virtual ~MetaClassSet();

    virtual void          insert( Object& );
    virtual classType     isA() const;
    virtual char          *nameOf() const;
};

class PrimClassSet: public ObjectSet
{
public:
PrimClassSet( sizeType setSize = DEFAULT_SET_SIZE ) :
    ObjectSet( setSize ) {}

    virtual ~PrimClassSet();
};
```

```

        virtual void          insert( Object& );
        virtual classType     isA() const;
        virtual char          *nameOf() const;
};

class UClassSet: public ObjectSet
{
public:
    UClassSet( sizeType setSize = DEFAULT_SET_SIZE ) :
        ObjectSet( setSize ) {}

    virtual ~UClassSet();

    virtual void          insert( Object& );
    virtual classType     isA() const;
    virtual char          *nameOf() const;
};

class MessageSet: public ObjectSet
{
public:
    MessageSet( sizeType setSize = DEFAULT_SET_SIZE ) :
        ObjectSet( setSize ) {}

    virtual ~MessageSet();

    virtual void          insert( Object& );
    virtual classType     isA() const;
    virtual char          *nameOf() const;
};

class AttributeSet: public ObjectSet
{
public:
    AttributeSet( sizeType setSize = DEFAULT_SET_SIZE ) :
        ObjectSet( setSize ) {}

    virtual ~AttributeSet();

    virtual void          insert( Object& );
    virtual classType     isA() const;
    virtual char          *nameOf() const;
};

class MetaClass : public Object
{
public:
    MetaClass( const MetaClass& c ) :
        className(c.getClassName()), methods(c.getMethods()),
        allInstances( c.getAllInstances() ) {}
    MetaClass(String& n, MessageSet& m, ObjectSet& i ):

```

```

    className( n ), methods( m ), allInstances( i ){}

    virtual ~MetaClass();

    String&      getClassName() const { return className; };
    MessageSet& getMethods() const { return methods; };
    void         addMethod(Object& );
    ObjectSet&   getAllInstances() const
                { return allInstances; };

    virtual classType      isA() const;
    virtual char           *nameOf() const;
    virtual hashValueType  hashValue() const;
    virtual int            isEqual( const Object& ) const;
    virtual int            isAssociation() const;
    virtual void           printOn( ostream& ) const;

protected:
    String&      className;
    MessageSet&  methods;
    ObjectSet&   allInstances;
};

class PrimClass : public MetaClass
{
public:

    PrimClass( const PrimClass& c ) :MetaClass(c){}
    PrimClass(String& n, MessageSet& m, ObjectSet& i):
        MetaClass(n,m,i){}

    virtual ~PrimClass();

    virtual classType      isA() const;
    virtual char           *nameOf() const;
    virtual hashValueType  hashValue() const;
    virtual int            isEqual( const Object& ) const;
    virtual int            isAssociation() const;
    virtual void           printOn( ostream& ) const;
};

class UserClass : public MetaClass
{
public:
    UserClass( const UserClass& c ) :
    MetaClass(c.getClassName(),c.getMethods(),
        c.getAllInstances()), superclass(c.getSuperclass()),
        subclasses(c.getSubclasses()),
        attributes(c.getAttributes()) {}
    UserClass(String& n, AttributeSet& a, MessageSet& m,
        ObjectSet& i, UserClass& sup, UserClassSet& sub):
        MetaClass(n, m ,i), superclass( sup ), subclasses(sub),

```

```

        attributes( a ){}

virtual ~UserClass();

AttributeSet&    getAttributes() const { return attributes; };
UserClassSet&    getSubclasses() const { return subclasses; };
UserClass&       getSuperclass() const { return superclass; };
void            addSubclass(UserClass&);
void            addAttribute(Object&);

virtual classType    isA() const;
virtual char         *nameOf() const;
virtual hashValueType hashValue() const;
virtual int          isEqual( const Object& ) const;
virtual int          isAssociation() const;
virtual void         printOn( ostream& ) const;

private:
    AttributeSet&    attributes;
    UserClassSet&    subclasses;
    UserClass&       superclass;
};

class Attribute : public Object
{
public:
    Attribute(String& n, MetaClass& d, UserClass& o) :
        attrName( n ), domain( d ), owner( o ){}
    Attribute( const Attribute& a ) :
        attrName( a.getAttrName() ), domain( a.getDomain() ),
        owner( a.getOwner() ){}

    virtual ~Attribute();

    String&    getAttrName() const { return attrName; }
    MetaClass& getDomain() const { return domain; }
    UserClass& getOwner() const { return owner; }

    virtual classType    isA() const;
    virtual char         *nameOf() const;
    virtual hashValueType hashValue() const;
    virtual int          isEqual( const Object& ) const;
    virtual int          isAssociation() const;
    virtual void         printOn( ostream& ) const;

private:
    String&    attrName;
    MetaClass& domain;
    UserClass& owner;
};

class Message : public Object

```

```

{
public:
    Message(String& s, MetaClass& c) :
        selector( s ), receiver( c ) {}
    Message( const Message& m ) :
        selector( m.getSelector() ), receiver( m.getReceiver() ) {}

    virtual ~Message();

    String&      getSelector() const { return selector; }
    MetaClass&   getReceiver() const { return receiver; }

    virtual classType      isA() const;
    virtual char           *nameOf() const;
    virtual hashValueType  hashCode() const;
    virtual int            isEqual( const Object& ) const;
    virtual int            isAssociation() const;
    virtual void           printOn( ostream& ) const;

private:
    String&      selector;
    MetaClass&   receiver;
};

```

Methods Implementation for Metadata Classes

```
#ifndef __IOSTREAM_H
#include <iostream.h>
#define __IOSTREAM_H
#endif
```

```
#ifndef __CLSTYPES_H
#include <clstypes.h>
#endif
```

```
#ifndef __METACLAS_H
#include <metacclas.h>
#endif
```

```
AttributeSet::~AttributeSet()
{
}
```

```
classType AttributeSet::isA() const
{
    return setClass;
}
```

```
char *AttributeSet::nameOf() const
{
    return "AttributeSet";
}
```

```
void AttributeSet::insert( Cbject& objectToAdd )
{
    if (objectToAdd.isA()==attributeClass )
        ObjectSet::add( objectToAdd );
    else cerr<<"Only instances of class Attribute can be inserted
        into AttributeSet";
}
```

```
MetaClassSet::~MetaClassSet()
{
}
```

```
classType MetaClassSet::isA() const
{
    return setClass;
}
```

```
char *MetaClassSet::nameOf() const
```



```

{
    return "MetaClassSet";
}

void MetaClassSet::insert( Object& objectToAdd )
{
    if (objectToAdd.isA()==metaclassClass )
        ObjectSet::add( objectToAdd );
    else cerr<<"Only instances of class MetaClass can be inserted
        into MetaClassSet";
}

PrimClassSet::~PrimClassSet()
{
}

classType PrimClassSet::isA() const
{
    return setClass;
}

char *PrimClassSet::nameOf() const
{
    return "PrimClassSet";
}

void PrimClassSet::insert( Object& objectToAdd )
{
    if (objectToAdd.isA()==primclassClass )
        ObjectSet::add( objectToAdd );
    else cerr<<"Only instances of class PrimClass can be inserted
        into PrimClassSet";
}

UserClassSet::~UserClassSet()
{
}

classType UserClassSet::isA() const
{
    return setClass;
}

char *UserClassSet::nameOf() const
{
    return "UserClassSet";
}

void UserClassSet::insert( Object& objectToAdd )
{
    if (objectToAdd.isA()==userclassClass )
        ObjectSet::add( objectToAdd );
}

```

