AN OBJECT MEMORY FOR AN OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND
INFORMATION SCIENCES
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
F. Nihan Kesim
July 1988

AN OBJECT MEMORY FOR AN OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING AND

INFORMATION SCIENCES

AND THE INSTITUTE OF ENGINEERING AND SCIENCES

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

F. Nihan Kesim F. Nihan Kesim

July 1988 tarafından bağışlanmıştır.

QA
76.9
.D3
K481
1988

B 1868

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Prof. Dr. Erol Arkun (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Bülent Özgüç

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Murat Tanık

Assoc. Prof. Murat Tanık

Approved for the Institute of Engineering and Sciences:

M. Baray

Prof. Dr. Mehmet Baray, Director of Institute of Engineering and Sciences

ABSTRACT

AN OBJECT MEMORY FOR AN OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM

F. Nihan Kesim
M.S. in Computer Engineering and
Information Sciences
Supervisor: Prof. Dr. Erol Arkun
July 1988

Object-oriented paradigm is an approach that can be applied in various areas of computing. In this approach, each entity is represented by an object which captures the state and the behaviour of the entity. In this thesis, a focused survey of object-oriented paradigm in general and object-oriented database management systems in particular has been carried out and an object memory module is designed and implemented for an object-oriented database management system prototype. The object memory module handles the representation, access and manipulation of objects in the system and provides the primitive functions that are necessary in the development of the prototype.

Keywords : object-oriented database management system, object, class, method, message, data abstraction, encapsulation, inheritance, class hierarchy, object memory, message passing.

ÖZET

NESNESEL BİR VERİ TABANI SİSTEMİNDE NESNE BELLEĞİ

F. Nihan Kesim
Bilgisayar Mühendisliği ve Enformatik Bilimleri Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Erol Arkun
Temmuz 1988

Çeşitli bilgisayar kullanım alanlarında uygulanabilen nesnesel yaklaşımda her bir varlık, kendi durumunu ve işlevlerini kapsayan bir nesne olarak modellenir. Bu tezde nesnesel yaklaşım kavramı ve nesnesel veri tabanı işletim sistemleri üzerinde araştırma yapılmış ve bir nesnesel veri tabanı sistemi prototipi için nesne belleği tasarlanıp gerçekleştirilmiştir. Nesne belleği, nesnelerin gösterimini, erişimini, kullanımını ve bütün prototip sistemin geliştirilmesi için gereken temel fonksiyonları sağlar.

Anahtar kelimeler : nesnesel veri tabanı sistemleri, nesne, sınıf, metod, mesaj, aktarım, veri soyutlaması, sınıf hiyerarşisi, nesne belleği, mesaj yollama.

ACKNOWLEDGEMENT

I wish to express my considerable gratitude to my supervisor Prof. Dr. Erol Arkun who has given me guidance and encouragement throughout the development of this thesis. I would also like to thank Dr. Nierstrasz for his valuable remarks during his visit at Bilkent University. I am especially grateful to my friends Sibel Özelçi and Murat Karaorman for their comments and collaboration on this work. My sincere thanks are due to Oğuz Gülseren for his kindly helps in bringing the thesis to its final form and to İmren Kızıler for her typing part of the thesis. I owe a great debt of thanks to my family who have stood by me and supported me well beyond the call of duty. Finally, I would like to thank all the research assistants of the Engineering and Science Faculty of Bilkent University for their morale support during this study.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

With the increasing efficiency of computer systems, the sophistication and demand of the users have been increasing. There have been some important changes in both general-purpose programming and database languages. In general purpose programming, the improvement started with assemblers and went to high-level languages and more recently to logic and functional languages. In database languages, from navigational models more declarative relational models have been reached. Especially, in newer data-intensive applications such as computer aided design and manufacturing (CAD/CAM), document retrieval, expert systems and decision support systems, the need to introduce more powerful data modeling concepts in both programming languages and database models became obvious. This resulted in the development of object-oriented programming environments and this approach was extended to other fields. Although there is no clear definition of what object-orientation is, the basic characteristic of this approach is that instead of passing data to procedures as in the traditional data processing methods, the objects which represent real world entities are requested to perform operations on themselves.

Informally, an object-oriented database management system can be defined as follows. It is a system which is based on a data model that allows to represent an application entity, whatever its complexity and structure, by exactly one object of the database. Thus no artificial decomposition into simpler concepts is necessary. As entities might be composed of subentities which are entities themselves, an object-oriented data model has to allow recursively composed objects.

Object-orientation is very suitable for database applications. Conventional record-oriented database systems reduce application development time and improve data sharing among applications. However they are subject to the limitations of a finite set of data types and the need to normalize data. Furthermore, there is a semantic gap between the application semantics and its database representation. In contrast, object-oriented systems offer flexible abstract data-typing facilities and the ability to encapsulate data and operations with the message metaphor. In addition, they reduce application development efforts. Also, one can easily represent models which can not be represented using normalized relations, thus keeping the semantic gap as small as possible and representing most of the problem semantics in the database itself. Another point is that, object-oriented systems aim at solving the impedance mismatch problem [41] seen in conventional database systems in which there are separate data definition and data manipulation languages by providing a single language.

In this thesis, a focused survey on object-oriented paradigm and object-oriented database management systems has been carried out and a single-user memory-based object-oriented database management system prototype has been designed and implemented. The designed and implemented object-oriented database management system prototype consists of four major modules which are object memory and schema evolution; message passing; secondary storage management, indexing and the user interface [30] [16]. The implementation of the system has been carried out on Sun workstations running Berkeley Unix[1] 4.2 and using the C programming language [38] [6] [17] [18]. The goal of this study was to understand the fundamental concepts of object-orientation, realize the possible difficulties in the implementation of such systems and discover the open problems for future research.

The first part of the thesis presents the results of the survey. The basic concepts, characteristics and application areas of object-oriented approach are introduced. Then the application of this approach in programming languages and in database systems is explained by giving examples of the existing systems.

[1]Unix is a trademark of AT&T Bell Laboratories.

In the second part, the designed object-oriented database management system prototype is presented. The object memory module is explained in detail and the functions of the other modules are summarized. Finally, the current research issues in the object-oriented database systems are stated.

2. GENERAL PROPERTIES OF OBJECT-ORIENTATION

There is much confusion about the term *object-oriented.* It appears that the concept can be applied to anything from an operating system to an interface for a text editor. There is no clear way of telling whether a system is "really" object-oriented or not. In fact, object-orientation is an approach that can be applied in various areas of computing [27].

The basic principles of the object-oriented approach are *encapsulation* and *inheritance.* Encapsulation means that, an *object* packages an entity and the operations that apply to it and inheritance enables specialization of existing objects. In object-oriented systems, all conceptual entities are modeled as objects. An integer, a string or more complex entities such as an employee or an aircraft is an object.

2.1 Basic Concepts

An object consists of a *private memory* and a *public interface part* defined by a set of *messages.* The private memory is made up of the values for a collection of *instance variables.* The value of an instance variable is itself an object and therefore has its own private memory. A simple object, such as an integer or a character, has no instance variables. A simple object has a value which itself is an object.

A message is a request for an object to access, modify or return a portion of its private memory [4]. A message specifies which operation is desired, but not how that operation should be carried out. They invoke methods which describe how to carry out the operations that apply to the object. Methods are not visible from outside of the object. An important property of an object is that, its private memory can be manipulated only by its own methods, and the messages are the only way to invoke an object's methods. Therefore, the set of messages to which an object can respond constitutes its *interface*.

A typical application may create and reference a large number of objects. If every object is to carry its own instance variables and its own methods, the amount of information to be specified and stored can become unmanageably large. For this reason and also for conceptual simplicity, similar objects are grouped together into a *class* [4]. All objects belonging to the same class are described by the same instance variables and the same methods. They all respond to the same messages. Objects that belong to a class are called *instances* of that class. A class describes the instance variables of its instances and the methods that are applicable to its instances. Thus when a message is sent to an instance, the method which implements that message is found in the definition of the class. Classes are not only collections of objects, but also templates for the creation of objects. A class provides the create and destroy operations for its instances. Since classes are also objects, a message is sent to a class when a new instance of that class is created.