```

        else cerr<<"Only instances of class UClass can be inserted
                    into UClassSet";
    }

    MessageSet::~MessageSet()
    {
    }

    classType MessageSet::isA() const
    {
        return setClass;
    }

    char *MessageSet::nameOf() const
    {
        return "MessageSet";
    }

    void MessageSet::insert( Object& objectToAdd )
    {
        if (objectToAdd.isA()==messageClass )
            ObjectSet::add( objectToAdd );
        else cerr<<"Only instances of class Message can be inserted
                    into MessageSet";
    }

    MetaClass::~MetaClass()
    {
    }

    classType MetaClass::isA() const
    {
        return metaclassClass;
    }

    char *MetaClass::nameOf() const
    {
        return "MetaClass";
    }

    void MetaClass::printOn( ostream& outputStream ) const
    {
        outputStream << " " << nameOf() << " { ";
        outputStream<<className;
        outputStream << " }\n";
    }

    hashValueType MetaClass::hashValue() const
    {
        return className.hashValue();
    }

```

```

}

int MetaClass::isEqual( const Object& toObject ) const
{
    return
        className.isEqual(((MetaClass&)toObject).getClassName());
}

int MetaClass::isAssociation() const
{
    return 0;
}

void MetaClass::addMethod(Object& m){methods.add(m);}

Message::~Message()
{
}

classType Message::isA() const
{
    return messageClass;
}

char *Message::nameOf() const
{
    return "Message";
}

void Message::printOn( ostream& outputStream ) const
{
    outputStream << " " << nameOf() << " { ";
    outputStream<<selector;
    outputStream << ", ";
    outputStream <<receiver;
    outputStream << " }\n";
}

hashValueType Message::hashValue() const
{
    return selector.hashValue();
}

int Message::isEqual( const Object& toObject ) const
{
    return
        ((selector.isEqual(((Message&)toObject).getSelector()))
        &&(receiver.isEqual(((Message&)toObject).getReceiver())));
}

```

```

}

int Message::isAssociation() const
{
    return 0;
}

PrimClass::~~PrimClass()
{
}

classType PrimClass::isA() const
{
    return metaclassClass;
}

char *PrimClass::nameOf() const
{
    return "PrimClass";
}

void PrimClass::printOn( ostream& outputStream ) const
{
    MetaClass::printOn(outputStream);
}

hashValueType PrimClass::hashValue() const
{
    return MetaClass::hashValue();
}

int PrimClass::isEqual( const Object& toObject ) const
{
    return MetaClass::isEqual(toObject);
}

int PrimClass::isAssociation() const
{
    return 0;
}

UserClass::~~UserClass()
{
}

classType UserClass::isA() const
{
    return metaclassClass;
}

```

```

}

char *UserClass::nameOf() const
{
    return "UserClass";
}

void UserClass::printOn( ostream& outputStream ) const
{
    MetaClass::printOn(outputStream);
}

hashValueType UserClass::hashValue() const
{
    return MetaClass::hashValue();
}

int UserClass::isEqual( const Object& toObject ) const
{
    return MetaClass::isEqual( toObject);
}

int UserClass::isAssociation() const
{
    return 0;
}

void UserClass::addAttribute(Object& a){attributes.add(a);}
void UserClass::addSubclass(UserClass& s){subclasses.add(s);}

Attribute::~Attribute()
{
}

classType Attribute::isA() const
{
    return attributeClass;
}

char *Attribute::nameOf() const
{
    return "Attribute";
}

void Attribute::printOn( ostream& outputStream ) const
{
    outputStream << " " << nameOf() << " ";
    outputStream<<attrName;
}

```

```

        outputStream << " , ";
        outputStream<<domain;
        outputStream << " }\n";
    }

hashValueType Attribute::hashValue() const
{
    return attrName.hashValue();
}

int Attribute::isEqual( const Object& toObject ) const
{
    return
        (attrName.isEqual(((Attribute&)toObject).getAttrName()));
}

int Attribute::isAssociation() const
{
    return 0;
}

```

ObjectSet Class Definition

```
#ifndef __OBJECTSET_H
#define __OBJECTSET_H
#endif
```

```
#ifndef __SET_H
#include <set.h>
#endif
```

```
#ifndef __IOSTREAM_H
#include <iostream.h>
#endif
```

```
class ObjectSet: public Set
{
public:
ObjectSet( sizeType setSize = DEFAULT_SET_SIZE ) :
    Set( setSize ) {}

    virtual ~ObjectSet();

    virtual void          insert( Object& );
    virtual void          deleting( condFuncType, void* );
    virtual classType     isA() const;
    virtual char          *nameOf() const;
    ObjectSet&            select( condFuncType, void* );
    ObjectSet&            project( projectFuncType, void* );
    ObjectSet&            join( ObjectSet&, condFuncType, void* );
};
```

Methods Implementation For ObjectSet

```
#ifndef __CLSTYPES_H
#include <clstypes.h>
#endif

#ifndef __OBJECTSET_H
#include <objectset.h>
#endif

ObjectSet::~ObjectSet()
{
}

classType ObjectSet::isA() const
{
    return setClass;
}

char *ObjectSet::nameOf() const
{
    return "ObjectSet";
}

void ObjectSet::insert( Object& objectToAdd )
{
    Set::add( objectToAdd );
}

ObjectSet& ObjectSet::select( condFuncType testFuncPtr, void
                             *paramListPtr )

{
    ObjectSet& result=*(new ObjectSet(5));
    ContainerIterator &containerIterator =
        HashTable::initIterator();

    while( int(containerIterator) != 0 )
    {
        Object& testResult =
            containerIterator++.firstThat( testFuncPtr, paramListPtr );

        if ( testResult != NOOBJECT )
            result.add(testResult);
    }
    delete &containerIterator;
    return result;
}

void ObjectSet::deleting( condFuncType testFuncPtr, void
```



```

                                *paramListPtr )

{   ContainerIterator &containerIterator =
    HashTable::initIterator();

    while( int(containerIterator) != 0 )
    {
        Object& testResult =
            containerIterator++.firstThat( testFuncPtr, paramListPtr );

            if ( testResult != NOOBJECT )
                detach(testResult);
    }
    delete &containerIterator;
}

ObjectSet& ObjectSet::project(FuncType projectFuncPtr,
                               void *parameter)

{
    Objectset& result;
    ContainerIterator &containerIteraror = initIterator();
    while ( int(containerIterator) != 0 )
    {
        if (containerIterator != NOOBJECT )
        {
            ObjectSet& os =*(new ObjectSet ());
            os=((Object&)containerIterator++.(projectFuncPtr,
                                                *parameter);

            result.add(os);
        };
    }
    delete &containerIterator;
    return result;
};

```

Constants Definition

```
#ifndef __MAIN_H
#define __MAIN_H
#endif
```

```
#ifndef __UNIVERSITY_H
#include <univrsity.h>
#endif
```

```
#ifndef __METACLAS_H
#include <metacclas.h>
#endif
```

```
#define noOfStudents 10
#define noOfCourses 6
#define noOfFaculties 5
#define noOfDepts 3
#define maxNoOfMessages 10
#define noOfUserClasses 15
#define noOfPrimClasses 5
#define maxNoOfAttributes 10
#define maxNoOfSubclasses 5
#define noOfAttributes 30
#define noOfMessages 30
```

```
void createUniversity(DeptSet&, FacultySet&, StudentSet&,
                     CourseSet&, Crs_StdSet&);
void creatMetadata(PrimClassSet&, UserClassSet&, AttributeSet&,
                  MessageSet&);
```

Appalaication Classes Definition

```
#ifndef __UNIVSITY_H
#define __UNIVSITY_H
#endif
```

```
#ifndef __STRNG_H
#include <strng.h>
#endif
```

```
#ifndef __OBJECTSET_H
#include <objectset.h>
#endif
```

```
enum Sex {male, female};
enum Term {fall, winter, summer};
enum Rank {assistantProf, associateProf, prof};
```

```
//define ObjectSet Classes
```

```
class StudentSet: public ObjectSet
{
public:
StudentSet( sizeType setSize = DEFAULT_SET_SIZE ) :
    ObjectSet( setSize ) {}

    virtual ~StudentSet();

    virtual void      insert( Object& );
    virtual classType isA() const;
    virtual char      *nameOf() const;
    ObjectSet&        project (ObjectSet&);
};
```

```
class DeptSet: public ObjectSet
{
public:
DeptSet( sizeType setSize = DEFAULT_SET_SIZE ) :
    ObjectSet( setSize ) {}

    virtual ~DeptSet();

    virtual void      insert( Object& );
    virtual classType isA() const;
    virtual char      *nameOf() const;
};
```

```

class FacultySet: public ObjectSet
{
public:
FacultySet( sizeType setSize = DEFAULT_SET_SIZE ) :
    ObjectSet( setSize ) {}

    virtual ~FacultySet();

    virtual void        insert( Object& );
    virtual classType    isA() const;
    virtual char         *nameOf() const;
};

class CourseSet: public ObjectSet
{
public:
CourseSet( sizeType setSize = DEFAULT_SET_SIZE ) :
    ObjectSet( setSize ) {}

    virtual ~CourseSet();

    virtual void        insert( Object& );
    virtual classType    isA() const;
    virtual char         *nameOf() const;
};

class Crs_StdSet: public ObjectSet
{
public:
Crs_StdSet( sizeType setSize = DEFAULT_SET_SIZE ) :
    ObjectSet( setSize ) {}

    virtual ~Crs_StdSet();

    virtual void        insert( Object& );
    virtual classType    isA() const;
    virtual char         *nameOf() const;
};

class Person: public Object
{
public:
Person(String& aName, long aSsn, int aSex, int aBDate,
    String& aAddress):
    personName(aName), ssn(aSsn), sex(aSex), birthDate(aBDate),
    address(aAddress){}

    virtual ~Person();

```

```

virtual classType      isA() const;
virtual char           *nameOf() const;
virtual hashValueType  hashCode() const;
virtual int            isEqual( const Object& ) const;
virtual int            isAssociation() const;
virtual void           printOn( ostream& ) const;

```

```

String&    getPName()const {return personName;};
long       getSsn() const {return ssN;};
int        getSex() const {return sex;};
int        getBirthDate() const {return birthDate;};
String&    getAddress() const {return address;};

```

protected:

```

String&    personName;
long       ssN;
int        sex;
int        birthDate;
String&    address;

```

};

//Forward reference

class Faculty;

class Dept;

class Student: public Person

{

public:

```

Student(String& aName, long aSsn, int aSex, int aBDate,
String& aAddress, int aYear, Dept& aMajor,
Crs_StdSet& rgsCourses, Faculty& aAdvisor):
Person(aName, aSsn, aSex, aBDate,aAddress), year(aYear),
major(aMajor), registeredCrs(rgsCourses), advisor(aAdvisor){}

```

virtual ~Student();

```

virtual classType      isA() const;
virtual char           *nameOf() const;
virtual hashValueType  hashCode() const;
virtual int            isEqual( const Object& ) const;
virtual int            isAssociation() const;
virtual void           printOn( ostream& ) const;

```

```

int        getYear() const {return year;};
Dept&       getMajor() const {return major;};
Crs_StdSet& getRegistered() const {return registeredCrs;};
Faculty&    getAdvisor() const {return advisor;};
ObjectSet&  sProject() {
    ObjectSet& result=*(new ObjectSet(5));
    result.add((Object&)major);
}

```

```

        result.add((Object&)advisor);
        return result;
    }

protected:
    int          year;
    Dept&         major;
    Crs_Set&      registeredCrs;
    Faculty&      advisor;
};

class Faculty: public Person
{
public:
    Faculty(String& aName, long aSsn, int aSex, int aBDate,
        String& aAddress, int aPhone, int aRank, Dept& aDept,
        StudentSet& aStSet, CourseSet& aCrsSet):
        Person(aName, aSsn, aSex, aBDate, aAddress), phone(aPhone),
        rank(aRank), belongs(aDept), supervise(aStSet), teach(aCrsSet){}

    virtual ~Faculty();
    virtual classType      isA() const;
    virtual char           *nameOf() const;
    virtual hashValueType  hashCode() const;
    virtual int            isEqual( const Object& ) const;
    virtual int            isAssociation() const;
    virtual void           printOn( ostream& ) const;

    int                    getPhone() const {return phone;};
    int                    getRank() const {return rank;};
    Dept&                  getBelongs() const {return belongs;};
    StudentSet&            getSupervise() const {return supervise;};
    CourseSet&             getTeach() const {return teach;};

private:
    int                    phone;
    int                    rank;
    Dept&                  belongs;
    StudentSet&            supervise;
    CourseSet&             teach;
};

class Dept:public Object
{
public:
    Dept(String& aDeptName, long aDeptPhone, String& theMainOffice,
        FacultySet& aFacultyMembers, StudentSet& theRgstStudents,
        CourseSet& theCourses):
        deptName(aDeptName), deptPhone(aDeptPhone),
        mainOffice(theMainOffice), facultyMembers(aFacultyMembers),
        rgstStudents(theRgstStudents), courses(theCourses){}

```

```

virtual ~Dept();

virtual classType      isA() const;
virtual char           *nameOf() const;
virtual hashValueType  hashCode() const;
virtual int            isEqual( const Object& ) const;
virtual int            isAssociation() const;
virtual void           printOn( ostream& ) const;

String&                getDeptName() const {return deptName;};
long                  getDeptPhone() const {return deptPhone;};
String&                getMainOffice() const {return mainOffice;};
FacultySet&            getFacultyMembers() const
                        {return facultyMembers;};
StudentSet&            getRgstStudents() const
                        {return rgstStudents;};
CourseSet&             getCourses() const { return courses;};

private:

String&                deptName;
long                  deptPhone;
String&                mainOffice;
FacultySet&            facultyMembers;
StudentSet&            rgstStudents;
CourseSet&             courses;
};

class Course:public Object
{
public:
Course(long aCourseNo, String& aCourseName, Dept& aDept,
      Crs_StdSet& theAttendrs, Faculty& aTeacher):
      courseNo(aCourseNo), courseName(aCourseName),
      attendrs(theAttendrs), offeredBy(aDept),
      teacher(aTeacher){}

virtual ~Course();

virtual classType      isA() const;
virtual char           *nameOf() const;
virtual hashValueType  hashCode() const;
virtual int            isEqual( const Object& ) const;
virtual int            isAssociation() const;
virtual void           printOn( ostream& ) const;

long                  getCourseNo() const {return courseNo;};
String&                getCourseName() const {return courseName;};
Crs_StdSet&            getCourseAtt() const {return attendrs;};
Dept&                  getOfferedBy() const {return offeredBy;};
Faculty&                getTeacher() const {return teacher;};

```

private:

long	courseNo;
String&	courseName;
Dept&	offeredBy;
Crs_StdSet&	attenders;
Faculty&	teacher;

};

class Crs_Std:public Object

{

private:

Crs_Std(Course& aCourse, Student& aStudent, char aGrade):
course(aCourse), student(aStudent), grade(aGrade) {}

virtual ~Crs_Std();

virtual classType	isA() const;
virtual char	*nameOf() const;
virtual hashValueType	hashValue() const;
virtual int	isEqual(const Object&) const;
virtual int	isAssociation() const;
virtual void	printOn(ostream&) const;

Course& getCourse()	const {return course;};
Student& getStudent()	const {return student;};
char getGrade()	const {return grade;};

Course&	course;
Student&	student;
char	grade;

friend class Student, Course;

};

Methods Implementation for Application Classes