The class concept provides modularization and conceptual simplicity as well as reducing duplication, since all messages, methods and instance variables shared by the instances only appear in the corresponding class definition. Another such tool is *inheritance* [4] [24] : Classes are organized in a *hierarchy*. A class hierarchy is a hierarchy of classes in which an edge between a node and a child node represents the IS-A relationship; that is the child node is a specialization of the parent node and conversely, the parent node is a generalization of the child node. The parent node is called the *superclass* of the child, and the child node is called the *subclass* of the parent. Objects at any level of this hierarchy *inherit* all instance variables and methods of higher level objects. A class needs to inherit properties only from its immediate superclass. So by induction, a class inherits properties from every class in its superclass chain. A subclass may modify the definitions and implementations it inherits from its superclasses or may add new ones.

There are two types of inheritance : simple inheritance and multiple inheritance. In *simple inheritance* a class may have a single superclass [4]. Thus the class hierarchy is restricted to being a tree. In *multiple inheritance* a class may have more than one superclass, and the class hierarchy becomes a directed acyclic graph (or a lattice structure) [4] [37] [13]. In multiple inheritance, a class inherits the union of variables and methods from all its superclasses. Multiple inheritance increases sharing by making it possible to combine definitions from several classes. It also simplifies data modeling and often requires fewer classes than are required with simple inheritance.

In both types of inheritance, there is the name conflict problem. Name conflicts occur when two or more classes have instance variables or methods with the same name. Two types of inheritance conflicts may arise. One is the conflict between a class and its superclass. The other is between the superclasses of a class and occurs only in multiple inheritance. The name conflict between a class and its superclass is solved by giving precedence to the definitions of the instance variables or methods of the class. In this way the definitions local to the class overrides the definitions in its superclass. The name conflict between the superclasses can be solved in two ways. Either all instance variable or method names in the superclasses must be distinct or a priority order is accepted on the superclasses. The priority order is specified by starting with the first superclass in the superclass specification and proceeding depth-first up to joins.

In all object-oriented systems, each object is associated with an identifier, which is unique and is never reused for another object [19] [5]. Identity concept makes it possible for objects to be distinguished from one another regardless of their content. One powerful technique for supporting identity is through surrogates. Surrogates are system generated globally unique identifiers, completely independent of any physical location or object values [19].

2.2 Characteristics of Object-Oriented Approach

To fully support object-oriented approach, a system must show the following five characteristics [27]:

1. Data Abstraction
2. Independence
3. Message passing paradigm
4. Inheritance
5. Homogeneity

Data Abstraction

The most important concept in the object-oriented approach is data abstraction. That is, the behaviour of an object rather than its implementation is of interest. Every object has a clearly defined interface which is independent of the object's internal representation. The interface provides the communication to other objects. This representation independence makes it easier to experiment with different implementations of an object and increases the portability of software. The class concept provides the data abstraction.

Independence

Independence states that objects have control over their own state and existence. The state of an object can only be changed by its own methods. Once an object is created, it will continue to exist, unless it is detected that the object neither refers to nor is accessible from another object. Another form of independence is the ability to add new object types at run-time. When new types are created dynamically, the new type objects and the old objects must be able to communicate with each other.

Message passing paradigm

Message passing is a model for object communication. Independence of objects is supported by the message passing paradigm. In this paradigm an object can interact with other objects only by sending and receiving messages. When an object sends a message to another object, one of the methods of that object is invoked. In fact the receiver object is free in the interpretation of the message. It may delay responding or it may refuse to handle the request or it may perform the method and return the result. Returning something is again accomplished by message passing. When there is no concurrency, message passing is implemented just as procedure calls.

In the case of a concurrent environment, message passing may be synchronous or asynchronous. In asynchronous message passing the message is put in a queue, and the sender can switch to another task. In synchronous message passing, the sender blocks until the response to the message is received.

Inheritance

One of the most important characteristics of the object-oriented approach is inheritance. Inheritance provides the ability to specialize object classes. A specialized class inherits properties (i.e., the instance variables and the methods) of its superclass, and also may add more properties. An instance of a subclass responds to all messages that its superclass understands but the reverse of this is not true. The superclass cannot respond to the messages corresponding to the new methods that the subclass adds.

Inheritance can be considered in four categories [13]:

- Type theory inheritance : It is related to the similarity of structure between a subclass and a superclass. Here a subclass contains all instance variables of the superclass. In addition, it can have more instance variables.
- External interface inheritance : Here a subclass provides all the methods of the superclass and can also provide additional methods.

- Code sharing and reusability : Here a subclass can use the methods provided by its superclass as if they were defined in the subclass itself. Hence multiple copies of the same code are eliminated. Thus more complex programs can be built out of simpler ones.
- Polymorphism : It is related to operator overloading and allows a concrete operation to inherit its definition and properties from a generic operation.

Conventional object-oriented systems combine some or all of these kinds of inheritance into one structure.

Homogeneity

In a fully "object-oriented" system, everything is an object. That is, a class or a method or an instance is an object. This notion provides a very consistent view of the system. But there are some points that must be considered. How far the principle of homogeneity is carried must be decided. For example, if the messages are objects also, then to manipulate them, another message should be sent. Another example is thinking of instance variables as objects. This provides constructing complex objects whose parts are also objects. Again the circularity must be considered. Because, it is not efficient to treat "parts" of an integer as objects. To break such kinds of circularity certain basic objects may be accepted and then all other objects can be built up from them.

2.3 Application Areas

The object-oriented approach first appeared in programming languages and then applied in other areas. The major application areas of the object-oriented approach are programming languages, database management systems, knowledge representation, CAD/CAM systems and office information systems [41].

Knowledge representation and CAD/CAM systems require unifying the treatment of data and metadata. They require versions and multiple design transaction support.

Object-oriented approach can easily support menu and icon interfaces and multimedia document management. Therefore it is appropriate for office information systems.

In the following sections, the application of the approach in programming languages and database systems will be explained.

3. OBJECT-ORIENTED APPROACH IN PROGRAMMING LANGUAGES

In recent years object-oriented programming has gained a great popularity in the design and implementation of emerging data-intensive application systems. These include artificial intelligence (AI), computer-aided design and manufacturing (CAD/CAM) and office information systems (OIS) with multimedia documents. Object-oriented programming offers a number of important advantages for these applications over traditional control-oriented programming, which are discussed in this chapter.

3.1 General Properties of Object-Oriented Programming

Most programming languages support the "data-procedure" paradigm. That is, active procedures act on passive data that are passed to them as parameters. For example, a square root function takes a number as its input parameter and returns the square root of that number. Data type is also important. In the above example, if a strongly typed language is used, then for each data type of the input parameter, a different version of the square root function would be implemented. On the other hand a late-binding language such as LISP detects the data type of the number at run time and performs the appropriate operations for that data type. Object-oriented languages use a data (object) centered approach. Here, data are not passed to procedures, but they are asked to perform operations on themselves. For example, to take the square root of a number, the number is asked to perform the operation on itself. So the number is the receiver of the message "square root".

In object-oriented languages, objects combine the properties of procedures and data since they perform computations and save local state. All of the action in object-oriented programming comes from sending messages between objects. Message sending is a form of indirect procedure call. Objects respond to messages using their own procedures. Message sending supports data abstraction. The principle is that calling programs should not make assumptions about the internal representations of data types they use. A data type is implemented by choosing a representation for values and writing a procedure for each operation. A language supports data abstraction when it has a mechanism for collecting together all of the procedures for a data type. Thus class concept represents the data types [32].

Another feature of object-oriented programming is *operator overloading.* Operator overloading is the use of the same operator symbol to denote distinct operations on different data types (e. g. it may be possible to use the minus sign to denote both integer difference and set difference). The meaning of the operator can be resolved only by the data types of its operands. Operator overloading provides the usage of the same message for different methods in different classes. When the message is sent to an object, it is first bound to the class of that object and then the method for that class is executed. When operators have two or more operands, one operand (usually the first one) must be selected as the message receiver that controls the overloading and the others are treated as message arguments. However there is an unfairness if the computation is based only on the type of the first argument. For example,

Multiply (x: real, y: real)
and
Multiply (x: real, y: vector)

should be understood as real number multiplication and scalar-vector multiplication, respectively. Such a distinction can be done only if all argument types are considered as characterizing the function to be applied [41].

The advantages of overloading may be more apparent with the following example. For instance, an application requires the printing of different objects, each with their own format. The object-oriented approach gives the responsibility onto the objects themselves. Each object is sent exactly the same message, print, so that it will print itself in the proper way. The addition of new types only requires writing new procedures for the common operations. So, new objects with their own print method, can simply be added and no further program modification will be required.

Inheritance concept enables programmers to create new classes of objects by specifying the differences between a new class and an existing class, instead of starting from scratch each time. A large amount of code can be reused by this way. During execution, the search for an attribute begins at some level of the class hierarchy and proceeds to the top, taking the first instance of the attribute that is found. The mechanism to add new behaviour to an existing class is language dependent. In most object-oriented languages, such as Smalltalk, this is accomplished by embedding a message-send to the pseudovariable super in the new definition of a method. Syntactically pseudovariables are treated the same as normal variables. However, their semantics are different. They cannot be assigned a new value. There are two important pseudovariables self and super. Both of them refer to the object that receives the message currently being processed. The difference between the two lies in the way that the message lookup is performed. When self is sent a message, the message lookup is performed as when the message is sent externally, starting in the object's direct class. When super is sent a message, the lookup is performed starting in the superclass of the class in which the method is currently being executed. The superclass where the method is found, is not necessarily the superclass of the object's class. This pseudovariable mechanism gives objects a controlled way of accessing superclass methods.

3.1.1 Object Identity in Programming Languages

Most programming languages employ user-defined names (i.e. variable names) to represent identity. The actual binding of an object to its name can be dynamic or static. This approach mixes addressability and identity, although the concepts are quite different. Addressability is external to an object within a particular environment and is therefore environment dependent. On the other hand, identity is internal to an object. Its purpose is to provide a way to represent the individuality of an object independently of how it is accessed. Object-oriented languages provide separate mechanisms for these concepts, so that neither is compromised [19].

A single object may be accessed in different ways and bound to different variables. There must be some way to find out if these variables refer to the same object. For example an employee object may be accessed as the manager of some department and bound to variable x. The same employee object may be accessed as an employee with a specified status, and bound to another variable y. Object-oriented languages provide both identity test operators and equality test operators. Also different copy operators are supported. There are two ways to make copies of an object. These are *shallow copy* and *deep copy*. The distinction is whether or not the values of the object's variables are copied. In shallow copy the values are not copied but they are shared. When the copy is changed another object with the new value is created and the value of the original object remains the same. In deep copy the values are copied, they are not shared. As an example, a shallow copy of an array refers to the same elements as in the original array, but the copy is a different object. Replacing an element in the copy does not change the original. A deep copy of the array, is a new array with its own identity whose elements are also new objects with their own identity but which have the same values as those of the original array. Figure 3.1 shows the relationship between shallow and deep copying.

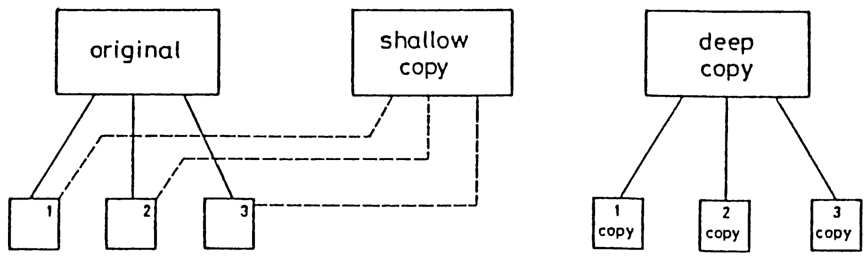

Figure 3.1: Shallow copy and deep copy

3.1.2 Advantages of Object-Oriented Programming

Object-oriented languages have many advantages over traditional procedure-oriented languages. Data abstraction and information hiding, that is the hiding of internal representation and implementation of an object, increase reliability and modifiability of software systems by reducing interdependencies. In addition, since the internal state variables and methods are not directly accessed, a carefully designed interface may permit the internal data structures and procedures to be changed without affecting the implementation of other modules.

Also dynamic binding and operator overloading increases flexibility by permitting the addition of new classes of objects (data types) without having to modify existing code. Inheritance and dynamic binding together permits code to be reused. This has the advantage of reducing overall code and increasing programmer productivity.

Object-orientation provides a natural way to translate the real world problem into a program. Because working with objects seems more natural than working with constructs found in standard languages. Dividing a problem into objects and defining actions for those objects simplifies programs. Also the object-message paradigm provides promotion to a more modular system.

Therefore object-oriented programming provides major advantages in the production and maintenance of software. It requires shorter development times and provides a high degree of code sharing. These advantages make object-oriented programming an important technology for building complex software systems now and in future [32].

3.1.3 Disadvantages of Object-Oriented Programming

Object-oriented languages have some characteristics that are considered disadvantages by some. The one most often debated is the run-time cost of the dynamic binding mechanism. A message-send takes more time than a straight procedure call [32].

Another disadvantage is that implementation of object-oriented languages is more complex than procedure-oriented languages, since the semantic gap between these languages and typical hardware machines is greater.

Another potential problem is that a programmer must learn a large class library before becoming proficient in an object-oriented language. As a result object-oriented languages are more dependent on good documentation and development tools.

3.2 Some Examples of Object-Oriented Programming Languages

Many of the ideas behind the object-oriented programming have roots going back to SIMULA [13] [37], which is an Algol-based simulation language. The first substantial interactive, display-based implementation was the Smalltalk language [12]. The object-oriented style has often been supported for simulation programs, systems programming, graphics and artificial intelligence programming. There are probably fifty or more object-oriented programming languages now in use, mostly with very limited distribution [37]. Below, only three object-oriented languages will be summarized. These are Smalltalk, Smallworld and Hybrid.

3.2.1 Smalltalk

Smalltalk is probably the best-known object-oriented language [7] [10] [12]. In Smalltalk, there are two basic units:objects and classes. Classes contain function definitions (methods) and data declarations. Every object is an instance of some class. The top-level superclass is class *Object*. All classes are refinements of the superclass Object. They add new or different methods or allow more variables in their instances. Classes themselves are objects and are instances of classes, called *metaclasses*. Metaclasses are instances of the class MetaClass. Because a class is an object, it cannot contain its own specialized methods. These special methods are kept in the metaclass of the class. In other words, a superclass is a more general version of a class, while a metaclass specifies the operations that can be performed on a class.

Smalltalk objects interact by exchanging messages. In addition to message passing, different objects of a class can share variables, called *class variables*. Class variables are defined in the metaclass of the class and are accessible to any method defined in the class.

Smalltalk supports simple inheritance. That is each class has a unique superclass. In Smalltalk, control structures are also objects. That is, a block expression is an object that can be executed by sending it the message value.

Smalltalk's object-method paradigm is used for menu-based interfaces. For example, the user specifies an object by pointing its icon with a mouse, and then selects the operation from a small menu.

Smalltalk is interpreted, rather than compiled. This obviously degrades performance.

3.2.2 Smallworld

Smallworld is a shell language that uses the object-oriented approach [21]. Smallworld is not intended to be a programming language. Rather, it is a system for organizing files, programs and system commands. The object model is different from that of Smalltalk. A Smallworld object is an independent entity, not an instance of some class. An object is a collection of properties. Each property has a name and a value. Smallworld methods are merely properties with the suffix `.method`. Objects are grouped into classes, as defined by the class property of each object. Classes are used only for organizational purposes. The member objects of a class need not have the same or even similar structures.