```
#ifndef __IOSTREAM_H
#include <iostream.h>
#define __IOSTREAM_H
#endif
```

```
#ifndef __CLSTYPES_H
#include <clstypes.h>
#endif
```

```
#ifndef __UNIVERSITY_H
#include <university.h>
#endif
```

```
StudentSet::~StudentSet()
{
}
```

```
classType StudentSet::isA() const
{
    return setClass;
}
```

```
char *StudentSet::nameOf() const
{
    return "StudentSet";
}
```

```
void StudentSet::insert( Object& objectToAdd )
{
    if (objectToAdd.isA()==studentClass )
        ObjectSet::add( objectToAdd );
    else cerr<<"Only instances of class Student can be inserted
               into StudentSet";
}
```

```
CourseSet::~CourseSet()
{
}
```

```
classType CourseSet::isA() const
{
    return setClass;
}
```

```
char *CourseSet::nameOf() const
{
    return "CourseSet";
}
```

```

}

void CourseSet::insert( Object& objectToAdd )
{
    if (objectToAdd.isA()==courseClass )
        ObjectSet::add( objectToAdd );
    else cerr<<"Only instances of class Course can be inserted
        into CourseSet";
}

FacultySet::~FacultySet()
{
}

classType FacultySet::isA() const
{
    return setClass;
}

char *FacultySet::nameOf() const
{
    return "FacultySet";
}

void FacultySet::insert( Object& objectToAdd )
{
    if (objectToAdd.isA()==facultyClass )
        ObjectSet::add( objectToAdd );
    else cerr<<"Only instances of class Faculty can be inserted
        into FacultySet";
}

DeptSet::~DeptSet()
{
}

classType DeptSet::isA() const
{
    return setClass;
}

char *DeptSet::nameOf() const
{
    return "DeptSet";
}

void DeptSet::insert( Object& objectToAdd )
{
    if (objectToAdd.isA()==studentClass )
        ObjectSet::add( objectToAdd );
    else cerr<<"Only instances of class Dept can be inserted into

```

```

        DeptSet";
    }

    Crs_StdSet::~Crs_StdSet()
    {
    }

    classType Crs_StdSet::isA() const
    {
        return setClass;
    }

    char *Crs_StdSet::nameOf() const
    {
        return "Crs_StdSet";
    }

    void Crs_StdSet::insert( Object& objectToAdd )
    {
        if (objectToAdd.isA()==crs_stdClass )
            ObjectSet::add( objectToAdd );
        else cerr<<"Only instances of class Crs_Std can be inserted
                    into Crs_StdSet";
    }

    Person::~Person()
    {
    }

    classType Person::isA() const
    {
        return userclassClass;
    }

    char *Person::nameOf() const
    {
        return "Person";
    }

    void Person::printOn( ostream& outputStream ) const
    {
        outputStream << " " << nameOf() << " { ";
        outputStream<<personName;
        outputStream << ", ssn: ";
        outputStream<<ssn;
        outputStream << " }\n";
    }

```

```

hashValueType Person::hashValue() const
{
    return personName.hashValue();
}

int Person::isEqual( const Object& toObject ) const
{
    return ssn == ( (Person&)toObject ).getSsn();
}

int Person::isAssociation() const
{
    return 0;
}

Student::~~Student()
{
}

classType Student::isA() const
{
    return studentClass;
}

char *Student::nameOf() const
{
    return "Student";
}

void Student::printOn( ostream& outputStream ) const
{
    Person::printOn( outputStream ) ;
}

hashValueType Student::hashValue() const
{
    return Person::hashValue();
}

int Student::isEqual( const Object& toObject ) const
{
    return Person::isEqual(toObject);
}

int Student::isAssociation() const
{
    return 0;
}

```

```

Faculty::~~Faculty()
{
}

classType Faculty::isA() const
{
    return facultyClass;
}

char *Faculty::nameOf() const
{
    return "Faculty";
}

void Faculty::printOn( ostream& outputStream ) const
{
    Person::printOn( outputStream ) ;
}

hashValueType Faculty::hashValue() const
{
    return Person::hashValue();
}

int Faculty::isEqual( const Object& toObject ) const
{
    return Person::isEqual(toObject);
}

int Faculty::isAssociation() const
{
    return 0;
}

Dept::~~Dept()
{
}

classType Dept::isA() const
{
    return deptClass;
}

char *Dept::nameOf() const
{
    return "Dept";
}

```

```

}

void Dept::printOn( ostream& outputStream ) const
{
    outputStream << " " << nameOf() << " { ";
    outputStream <<deptName;
    outputStream << ",Main Office: ";
    outputStream <<mainOffice;
    outputStream << " }\n";
}

hashValueType Dept::hashValue() const
{
    return deptName.hashValue();
}

int Dept::isEqual( const Object& toObject ) const
{
    return deptName.isEqual ( ( (Dept&)toObject ).getDeptName() );
}

int Dept::isAssociation() const
{
    return 0;
}

Course::~~Course()
{
}

classType Course::isA() const
{
    return courseClass;
}

char *Course::nameOf() const
{
    return "Course";
}

void Course::printOn( ostream& outputStream ) const
{
    outputStream << " " << nameOf() << " { ";
    outputStream <<courseName;
    outputStream << ", Course#: ";
    outputStream <<courseNo;
    outputStream << " }\n";
}

```

```

}

hashValueType Course::hashValue() const
{
    return courseName.hashValue();
}

int Course::isEqual( const Object& toObject ) const
{
    return courseNo == ( (Course&)toObject ).getCourseNo();
}

int Course::isAssociation() const
{
    return 0;
}

Crs_Std::~Crs_Std()
{
}

classType Crs_Std::isA() const
{
    return crs_stdClass;
}

char *Crs_Std::nameOf() const
{
    return "Crs_Std";
}

void Crs_Std::printOn( ostream& outputStream ) const
{
    outputStream << " " << nameOf() << " { ";
    outputStream << student.getPName();
    outputStream << " is taking ";
    outputStream << course.getCourseName();
    outputStream << " }\n";
}

hashValueType Crs_Std::hashValue() const
{
    return (course.hashValue()+student.hashValue());
}

int Crs_Std::isEqual( const Object& toObject ) const
{

```

```
        return (course.isEqual (((Crs_Std&)toObject).getCourse()))
            && (student.isEqual (((Crs_Std&)toObject).getStudent())));
    }

    int Crs_Std::isAssociation() const
    {
        return 0;
    }
```


Function for Creating Metadata