A Smallworld object can define its own properties and methods. If an object does not have a requested property p or method m, it then asks its class for property *inherited:p* or method *inherited:m.* If not found, the request follows the superclass chain until it reaches the top-level class, *Universe.* Smallworld classes are normal objects. As a result, they can customize their own methods, eliminating the need for metaclasses. Furthermore, classes can provide inherited properties, and this permits default values to be inherited from a class. Smallworld also supports simple inheritance.

3.2.3 Hybrid

Hybrid is an object-oriented programming language in which objects are active entities [29]. Active objects are persistent and concurrent. Thus they unify the concept of an object with those of processes and files. Hybrid is an attempt to provide a fully object-oriented language that is strongly-typed and concurrent. The goal is to design a programming language based on a small number of concepts that support reusable code, reliability through well-defined interfaces, and concurrency.

In Hybrid, objects are structured, and may contain other objects. A small set of constructors is available for structuring objects and defining object "types". A unit of concurrency is called a *domain* and is comparable to a process. Each domain contains a single top-level "root" object and any number of subobjects. Independent objects, in different domains may execute concurrently. An *activity* is defined as a thread of control, which may pass between domains when independent objects communicate. Activities may be scheduled by objects through the use of *delay queues.* Objects may switch between several activities by delegating calls to other objects. The notion of *subactivities* manages sets of related activities. Longer term mutual exclusion and atomicity are provided by transaction mechanism.

4. OBJECT-ORIENTED APPROACH IN DATABASE SYSTEMS

Merging of the object-oriented programming language and data model ideas has given rise to the idea of applying the object-oriented approach to the database field. Object-oriented languages offer flexible abstract data typing facilities and the ability to encapsulate data and operations via the message paradigm. Combining object-oriented language capabilities with the storage management functions of a traditional data management system would result in a system that reduces application development efforts and increases modeling power.

4.1 Importance of a Data Model

Every database is a model of some real world system. At all times, the contents of a database are supposed to represent the semantics of an application environment as completely and accurately as possible. Also, each change to the database should reflect an event occurring in that environment. Therefore it is expected that the structure of a database reflects the structure of the system that it models.

The data model supported by a database management system defines a framework of concepts that can be used to express the application semantics. The primary purpose of any data model, is to provide a formal means of information representation and a formal means of manipulating such a representation. A data model consists of three components [9] :

- a collection of object types
- a collection of operators
- a collection of general integrity rules.

The *object types* are the basic building blocks of the data model. For example in the relational model the objects are relations and domains. The *operators* provide a means for manipulating a database that is composed of valid instances of the object types. The relational operators are those of the relational algebra. The *integrity rules* constrain the set of valid states of the database that conform to the model. In relational model for example, there are two integrity rules : entity integrity rule and referential integrity rule.

For a particular model to be useful for a particular application, there must exist some simple correspondence between the components of that model and the elements of that application. In other words the process of mapping elements of the application into constructs of the model must be straightforward. One of the objectives of data models is to keep the gap between the semantics of the application itself and the semantics of the application as represented in the database as small as possible.

Today's database systems are mostly based upon one of the classical data models : hierarchical, network or relational. The data structures provided by these data models have significantly limited capabilities to express the meaning of a database and to relate a database to its corresponding application environment. In these data models, one conceptual entity has to be represented by a number of database objects (e.g. records, tuples). Since classical business/ administrative database applications usually deal with rather simple entities, traditional data models may be adequate for them. But this is not true for applications like VLSI-design, image processing and office automation. In these areas, entities usually show very complex internal structures and may consist of a larger number of properties [41].

In response to the inadequacies of the traditional data models, several semantic data models [1] [2] [14] have been proposed. The basis of all these studies is to use data models that provide the representation of a large portion of the meaning of the data in the database [20] [26]. The basic rule of semantic modeling is that the model represents data about objects and relationships between them in a direct manner. Such object-based modeling has given rise to the object-oriented data modeling and hence object-oriented database systems. The basic object-oriented concepts form the basis of an object-oriented data model. Hence, it is a data model that allows to represent one application entity whatever its complexity and structure, by exactly one object of the database. Thus no artificial decomposition into simpler concepts is necessary.

Before discussing the characteristics of an object-oriented database system, it will be better to specify the shortcomings of the database systems that are based on the traditional data models.

4.2 Limitations of Existing Data Models

Although record-based data models have been successfully applied to a variety of data processing problems, they have also serious limitations. A fundamental problem of these models is that they provide a finite set of data types and need to normalize data. Thus they cannot easily handle the semantic connections present in most real world data. They support a fixed set of simple types, such as integer, real and character string. However, they do not have support for defining new types and for adding operations to these types. The constructors for abstract types are limited. The relational model supports "tuple"and "relation"; the hierarchical model supports "segment" and "tree of segments"; the network model supports "record" and "owned list of records". A given field of a record cannot be a structured data item. The operations are similar in the three models: access or set a field, traverse a relation, tree or list in some order; select a record according to a Boolean condition. In addition the relational model supports operations on entire relations, such as project and join. However the set of operators cannot be extended.

Another problem is that such models typically rely on symbolic identifiers to represent data objects and force users to think in terms of the resulting indirection. In the relational model the properties of an entity must be sufficient to distinguish it from all other entities. Thus, user-defined identifier keys represent the identity of an object. An identifier key is some subset of the attributes of an object which is unique for all objects in the relation. For example to identify department tuples, department names can be used as the distinguishing property. But when a department's name changes, update anomalies occur. Making up unique department numbers to distinguish departments, introduces artifacts into the database scheme that are not in the world being modeled.

Another problem with current systems is that update commands are machine oriented. Update commands insert, delete or modify records. Such updates do not correspond to the real world changes. Changes in the real world require updates to several database items. For example, adding a student into a database may require insertions into several relations. These limitations make integrity checking difficult and therefore may cause inconsistencies in the database.

Another problem is that commercial database systems separate data definition and data manipulation languages. This problem of having two languages is called *impedance mismatch problem* [41]. One mismatch is conceptual, that is the data definition and the data manipulation languages might support widely different programming paradigms. One is a declarative language, the other is a procedural language. The other mismatch is structural. The languages do not support the same data types. For example, a relational database can be accessed using SQL [8] from COBOL , but when some computation is necessary, COBOL can operate only at the tuple level. Thus the relational structure is lost.

Conventional relational systems are not bad at associative retrieval. But they are too slow at fetching and storing fields. Most data processing transactions require getting a few tuples from a relation and updating them or selecting tuples from two or more relations which is performed by taking joins. Each fetch or store costs the same as a procedure call from the application program to the database. This becomes a great overhead when accessing a single tuple. Also normalization and other design considerations increase the levels of indirection between an entity and its subcomponent.

4.3 Object-Oriented Database Management Systems

Object-Oriented Database systems have appeared with a goal to overcome these limitations of the existing database systems. They employ an object data model and use object-oriented programming language facilities.

4.3.1 Properties of Object-Oriented Database Systems

The object-oriented approach can be contrasted to the value-based paradigm supported by the original relational approach to databases. While in relational databases tuples can only be distinguished on the basis of their values, in object-oriented systems there is a hidden permanent unique identifier assigned to each entity (or object). Thus an entity referring to another entity can be implemented using the latter's unique identifier. This policy provides a simple way to support relationships between entities. This built-in identity approach has the advantage that no joins are required for entity relationships. Also it eliminates the disadvantages of the need for unique attribute names.

In the object-oriented approach a single entity is modeled as a single object not as multiple tuples in multiple relations. Properties of entities need not be simple data values but can be other entities of arbitrary complexity. For example the course taken by a student need not be just a string. It can be another object which itself having properties such as course name, credit and the room where it meets, and also its own behaviour.

A data object retains its identity through all changes in its own state. Entities with information in common can be modeled as two objects with a shared subobject containing the common information. For example, two employee objects that work in the same department will point to the department object by using its built in identity. Such sharing reduces the "update anomalies" that exist in the relational database systems and helps to solve the *referential integrity* problem. Referential integrity is automatically satisfied in object-oriented databases. Because one object refers directly to another object, not to a name or a property for that object. The reference cannot be created if the other object does not exist. Any change in an entity value is automatically seen by all entities which refer to it. This is not the case with relational systems. In relational systems, a change in the key value of an entity must be propagated to other tuples sharing that value.

In [25] a correspondence between object-oriented and conventional database systems is established. The three principal concepts, object, message and class, roughly correspond respectively to record, procedure call and record type in conventional systems. Class definitions are the analogue to schemes in database systems. But classes also package operations with the structure to encapsulate behaviour. Other correspondences are shown in figure 4.1.

4.3.2 Problem Areas in Object-Oriented Database Systems

Object-oriented database systems must store both large numbers of objects and objects that are large in size. This necessity requires new storage techniques. Searching a long collection by a sequential scan will give unacceptable performance. Searching for elements should be at most logarithmic in the size of the collection, rather than linear. Storing complex objects on disk presents some difficult problems. In order to illustrate the problems,

object _ oriented	conventional
object	record instance
instance variable	field , attribute
instance variable constraint	field type
message	procedure call
method	procedure body
class-defining object	record type , relation scheme
class hierarchy	database scheme
class instance	record instance, tuple
collection class	set , relation

Figure 4.1: Correspondence between O-O and conventional database systems

consider Employee objects with fields social security number, name, department and salary, where name and department fields are themselves compound objects. Employee objects can be stored basically in two ways [41]. One is to decompose them into their fields and represent each field as a binary relation. That is, binary relations of the form Employee-salary, Employee-department, etc. The other way is to group all the fields of one object together on disk. Binary relation representation is better for associative access, since few blocks need be read for the scan. On the other hand, they are not very good if all fields of an employee object are required to be fetched, since those fields are dispersed through many disk blocks. In the other storage scheme, only one block is read to get all the fields of a single employee object. However, for associative access, performance is not so good. For example, to find employees with salaries over some given value, many disk blocks must be read, because salary fields are separated by all the other fields.

Since both representations have some problems, a hybrid organization is reported in [7]. In this organization binary relations are used on disk to speed up associative access and object-based representation is used in main memory to speed manipulation of single objects.

The capability of processing predicate based queries against a large database is an important requirement in a database environment. Object-oriented systems should support associative access on elements of large collections. They should supply storage structures to support locating an element by its internal state. For example, an application may want to find all employee objects whose salary is more than 100000. Such queries will require searching of all the instances of a class, and cause poor performance. To avoid searching, indexing on instance variables must be supported. The instance variables can be nested several levels deep in an object to be indexed (e. g. the manager variable of the department object which is the value of an Employee's Worksin variable).

Another problem in developing database applications is the impedance mismatch between the programming language used to process data and the data manipulation language used to access the database. This problem is attacked in [7] by proposing an integration of databases and programming languages using objects. Their goal is to create an object-oriented database system with a single language for data manipulation and application programming.

4.4 Some Examples of Object-Oriented Database Management Systems

There are several object-oriented database system prototypes. None of them, except the GemStone Database System, has become commercial yet. The following sections give the summaries of three example object-oriented database systems.

4.4.1 The GemStone Database System

The GemStone database system is a result of a development project at Servio Logic Corporation [24] [25]. It has become commercial recently. It supports a model of objects similar to that of Smalltalk-80. GemStone provides an object-oriented database language called OPAL, which is used for data definition, data manipulation and general computation.

The major pieces of the GemStone system, Stone and Gem, correspond to the object memory and the virtual machine of the standard Smalltalk implementation. Stone provides secondary storage management, concurrency control, authorization, transactions and recovery. Stone also manages workspaces for active sessions. Stone uses unique surrogates called object-oriented pointers (OOPs) to refer to objects, and an object table to map an OOP to a physical location. This indirection means that objects can easily be moved in secondary memory. Object table can potentially have 2^{31} entries. Stone is built upon the underlying VMS file system. The data model that Stone provides is somewhat simpler than the full GemStone model, and only provides operators for structural update and access. An object may be stored separately from the objects it references, but the OOPs for the values of an object's instance variables are grouped together.

Stone supports five basic storage formats for objects, self identifying (e.g. small integer, character, boolean), byte (e.g. string, date, float), named, indexed and nonsequenceable collections. The byte format is used for classes whose instances may be considered atomic. The named format supports access to the components of an object by unique identifiers, instance variable names. The indexed format supports access to the components of an object by number, as in instances of class Array. This format supports insertions of components into the middle of an object and can grow to accommodate more components. The non-sequenceable collection (NSC) format is used for collection classes in which instance variables are anonymous. Members of such collections are not identified by name or by index, but a collection can be queried for membership, and have members added, removed or enumerated. Both the indexed and NSC format support dynamic growth of objects, and are bound in size only by the total number of objects in the system and the physical limits of secondary storage. When objects in these formats grow large, their representation changes from a contiguous one to a B-tree which maintains the members by OOP for NSCs and by offset for indexed objects. The byte format also supports dynamic growth in a manner similar to that for the indexed format. Stone groups objects into logical segments, which are the unit of conflict in concurrency control, and the unit of ownership for authorization.

Gem sits atop Stone, and elaborates Stone's storage model into the full GemStone model. Gem also adds the capabilities of compiling OPAL methods into bytecodes and executing that code, user authentication, and session control. The Gem layer contains the virtual image, that is the collection of OPAL classes, methods and objects that are supplied with every GemStone system. OPAL is a computationally complete language and can express various associative searches on a collection.

Class hierarchy in the current GemStone virtual image is similar to that of Smalltalk's. Comparing it to the Smalltalk hierarchy, classes for file access, communication, screen manipulation and the programming environment have been removed. The file classes are unnecessary as all GemStone objects are persistent. Computation for screen manipulation needs fast bytecode execution. GemStone is optimized toward maintaining large number of persistent objects, rather than fast bytecode execution. The programming environment classes are replaced by a browser application that runs on top of GemStone. Besides removing these classes, new classes and methods have been added to make the data management functions of transaction control, accounting, ownership, authorization, replication, user profiles and index creation controllable from within OPAL.

GemStone's database features can be summarized as follows [34]:

- Multiple concurrent users : The standard mechanism for sharing a database requires the concept of transaction. GemStone uses an optimistic concurrency control policy.

- Sharing of objects : A dictionary is a collection of key-value pairs and supports the naming of objects. GemStone provides each user with a distinct list of dictionaries. Although this list is private to the user, the dictionaries that it contains, can be shared by other users. This allows the sharing of objects in the shared dictionary.

- Security : GemStone secures the object database by first authenticating each user through a user name and password. Also groups of objects may be explicitly marked as either read only, read/write, or no privileges for selected users.

- Centralized server : GemStone is a centralized server for database objects. Currently it does not allow a database to be distributed among several servers.

- Primary and secondary storage management : GemStone hides from application designers the paging of objects between secondary and primary memory, and supports objects larger than the size of the server's primary memory.

- Method execution : GemStone supports a Smalltalk-like execution model.

- Resilience to common failure modes : If the reliability of the disk drives is insufficient then the users can selectively replicate the stored objects on line ensuring that the database survives single-point failures.

- Uniform language : GemStone presents one language, OPAL, to its users. Through OPAL the user manipulates the information in the database, defines new classes, writes portions of application programs, and controls the GemStone server. Thus it is a uniform language that can be used as either a data management or a data definition or a general computation or a system command language.

- Fast associative access : Database systems are traditionally efficient at finding all members of a set meeting a selection criteria. GemStone allows users to dynamically add or remove associative access structures to accelerate such tests.

4.4.2 The ORION Database System

ORION is a prototype object-oriented database system under implementation in the Database Program at Microelectronics and Computer Technology Corporation (MCC) as a research vehicle for developing a database technology for object-oriented applications from the CAD/CAM, AI and OIS domains [4]. It is implemented in Common LISP. The intended applications for ORION impose two types of requirements : advanced functionality and high performance. The ORION architecture has been designed to satisfy these requirements.