```
#ifndef __METACLAS_H
#include "metacclas.h"
#endif

#ifndef __STRNG_H
#include <strng.H>
#endif

#ifndef __OBJECTSET_H
#include <objectset.h>
#endif

#ifndef __UNIVSITY_H
#include <univsvity.h>
#endif
#ifndef __MAIN_H
#include "main.h"
#endif

void creatMetadata(PrimClassSet& pcSet, UserClassSet& ucSet,
    AttributeSet& aSet, MessageSet& mSet)
{
    int i, j, k;

    ObjectSet& emptySet=*(new ObjectSet(1));
    UserClass *uc[noOfUserClasses];
    char*
    cName[noOfUserClasses]={"Integer", "Character", "String",
        "Object", "Person", "Student", "Faculty", "Dept", "Course",
        "StudentSet", "FacultySet", "DeptSet", "CourseSet"};

    //Three primitive classes
    for (i=0; i<3; i++)
    {
        MessageSet& pms=*(new MessageSet(maxNoOfMessages));
        String& pcn=*(new String(cName[i]));
        PrimClass& primCls=*(new PrimClass(pcn,pms,emptySet));
        uc[i]=*(UserClass&)primCls;
        pcSet.add(primCls);
    };

    //UserClass of class Object. Object has no superclass and
    //attributes.
    String& oName=*(new String(cName[3]));
    MessageSet& oms=*(new MessageSet(maxNoOfMessages));
    UserClassSet& oSubcls=*(new UserClassSet(maxNoOfSubclasses));
    UserClass& objectMetaCls=
        *(new UserClass(oName, (AttributeSet&)NOOBJECT, oms,
            emptySet, (UserClass&)NOOBJECT,oSubcls));
}
```

```

ucSet.add(objectMetaCls);
uc[3]=&objectMetaCls;

//UserClass of class Person,Student,Faculty,Dept and Course
for(i=4;i<9;i++)
{
String& univName=*(new String(cName[i]));
MessageSet& univMsgSet=*(new MessageSet(maxNoOfMessages));
AttributeSet& univAttrSet=
    *(new AttributeSet(maxNoOfAttributes));
UserClassSet& univSubcls=*(new UserClassSet(maxNoOfSubclasses));

if ((i==4)|(i==7)|(i==8)) j=3;//superclass of Person Dept and
                        // Course is Object
else j=4;                //superclass of Student and
                        // Faculty is Person

UserClass& univSuper=*(new UserClass(*uc[j]));
UserClass& univCls=
    *(new UserClass(univName,univAttrSet,univMsgSet,emptySet,
        univSuper, univSubcls));
ucSet.add(univCls);
((UserClass&)*uc[j]).getSubclasses().add(univCls);
uc[i]=&univCls;
};

//metaClass ObjectSet, and four university objectSets metaclass
for (i=9; i<14; i++)
{
String& usName=*(new String(cName[i]));
MessageSet& univms=*(new MessageSet(maxNoOfMessages));
UserClassSet& usSubcls=*(new UserClassSet(maxNoOfSubclasses));

if (i==9) j=3; //the Superclass of ObjectSet is Object
else j=9;      //the superclass of others is ObjectSet
UserClass& super=*(new UserClass(*uc[j]));
UserClass& uSetMetaCls=
    *(new UserClass(usName, (AttributeSet&)NOOBJECT, univms,
        emptySet, super, usSubcls));
ucSet.add(uSetMetaCls);
((UserClass&)*uc[j]).getSubclasses().add(uSetMetaCls);
uc[i]=&uSetMetaCls;
};

//Attributes of class Person,Student,Faculty,Dept,Course
char* paName[noOfAttributes]={"personName", "ssn", "sex",
    "birthDate", "address", "year", "major", "registeredCourses",
    "advisor", "phone", "rank", "belongs", "supervise",
    "teach", "deptName", "deptPhone", "mainOffice",
    "facultyMembers", "rgsStudents", "courses", "courseNo",

```

```

        "courseName", "offeredBy", "attenders", "teacher"};

for (i=0;i<25;i++)
{ if (i<4) {
    j=4;                                //the owner is class Person
    if ((i==0)|(i==4)) k=2;             //domain is class String
    else k=0;                           //domain is class Integer
  }
  else if (i<9){
    j=5;                                //the owner is class Student
    if (i==5) k=0;                      //the domain is class Integer
    else if (i==6) k=7;                 //the domain is class Dept
    else if (i==7) k=13;                //the domain is class courseSet
    else k=6;                           //the domain is class Faculty
  }
  else if(i<14){
    j=6;                                //the owner is class Faculty
    if (i==9) k=0;                     //the domain is class Integer
    else if (i==10) k=0;                //the domain is class Integer
    else if (i==11) k=7;                //the domain is class Dept
    else if (i==12) k=10;               //the domain is class StudentSet
    else k=13;                          //the domain is class CourseSet
  }
  else if(i<20){
    j=7;                                //the owner is class Dept
    if (i==14) k=2;                    //the domain is class String
    else if (i==15) k=0;                //the domain is class Integer
    else if (i==16) k=2;                //the domain is class String
    else if (i==17) k=11;               //the domain is class FacultySet
    else if (i==18) k=10;               //the domain is class StudentSet
    else k=13;                          //the domain is class CourseSet
  }
  else {
    j=8;                                //the owner is class Course
    if (i==20) k=0;                    //the domain is class Integer
    else if (i==21) k=2;                //the domain is class String
    else if (i==22) k=7;                //the domain is class Dept
    else if (i==23) k=10;               //the domain is class StudentSet
    else k=6;                           //the domain is class Faculty
  };

String& pan=*(new String(paName[i]));
UserClass& ownerCls=*(new UserClass(*uc[j]));
MetaClass& domainCls=*(new MetaClass(*uc[k]));
Attribute& attr=*(new Attribute(pan, domainCls, ownerCls));

aSet.add(attr);
((UserClass&)*uc[j]).getAttributes().add(attr);
};

//Methods of classes Person,Student,Faculty,Dept and Course;

```

```

char* pmSelector[noOfMessages]=
    {"getPersonName", "getSsn", "getSex", "getBirthDate",
     "getAddress", "getYear", "getMajor",
     "getRegisteredCourses", "getAdvisor", "getPhone",
     "getRank", "getBelongs", "getSupervise", "getTeach",
     "getDeptName", "getDeptPhone", "getMainOffice",
     "getFacultyMembers", "getRgsStudents", "getCourses",
     "getCourseNo", "getCourseName", "getOfferedBy",
     "getAttenders", "getTeacher"};

for (i=0;i<25;i++)
{ if (i<4)      j=4;          //the receiver is class Person
  else if (i<9)j=5;          //the receiver is class Student
  else if (i<14)j=6;         //the receiver is class Faculty
  else if (i<20)j=7;         //the receiver is class Dept
  else          j=8;         //the receiver is class Course

  String& slct=*(new String(pmSelector[i]));
  UserClass& receiverCls=*(new UserClass(*uc[j]));
  Message& msg=*(new Message(slct, receiverCls));
  mSet.add(msg);
  ((UserClass&)*uc[j]).getMethods().add(msg);
};
}

```

Function for Creating Application Data

```
#ifndef __UNIVERSITY_H
#include <univrsity.h>
#endif

#ifndef __OBJECTSET_H
#include <objectset.h>
#endif

#ifndef __MAIN_H
#include "main.h"
#endif

void createUniversity(DeptSet& oSetD, FacultySet& oSetF,
    StudentSet& oSetS, CourseSet& oSetC, Crs_StdSet& CrsStdAssoc)
{
    int i, j, k, l[noOfStudents];
    char g[noOfStudents*noOfCourses];