ORION provides a number of advanced features that conventional database systems do not, including version control, storage and presentation of unstructured multimedia data and dynamic changes to the database schema.

ORION supports a number of major concepts found in many object-oriented systems such as objects, classes, class lattice, methods and inheritance and also two features to further reduce redundant storage and specification of objects: shared-value and default-value instance variables. For such variables, a value must be specified. For a shared-value variable of a class, all instances of the class take on the specified value. For a default-value variable, those instances of a class whose value for the instance variable is not specified take on the specified default value.

In ORION, as in most object-oriented systems, both classes and instances are viewed as objects. This is necessary mainly for uniformity in the handling of messages. To create an instance of a class, a message is sent to the corresponding class. There are also many other situations in which it is necessary to send messages to class objects, such as inquiry of the definition of the class, changing the definition of the class.

In ORION, a class can have more than one superclass. Thus the class structure is generalized to a lattice. The approach used in ORION to resolve name conflicts among superclasses of a given class is as follows. If an instance variable or a method with the same name appears in more than one superclass, the one chosen by default is that of the first superclass in the list of immediate superclasses for that class. Unlike most other systems, ORION allows the user to explicitly change the permutation of the superclasses. Further, the user may explicitly inherit one instance variable or method from among several conflicting ones.

Each class object belongs to a class, the system defined class Class. All class objects are instances of this class. To create a new class, a message needs to be sent to the class Class. For each user-defined class, ORION defines a corresponding class, a Set-Of class, as a subclass of the class Set. These Set-Of classes form a lattice parallel to the lattice of user-defined classes. One special instance of the Set-Of class of some user-defined class is the set of all instances of that class. Another special instance of the Set-Of class of a user-defined class is the set of all instances of that class and its subclasses. Predicate-based queries are messages to these set objects and return subsets of these sets.

ORION applications require flexibility in dynamically defining and modifying the class lattice. Changes to the class lattice can be categorized as follows: Changes to the contents of a node, changes to an edge and changes to a node. Changing the contents of a node implies adding or dropping instance variables or methods, or changing the properties of them. Changes to an edge imply the alteration of inheritance structure, such as changing the order of superclasses or removing one of the superclasses of a class. Adding or dropping a class, or changing the name of a class are the examples of changes to a node.

One of the enhancement goals of ORION is to support composite objects. A composite object is a complex object formed of a set of subobjects that are treated as units of storage, retrieval and integrity checking. For example, a vehicle is an object that contains a body object, which has a set of door objects, and each door has a position object and a color object. A body object is a part of a vehicle instance, and a set of doors in turn is a part of a body, and so on. Composite objects add to the integrity features of an object-oriented data model through the notion of dependent objects. A dependent object is one whose existence depends on the existence of other objects and that is owned by exactly one object. For example, the body of a vehicle is owned by one specific vehicle and cannot be created without that vehicle. ORION considers a composite object as a unit for clustering related objects on disk, because it is often likely to access all or most dependent objects when the root object is accessed.

In ORION, all instances of the same class are placed in the same storage segment. Thus a class is associated with a single segment and all instances reside in that segment. The user does not have to be aware of the segments. ORION automatically allocates a separate segment for each class. For clustering composite objects, however, it is often advantageous to store instances of multiple classes in the same segment. User assistance is required to determine which classes should share the same segment.

Another enhancement to the object-oriented data model that ORION supports is the handling of versions of objects. ORION distinguishes two types of versions on the basis of the types of operations that may be allowed on them. These are transient versions and working versions. A transient version can be updated and/or deleted by the user who created it. A new transient version may be derived from an existing transient version. The existing transient version can be promoted to a working version. A working version is considered stable and cannot be updated. It can be deleted by its owner. A transient version can be derived from a working version. In ORION version history is represented in a hierarchy. In other words there can be more than one transient version derived from a given working version.

4.4.3 The Iris Database Management System

The Iris database management system [11] is a research prototype of a next generation database management system intended to meet the needs of new and emerging database applications, including office information and knowledge-based systems, engineering test and measurement, and hardware and software design.

Iris database management system consists of a query processor that implements the Iris object-oriented data model, a storage manager that provides access paths and concurrency control, backup and recovery, and a collection of programmatic and interactive interfaces.

The query processor translates Iris queries and operations to an internal relational algebra format which is then interpreted. Instead of inventing a totally new formalism, the system relies on relational algebra. Storage manager is like a relational storage subsystem. It supports the dynamic creation and deletion of relations, concurrency control, logging and recovery, archiving, indexing, and buffer management.

The Iris database management system can be accessed with both interactive and programmatic interfaces. These interfaces are implemented using the library of C subroutines that define the Iris Object Manager interface. Currently two lexically oriented interactive interfaces are supported. One of these is an object-oriented extension to SQL, which is called Object SQL or OSQL [11] in short. The other interactive interface, called the Inspector, is an extension of a LISP structure browser. This interface allows the user to explore interactively the Iris metadata structures, as well as the interobject connection structures defined on a given Iris database.

Iris data model is a typical semantic model developed and partially implemented at Hewlett-Packard laboratories using relational database techniques. The data model is based on the three constructs : objects, types and operations; and supports inheritance and generic properties, constraints, complex or nonnormalized data, user-defined operations, version control, inference and extensible data types.

Iris data model distinguishes *literal objects*, such as character strings and numbers, and *nonliteral objects*, such as persons and departments. Literal objects are directly representable, whereas nonliteral objects are represented internally in the database by surrogates. The Object Manager provides operations for explicitly creating and deleting nonliteral objects, and for assigning values to their properties. Referential integrity is supported in the current prototype by allowing objects to be deleted only if they are not being referenced.

Objects are classified by type. Types are named collections of objects. Types may overlap; for example a person object may be an instance of the types Employee, Taxpayer and Manager. Properties of objects are expressed in terms of functions which are defined over types. They are applicable to the instances of the types. Therefore types are constraints. Types are organized in a type structure that supports generalization and specialization . The Iris type structure is a directed acyclic graph. A given type may have multiple subtypes and multiple supertypes. The subtypes may be overlapping and they do not necessarily partition the supertype. Each object of the subtype must belong to all the supertypes.

Properties may be generic; that is, properties defined on different types may have identical names even though their definitions may differ. The rules for property selection are not yet finalized. The type Object is the supertype of all other types. Types are themselves objects and their relationships to subtypes, supertypes and instances are expressed as functions in the system.

Object Manager allows the type graph to be changed dynamically. In the current implementation a type may be deleted only if it has no subtypes and no instances. Furthermore, new subtype-supertype relationships among existing types cannot be created.

Operations are defined on types and are applicable to the instances of types. The Iris data model and its current prototype support user-defined operations that are stored and executed under the control of the database management system. The specification of an Iris operation consists of two parts, a *declaration* and an *implementation* . A declaration specifies the name of the operation and the number and types of its parameters and results. An implementation specifies how the operation is implemented. The Iris functions are implemented by storing them as a table which maps input values to their corresponding result values.

Information about objects is modeled using *relationships*. Thus, for example the fact that a person has a name is represented as a relationship connecting the person object and the name object. This approach is different from that of the Entity-Relationship model, which allows objects to have attributes. The attribute concept is modeled in Iris by using functions .

The interfaces that have been implemented thus far, include OSQL, Inspector, OSQL embedded in LISP and the Iris database object. The two main extensions that have been made to SQL to adopt it to the object and function model are the following: Direct references to objects are used rather than their keys and also user-defined functions and Iris system functions may appear in WHERE and SELECT clauses. There are also a few keyword differences from existing SQL. The Iris Inspector provides a mechanism for a LISP user to examine Iris database entities in the same manner in which the usual LISP values would be examined. The Iris Inspector provides type-specific handling of the Iris types in much the same manner as the basic Inspector provides special handling for the primitive types from which LISP objects are built.

An object-oriented interface to Iris from Common LISP that presents the model of an Iris database object to the LISP programmer has been implemented. This interface presents to the LISP user a family of types and their methods, which can be manipulated and examined in the same way as any other LISP object types.

The Iris prototype is built on top of a conventional relational storage manager, namely, that of Hewlett-Packard's Allbase relational DBMS [22].

Among the extensions currently being considered are support for long transactions, extensible types, and multimedia objects.

5. THE OBJECT-ORIENTED DBMS PROTOTYPE

The object-oriented database management system prototype designed and implemented at Bilkent University consists of four major modules. These are object memory and schema evolution module; message passing module [30]; secondary storage management module and the user interface module [16]. The user interface is the highest level module. It is built on top of the message passing module which is in turn built on the object memory and schema evolution module. At the lowest level is the secondary storage management module. The overall architecture of the system is shown in figure 5.1.

The developed prototype is a single user memory based system. The persistence of the objects are provided by dumping the memory to disk at the end of each session. The system supports the basic object-oriented concepts, such as the concept of object, object identity, methods, messages, classes, inheritance, message passing and class hierarchy. The class hierarchy is in the form of a tree, thus provides simple inheritance. The system has its own command language which includes both data definition and data manipulation statements. The implementation of the system has been carried out on SUN workstations running under Unix 4.2 and using the C programming language.

In this thesis, the implementation of the object memory module is explained in detail. Also a schema evolution methodology is introduced. The other modules will only be summarized. The object memory module provides

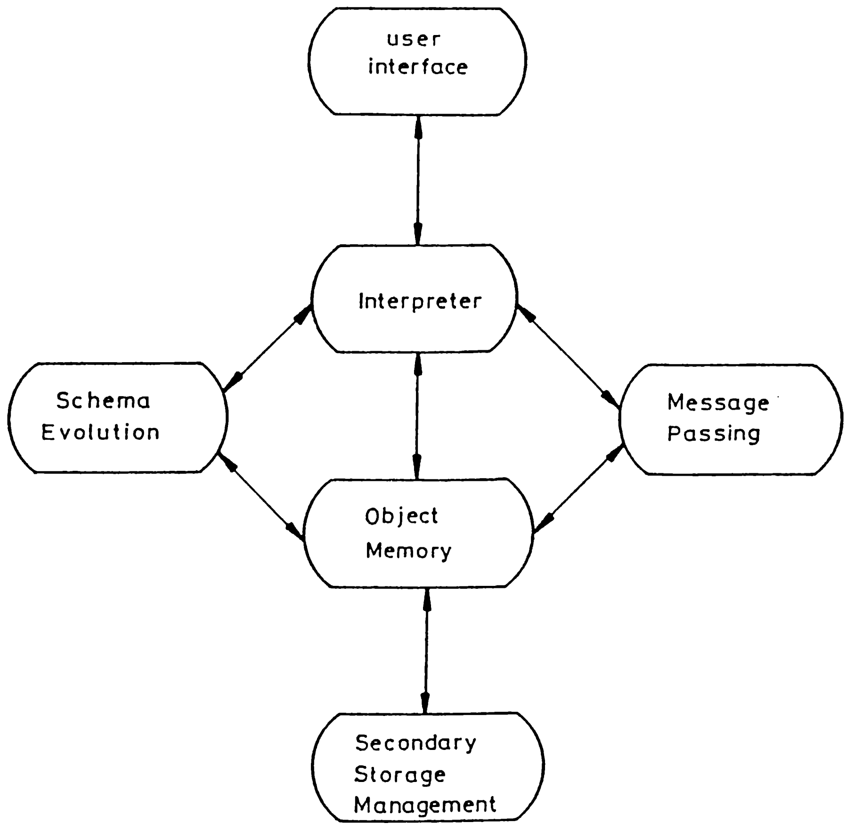

Figure 5.1: The architecture of the prototype system

the internal representations of objects and the functions that are necessary to access and manipulate the objects. The initialization of the system is also performed by the object memory. That is, it initializes all the system defined structures such as the hierarchy of system defined classes and the object table. The module implements the methods that are defined for the system defined classes and provides the primitive operations that can be used in the execution of the user defined methods. The message passing module supports a language which solves the impedance mismatch problem. The language is strongly typed and supports the primitive data types integer and character, collections, sets, arrays and strings. The name of a user defined class is also an allowed type. The module provides the execution of the user defined methods and error handling. The secondary storage management module saves all the existing objects at the end of the session. The aim is to cluster all the parts of an object together on disk. The message passing module and the secondary storage management module use the procedures provided by the object memory to perform their own tasks. The modules communicate with each other through the use of object surrogates.

5.1 The Object Memory Module

Object memory handles the representation, access and manipulation of the objects in the system. Object memory module provides the primitive functions that are necessary in the development of the whole system. In a way these functions are the building blocks of the system. Message passing module and the secondary storage management module use these functions in executing the methods and processing the user commands.

The system supports basically five types of objects:

- Primitive type objects
- Class objects
- Collection objects
- Instance objects
- Method context objects

The primitive type objects are integers and characters. Each type of object has its own internal representation, defined methods and messages. Object memory provides structural access procedures and the implementation of the basic methods necessary to manipulate these objects.

Each object is associated with a unique surrogate. Following the GemStone terminology, these surrogates are called *object-oriented pointers* (oops). Object-oriented pointers are used to identify objects independently of their values. The message passing module and the object memory communicate about objects using oops.

An oop is represented by a 32 bit positive even number allowing approximately 2^{30} objects to be referenced. The unique surrogates are generated by

oop	status	reference count	physical address

Figure 5.2: The format of an object table entry

a permanent counter, which is incremented each time a new object is created. There are two kinds of objects in the system : *persistent objects* and *temporary objects*. Temporary objects are present only during the session and not accessible to the users. They are used in the execution of the user-defined and system-defined methods. The persistent and temporary objects are distinguished by examining their oops. The oops of the temporary objects can take on values between 512 and 1022 and the oops of the persistent objects start at 1024 and can go up to the maximum even number. These limits are not strictly defined. They can be changed to satisfy the system needs. Temporary oop generator is set to its initial value at the beginning of each session. On the other hand the persistent oop generator is set to the value at the end of the previous session. There is also a special oop, NIL, which represents undefined values. It also represents meaningless results.

Object memory uses an *object table* which maps the oops of the objects to their physical locations in the memory. All references to an object are indirected through the object table. Thus, the oops of the objects are in fact indices into the object table. This indirection provides the benefit of moving the objects easily in memory. The format of an object table entry is shown in figure 5.2.

Status field indicates whether the entry is used or not, and also it is used for denoting deleted objects. Reference count field is used to store the number of objects referring to the object for which this entry is allocated. Object table is implemented as a hash table in which oops are used to provide direct access.

The object memory provides the following fundamental functions on the objects :

- Determine an object's size, class, and implementation.

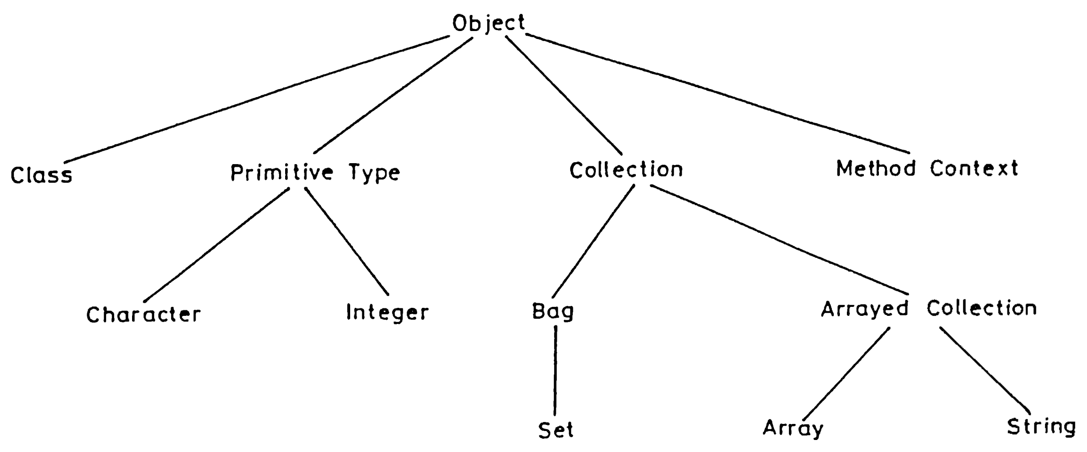

Figure 5.3: The hierarchy of system-defined classes

- Access and change the value of an object's instance variable.
- Access a class object.
- Create a new object.