    //create three instances of class Dept.

    char *dName[noOfDepts]=
    { "Mathematics", "Computer Science", "Physics"};
    long phoneNo[noOfDepts]={ 7886789, 7884555, 7887777};
    char* deptOffice[noOfDepts]={ "DT-610", "HP-565", "MZ-208"};

    Dept* department[noOfDepts];

    for ( i=0; i<noOfDepts; i++ )
    {
        String& name=*( new String( dName[i] ) );
        String& mainOff=*(new String(deptOffice[i]));
        FacultySet& facultMemb=*(new FacultySet(noOfFaculties));
        StudentSet& rStudents=*(new StudentSet(noOfStudents));
        CourseSet& coursesOffd=*(new CourseSet(noOfCourses));
        Dept& aDept=*( new Dept( name, phoneNo[i], mainOff, facultMemb,
            rStudents, coursesOffd));
        oSetD.add(aDept);
        department[i]=&aDept;
    };

    //Create five instances of class Faculty.

    char *fName[noOfFaculties]=
    {"John Smith", "Frank Wong", "Jennifer Wallance",
        "Alicia Zelaya", "Ramesh Jabbar"};
    long fssn[noOfFaculties]={ 123456789, 33344555, 999887777,
        987654321, 666884444 };
    int fsex[noOfFaculties]={ male, male, female, female,male};
}
```

```

int fbDate[noOfFaculties]={ 90155, 81245, 190758, 200631,
    150952};
char *fAddress[noOfFaculties]={ "731 Fondren,Houston,TX",
    "638 Voss,Houston,TX", "3321 Castle, Spring,TX",
    "291 Berry,Belliar,TX","975 Fire Oak,Humble,TX" };
long fPhone[noOfFaculties]={8660155, 8661245, 8660765, 8669766,
    8667231};
int rank[noOfFaculties]= {assistantProf, associateProf,
    assistantProf, prof, associateProf};

Faculty* faculty[noOfFaculties];
for ( i=0; i<noOfFaculties; i++ )
{
    String& name=*( new String( fName[i] ) );
    String& address=*( new String( fAddress[i] ) );

    if      (i<2) j=0;           //Mathematics
    else if (i<4) j=1;          //Computer Science
    else j=2;                    //Physics

    Dept&      fdept=*(new Dept(*department[j]));
    StudentSet& advStudents=*(new StudentSet(noOfStudents));
    CourseSet&  coursesTeach=*(new CourseSet(noOfCourses));
    Faculty& aFaculty=*( new Faculty( name, fssn[i], fsex[i],
        fbDate[i], address, fPhone[i], rank[i], fdept,
        advStudents, coursesTeach));
    oSetF.add(aFaculty);
    faculty[i]=&aFaculty;

//fill the faculties into their dept.
    ((Dept&)*department[j]).getFacultyMembers().add(aFaculty);
};

//Create ten instances of class Student
char *sName[noOfStudents]={ "Benjamin Bayer", "Katherine Ashly",
    "Dick Davison", "Charles Cooper", "Barbara Benson",
    "Joyce English", "Ahmad Narayan", "James Borg",
    "Shiela Whyte", "Mordy Brown"};
long sssn[noOfStudents]={ 305612335, 381621245, 422112320,
    489221100, 533691238, 467829278, 938735661, 816340021,
    100938748, 736352019};
int ssex[noOfStudents]= { male, female, male, male, female,
    female, male, male, female, male};
int sbDate[noOfStudents]= {90167, 81270, 190769, 200660, 150973,
    60572, 180762, 311272, 40568, 121067};
char *sAddress[noOfStudents]={ "2918 Bluebonnet Lane,Houston,TX",
    "125 Voss,Houston,TX", "3452 Elgin Road, Spring,TX",
    "256 Lark Lane,Belliar,TX", "7384 Fontana Lane, Humble,TX",
    "119 Friel Street, Sunnyvill, TX", "7265 Broadway Blvd.,
    St.Jose, CA", "79 Fir Oak, Humble, TX",
    "546 Springfield Road, Newmart, TX", "888 Parkdale Blvd.,
    Greenbank, TX"};

```

```

    int yr[noOfStudents]={2, 1, 3, 4, 1, 2, 4, 2, 3, 4};

    Student *student[noOfStudents];
    for ( i=0; i<noOfStudents; i++ )
    {
        String& name=*( new String( sName[i] ) );
        String& address=*( new String( sAddress[i] ) );

        if (i<2)      j=0;          //prof.Smith.
        else if (i<4) j=1;          //Prof.Wong.
        else if (i<6) j=2;          //Prof.Wallance
        else if (i<8) j=3;          //Prof.Zelaya
            else j=4;              //Prof.Jabbar

        if (j<2) k=0;              //Mathematics
        else if (j<4) k=1;         //Computer Science
            else k=2;              //Physics

        Dept& sdept=*(new Dept(*department[k]));
        Faculty& advsr=*(new Faculty(*faculty[j]));
        Student& aStudent=*( new Student( name, ssn[i], ssex[i],
            sbDate[i], address, yr[i], sdept,CrsStdAssoc, advsr ));
        oSetS.add(aStudent);
        student[i]=&aStudent;

//fill the faculties and departments with their students.
        ((Faculty&)*faculty[j]).getSupervise().add(aStudent);
        ((Dept&)*department[k]).getRgstStudents().add(aStudent);
    };

//Create six instances of class Courses
    long cNo[noOfCourses]={95203, 95305, 70301, 70205, 89306, 89101};
    char *cName[noOfCourses]={ "Data Structure", "Database Systems",
        "Statistics", "Linear Algebra", "General Physics",
        "Semiconductor Devices"};

    for ( i=0; i<noOfCourses; i++ )
    {
        if (i<2) j=1;              //set the dept that offer the course
        else if (i<4) j=0;
            else j=2;

        if (i==0) k=3;              //set the prof. who teaches the course
        else if (i==1) k=3;
        else if (i==2) k=0;
        else if (i==3) k=1;
        else if (i==4) k=4;
        else if (i==5) k=2;

        String& name=*( new String( cName[i] ) );
        Dept& cDept=*(new Dept((Dept&)*department[j]));
        Faculty& aTeacher=*(new Faculty((Faculty&)*faculty[k]));
    }

```

```

Course& aCourse=*( new Course( cNo[i], name, cDept,
    CrsStdAssoc, aTeacher));
oSetC.add(aCourse);

//put the course into Faculty and Dept
((Faculty&)*faculty[k]).getTeach().add(aCourse);
((Dept&)*department[j]).getCourses().add(aCourse);

//add elements into Course_Student association map
if (i==0) {l[2]=1;l[5]=1;l[1]=1;l[9]=1;l[0]=1;
    g[2]='A';g[5]='B';g[1]='A';g[9]='D';g[0]='F';}
else if(i==1) {l[1]=1;l[2]=1;l[6]=1;
    g[1]='C';g[2]='A';g[6]='C';}
else if(i==2) {l[4]=1;l[7]=1;l[5]=1;l[3]=1;
    g[4]='D';g[7]='A';g[5]='D';g[3]='B';}
else if(i==3) {l[3]=1;l[2]=1;g[3]='A';g[2]='C';}
else if(i==4) {l[2]=1;l[6]=1;l[1]=1;l[8]=1;l[9]=1;l[0]=1;
    g[2]='A';g[6]='B';g[1]='C';g[8]='D';g[9]='B';g[0]='F';}
else {l[2]=1;l[5]=1;g[2]='B';g[5]='A';};

for (j=0;j<noOfStudents; j++)
{
    Student& std=*(new Student((Student&)*student[j]));
    Crs_Std& cs=*(new Crs_Std(aCourse,std,g[j]));
    if (l[j]==1) { CrsStdAssoc.add (cs); l[j]=0;};
};
}

```


Class Object Definition

```
#ifndef __OBJECT_H
#define __OBJECT_H

//
//      Object
//      NOOBJECT
//      Object::Object          constructor
//      Object::Object          copy constructor
//
//      Error
//
//      operator <<
//      operator ==
//      operator !=
//
// Description
//
//      Defines the abstract base class Object.  Object is the
//      class at the root of the class library hierarchy.  Also
//      defines the instance class Error, which is used to indicate
//      the presence of no object reference.
//
//Interface Dependencies
-----

#ifndef __IOSTREAM_H
#include <iostream.h>
#define __IOSTREAM_H
#endif

#ifndef __STDDEF_H
#include <stddef.h>
#define __STDDEF_H
#endif

#ifndef __CLSTYPES_H
#include <clstypes.h>
#endif

#ifndef __CLSDEFS_H
#include <clsdefs.h>
#endif

// End Interface Dependencies
-----
```

```

// Class //

class Object
{
public:
    Object();
    Object( Object& );
    virtual ~Object();

    virtual classType      isA() const = 0;
    virtual char           *nameOf() const = 0;
    virtual hashValueType  hashCode() const = 0;
    virtual int            isEqual( const Object& ) const = 0;
    virtual int            isSortable() const;
    virtual int            isAssociation() const;

    void                  *operator new( size_t s );
    virtual void           forEach( iterFuncType, void * );
    virtual Object&        firstThat( condFuncType, void * )
const;
    virtual Object&        lastThat( condFuncType, void * ) const;

    virtual void           printOn( ostream& ) const = 0;

    static Object          *ZERO;

protected:
    friend    ostream& operator <<( ostream&, const Object& );
};

// Description
-----
//
// Defines the abstract base class Object. Object is the root
// of the hierarchy, as most classes within the hierarchy are
// derived from it. To create a class as part of this hierarchy,
// derive that class from Object and provide the required
// functions. You may then use the derived class anywhere Object
// is called for.
//
// Constructors
//
//     Object()
//
// Vanilla constructor. Forces each derived class to provide
// one, even if it's one that the compiler has to generate.
//
//     Object( Object& )
//
// Copy constructor. Constructs an object, then copies the
// contents of the given object onto the new object.
//

```

```

// Destructors
//
//   ~Object
//
//   Run-of-the-mill destructor. Turns out to be a useful place
//   to set breakpoints sometimes.
//
// Public Members
//
//   isA
//
//   Returns an unique identifying quantity of type classType.
//   You may test this quantity to make sure that the object
//   is of the class desired.
//
//   nameOf
//
//   Returns a pointer to the character string which is the
//   class name.
//
//   hashValue
//
//   Returns a unique key based on the value of an object. The
//   method used in obtaining the key depends upon the
//   implementation of the function for each class.
//
//   isEqual
//
//   Returns 1 if the objects are the same type and the elements
//   of the object are equal, 0 otherwise.
//
//   operator new
//
//   Returns ZERO if the allocation of a new object fails.
//
//   forEach
//
//   Performs a function on each of the subobjects in an object.
//   If an object has no subobject, forEach operates on that
//   object.
//
//   firstThat
//
//   Returns a reference to the first object for which the given
//   conditional function returns a 1. For object of
//   non-container classes, this will always be a reference to
//   the object.
//
//   lastThat
//
//   Returns a reference to the last object for which the given
//   conditional function returns a 1. For object of

```

```

//      non-container classes, this will always be a reference to
//      the object.
//
//      ZERO
//
//      A reference to an error object. Note that this differs
//      from a reference to a null object. This is used by the
//      Error class to handle problems when the operator new cannot
//      allocate space for an object.
//
//      printOn
//
//      Displays the contents of an object. The format of the
//      output is dictated by the implementation of the printOn
//      function for each class.
//
//      Remarks
//
//      Friends:
//          The operator << is overloaded and made of friend of the
//          class Object so that invocations of << may call the
//          protected member function, printOn.
//
//      End

```

```

// Macro //

```

```

#define NOOBJECT          *(Object::ZERO)

```

```

// Summary

```

```

//
//      Provides an easy reference to theErrorObject
//
//      End

```

```

// Constructor //

```

```

inline Object::Object()

```

```

// Summary

```

```

//
//      Default constructor for an object. Not useful for much,
//      but it does provide a good place for setting breakpoints,
//      because every time an object gets created, this function

```

```

//      must be called.
//
// End
-----
{
}
// End Constructor Object::Object //

// Constructor //
inline Object::Object( Object& )

// Summary
-----

//
//      Copy constructor for an object. Again, not useful for much
//      except breakpoints. This function will be called every
//      time one object is copied to another.
//
// End
-----
{
}
// End Constructor Object::Object //

// Class //

class Error: private Object
{
public:
    virtual ~Error();
    virtual classType      isA() const;
    virtual char           *nameOf() const;
    virtual hashValueType  hashValue() const;
    virtual int            isEqual( const Object& ) const;
    virtual void            printOn( ostream& ) const;
    virtual void            operator delete( void * );
};

// Description
-----
//
//      Defines the class Error. The is exactly one instantiation
//      of class Error, namely theErrorObject. The static object
//      pointer Object::ZERO points to this object. The define
//      NOOBJECT redefines Object::ZERO (see CLSDEFS.H). The
//      operator Object::new returns a pointer to theErrorObject
//      if an attempt to allocate an object fails. You may test
//      the return value of the new operator against NOOBJECT to

```

```

//      see whether the allocation failed.
//
// Public Members
//
//      isA
//
//      Returns the correct value for the Error class.
//
//      nameOf
//
//      Returns a pointer to the character string "Error".
//
//      hashValue
//
//      Returns a pre-defined value for the Error class. All
//      objects of class Error (there is usually only one,
//      theErrorObject) have the same hash value. This makes them
//      hard to distinguish from each other, but since there's only
//      one, it doesn't matter.
//
//      isEqual
//
//      Determines whether the given object is theErrorObject.
//
//      printOn
//
//      Overrides the default printOn function since the Error
//      class is an instance class.
//
// End

```

```

// Friend //

```

```

inline ostream& operator <<( ostream& outputStream, const Object&
anObject )

```

```

// Summary

```

```

//
//      Write an object value to an output stream.
//
// Parameters
//
//      outputStream
//      The stream on which to display the formatted contents of
//      the object.
//
//      anObject
//      The object to display.

```

```

//
// End
-----
{
    anObject.printOn( outputStream );
    return outputStream;
}
// End Friend operator << //

// Function //

inline int operator ==( const Object& test1, const Object& test2
)

// Summary
-----
//
//     Determines whether the first object is equal to the second.
//     We do type checking on the two objects (objects of
//     different classes can't be equal, even if they're derived
//     from each other).
//
// Parameters
//
//     test1
//
//     The first object we are testing.
//
//     test2
//
//     The second object we are testing.
//
// End
-----
{
    return ( (test1.isA() == test2.isA()) && test1.isEqual( test2
) );
}
// End Function operator == //

// Function //

inline int operator !=( const Object& test1, const Object& test2
)

// Summary
-----
//
//     Determines whether the given object is not equal to this.

```

```

//      We just reverse the condition returned from operator ==.
//
// Parameters
//
//      test1
//
//      The first object we are testing.
//
//      test2
//
//      The second object we are testing.
//
// End
-----
{
    return ( !( test1 == test2 ) );
}
// End Function operator != //

#endif // ifndef __OBJECT_H //

```


Methods Implementation for Class Object

// Interface Dependencies

```
-----  
#ifndef __OBJECT_H  
#include <object.h>  
#endif
```

// End Interface Dependencies

Object::~Object()

// Summary

```
-----  
//      Default destructor for an object. Doesn't do much, but it  
//      forces all classes derived from Object to have virtual  
//      destructors, which is essential for proper cleanup. It  
//      also provides a good place for setting breakpoints, because  
//      every time an object gets destroyed, this function will be  
//      called.  
// End
```

```
-----  
{  
}
```

// End Destructor Object::~Object //

// Member Function //

int Object::isSortable() const

// Summary

```
-----  
//  
//      indicates whether the object defines comparison operators  
//  
// Parameters  
//  
//      none  
//  
// Remarks  
//  
//      A basic Object is not sortable
```

```

//
// End
-----
{
    return 0;
}
// End Member Function Object::isSortable //

// Member Function //
int Object::isAssociation() const

// Summary
-----

//
//     indicates whether the object is derived from class
//     Association
//
// Parameters
//
//     none
//
// Remarks
//
//     A basic Object is not derived from class Association
//
// End
-----
{
    return 0;
}
// End Member Function Object::isAssociation //

// Member Function //
void *Object::operator new( size_t s )

// Summary
-----

//
//     replacement for the standard operator new(). Returns ZERO
//     if attempted allocation fails.
//
// Parameters
//
//     s
//
//     number of bytes to allocate

```

```

//
// Functional Description
//
//      we call the global operator new() and check whether it
//      succeeded. If it succeeded, we return the block that it
//      allocated. If it failed, we return ZERO.
//
// End
-----

{
    void *allocated = ::operator new( s );
    if( allocated == 0 )
        return ZERO;
    else
        return allocated;
}
// End Member Function Object::operator new //

// Member Function //

void Object::forEach( iterFuncType actionPtr, void *paramListPtr
)

// Summary
-----

//
//      Calls the given iterator function on this object.
//
// Parameters
//
//      actionPtr
//
//      Pointer to the action routine which is to be called for
//      this object.
//
//      paramListPtr
//
//      Pointer to the list of parameters which will be passed
//      along to the action routine.
//
// Functional Description
//
//      We call the given function, passing our object and the list
//      of parameters that was given to us.
//
// Remarks
//
// warnings:
//      The action routine must have a prototype of the form:

```

```

//          void action( Object&, void * );
//
// End
-----
{
    ( *actionPtr )( *this, paramListPtr );
}
// End Member Function Object::forEach //

// Member Function //

Object&  Object::firstThat(  condFuncType  testFuncPtr,  void
*paramListPtr ) const

// Summary
-----

//
//      Calls the given conditional test function on this object.
//
// Parameters
//
//      testFuncPtr
//
//      Pointer to the conditional test routine which is to be
//      called for this object.
//
//      paramListPtr
//
//      Pointer to the list of parameters which will be passed
//      along to the conditional test routine.
//
// Return Value
//
//      Returns this if the this satisfies the condition.  Returns
//      NOOBJECT otherwise.
//
// Functional Description
//
//      We call the given function, passing our object and the list
//      of parameters that was given to us.  If the function
//      returns a 1, we return this object, otherwise we return
//      NOOBJECT.
//
// Remarks
//
//      warnings:
//      The conditional test routine must have a prototype of the
//      form:
//      int test( Object&, void * );
//      The conditional test routine must return 1 if the given

```

```

//      object satisfies the condition.
//
// End
-----
{
    if( ( *testFuncPtr )( *this, paramListPtr ) )
    {
        return( *this );
    }
    else // our object doesn't satisfy the condition //
    {
        return( NOOBJECT );
    }
}
// End Member Function Object::firstThat //

// Member Function //

Object&  Object::lastThat(   condFuncType   testFuncPtr,   void
*paramListPtr ) const

// Summary
-----

//
//      Calls the given conditional test function on this object.
//      For non-container objects, lastThat is the same as
//      firstThat.
//
// Parameters
//
//      testFuncPtr
//
//      Pointer to the conditional test routine which is to be
//      called for this object.
//
//      paramListPtr
//
//      Pointer to the list of parameters which will be passed
//      along to the conditional test routine.
//
// Functional Description
//
//      We call the firstThat function.
//
// Remarks
//
//      warnings:
//      The conditional test routine must have a prototype of the
//      form:
//      int test( Object&, void * );

```

```

//      The conditional test routine must return 1 if the given
//      object satisfies the condition.
//
// End
-----
{
    return Object::firstThat( testFuncPtr, paramListPtr );
}
// End Member Function Object::lastThat //

// Destructor //
Error::~Error()

// Description
-----
//
//      We can't really destroy theErrorObject.
//
// End
-----
{
}
// End Destructor Error::~Error //

// Member Function //
void Error::operator delete( void * )

// Summary
-----

//
//      Can't delete an Error object... so we pretend that we did.
//
// End
-----
{
}
// End Member Function Error::operator delete //

// Member Function //
classType Error::isA() const

// Summary
-----

//

```

```

//      Returns the class type of the error object.
//
// End
-----
{
    return errorClass;
}
// End Member Function Error::isA //

// Member Function //
char *Error::nameOf() const

// Summary
-----

//
//      Returns a pointer to the character string "Error."
//
// End
-----
{
    return "Error";
}
// End Member Function Error::nameOf //

// Member Function //
void Error::printOn( ostream& outputStream ) const

// Summary
-----

//
//      Error class override of the usual printOn. Since there
//      isn't really any object to print, we emit an appropriate
//      message.
//
// Parameters
//
//      outputStream
//      The stream on which to display the formatted contents of the
//      object.
//
// End
-----
{
    outputStream << "Error\n";
}
// End Member Function Error::printOn //

```

```

// Member Function //
hashValueType  Error::hashValue() const

// Summary
-----

//
//      Returns the value for use when hashing an error object.
//      There should be only one object of class Error, so it's ok
//      to return the same value every time.
//
// End
-----
{
    return ERROR_CLASS_HASH_VALUE;
}
// End Member Function Error::hashValue //

// Member Function //
int Error::isEqual ( const Object& testObject ) const

// Summary
-----

//
//      Determines whether the given object is theErrorObject.
//
// Parameters
//
//      testObject
//
//      The object we are testing against theErrorObject.
//
// Return Value
//
//      Returns 1 if the given object is theErrorObject, 0
//      otherwise.
//
// Functional Description
//
//      The only way we get called here is if this is a pointer to
//      theErrorObject. We test the address of our given object
//      to see if it is the address of theErrorObject.
//
// End
-----
{

```



```

        return &testObject == this;
    }
    // End Member Function Error::isEqual //

    // Variable //
    Error    theErrorObject;

    // Description
    -----
    //
    //      Defines a dummy object to which Object::ZERO will point.
    //      We only need this so we don't ever try to dereference a null
    //      pointer.
    //
    // End
    -----

    // Initializer //
    Object *Object::ZERO = (Object *)&theErrorObject;

    // Description
    -----
    //
    //      Initializes Object::ZERO. We wait to do this here because
    //      we have to get theErrorObject defined before we initialize
    //      Object::ZERO.
    //
    // End
    -----

```

END

2/3·1 2·92

FIN