Classes are themselves objects, therefore they are also associated with oops. There are two types of classes : system-defined classes and user-defined classes. All the classes form a hierarchy, in which each class may have a single superclass. Thus the hierarchy structure is a tree. There are eleven system-defined classes which initialize the class hierarchy. As shown in figure 5.3, the class Object is defined as the root of the tree. It is the only class without a superclass. The class Object defines the common behaviour for all objects in the system. Every class is a direct or indirect subclass of this class. Since classes are objects and all objects are instances of a class, there must be a class of classes. System-defined class Class has all the classes as its instances. It is a subclass of Object. The class Class provides the methods that are applicable to all classes.

Primitive Type class is an abstract class, in the sense that, it has no instances. Currently it has two subclasses which are Integer class and Character class, but it can be extended to include classes for representing boolean

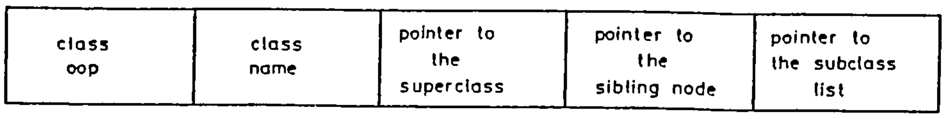
class oop
class name
pointer to the superclass
pointer to the sibling node
pointer to the subclass list

Figure 5.4: The format of a node in the class hierarchy

objects and real number objects.

Another direct subclass of Object class is the Collection class. The instances of collection class are collections of other objects which can be ordered or unordered. The ordered objects are accessed by using integer indices where the unordered objects are just a grouping and cannot be accessed by indices. The two subclasses of Collection represent this difference. Subclass Bag has collections with unordered elements and subclass Arrayed Collection has collections with ordered elements. These two classes in turn have their own subclasses. Bag has a subclass Set, each of whose instances is a collection of objects with no duplicates. Arrayed Collection has the classes Array and String as its subclasses. The instances of String class are the collections of character objects and the instances of Array class are the collections of arbitrary objects. Another direct subclass of the class Object is MethodContext class. Methods are the instances of this class.

In this hierarchy, a class has a sequence of classes starting from its superclass going up to the class Object. This sequence specifies all the inherited instance variables and method names and is called the superclass chain. The class hierarchy is implemented by keeping a subclass list for each class. In addition, each subclass has a pointer to its direct superclass. The format of a node in the class hierarchy is shown in figure 5.4. Thus, given a class its superclass can immediately be found and a specific subclass is found by searching the subclass list. Object memory provides the functions necessary to handle this structure, such as adding a class, deleting a class, searching for a class starting from a given node and going up to the root, etc.

5.1.1 Representations of the Objects Supported by the System

The internal representations of objects supported by the system are different from each other. The classes Class, Integer, Character, Bag, Set, Array, String, and MethodContext have instances with their own data structures. Also the instances of user-defined classes are represented by a different structure which is called an *instance object*. The other remaining classes have no instances separately created and are called abstract classes. Abstract classes define the shared aspects of their subclasses and do not directly contain instances. They are in a way logical grouping of their subclasses. The abstract classes of the prototype system are Object, Collection, Arrayed Collection, and Primitive Type. In the following sections the representation of each type of object will be explained and some examples will be given.

The Representation of Primitive Type Objects

The primitive types (i.e., integers and characters) are *immutable objects*. This means that, once created, they are not destroyed and then recreated when they are needed. The only state of an immutable object is its value which can never change. Therefore, for example, when a new character whose code is between 0 and 255 is requested, a reference is provided to an already existing character. Integers and characters have their value encoded in their oops. This provides efficiency in the retrieval and manipulation of these objects. Integer objects are distinguished by inspecting the least significant bit of an oop. If that bit is 1 then it is understood that it is an integer object and its value is decoded. The integer oops all have a 1 in their least significant bit position and the two's complement representation of their value in the remaining 31 bits.

The character code values range between 0 and 255. Taking this into

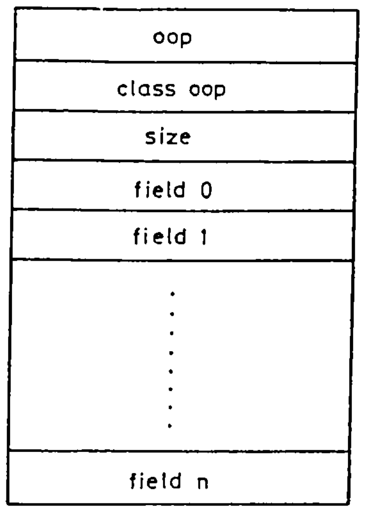

Figure 5.5: The format of an allocated object

consideration, the characters are encoded in their oops as follows: The code value is multiplied by two to obtain an even integer. This is for not to mix them with integer oops. This even number will be greater than or equal to zero and less than or equal to 510. If the value of an oop lies in this range, it shows a character object and its value is obtained by dividing the oop by 2.

The Representation of an Instance Object

The private memory of an instance object is a contiguous series of words, which is called a *chunk*. Each word in the chunk is used to store the value of an instance variable. Since the instance variables are also objects, the stored values are the oops of these objects. The private memories of these instance variables are accessed via their oops. Only the values of Integer and Character objects can be obtained directly by decoding their oops.

The size of the allocated space for a chunk is equal to the number of words necessary to hold the instance variables and header information. The actual data of the object are preceded by the header information which includes the oop of the object, the oop of the class to which the object belongs, and the size of the chunk. The format of an allocated chunk is shown in figure 5.5.

The physical location of the chunk is reached by using the Object Table.

The oop is hashed into the table and the address is retrieved. Fields numbered from 1 to n hold the oops of the instance variables, field 0 is the oop of the super chunk. The function of this field may be better understood if the instance variable inheritance mechanism is explained first.

An instance of a subclass has all the instance variables of its superclass and additionally its own instance variables (local instance variables). Thus, field i ($i = 1...n$) gives the value of the local instance variable i. To represent the inherited instance variables another chunk is allocated as an instance of the superclass. This chunk is referred to as super chunk. It has its own oop, its own size and oops of its local instance variables. Thus field 0 contains the oop of this chunk. The super chunk will have the oop of its super chunk at its field 0, and this continues up to the instance of the class whose superclass is the class Class. At that time, field 0 is set to NIL. An inherited instance variable is accessed by following the path constructed by the field 0. This inheritance mechanism may be best explained by an example.

Suppose there is a class named PersonNames having instance variables first_name and last_name. To create objects that represent person names with titles another class TitledName is created as a subclass of PersonNames. The instances of the TitledName class will automatically have instance variables first_name and last_name and an additional instance variable title to hold the title. Then, another class, TitledNameWithLetters is created as a subclass of TitledName. This new class has the additional instance variable letters (see figure 5.6).

Now, when a new instance of TitledNameWithLetters is created, three chunks will be allocated (assuming that the superclass of PersonNames is the class Class). Figure 5.7 shows the allocated chunks for the name "Dr.John Smith,OBE". In this example it is assumed that all the instance variables are string objects and S1, S2, S3, S4 are the oops of these objects. C1, C2, C3 are the oops of the classes and oop1, oop2, oop3 are the oops of the instances of the classes.

Each class can specialize the initialization of its instances by providing local methods. For the classes that do not provide separate initialization

Class	Instance variable
PersonNames	first - name last - name
TitledName	first - name last - name title
TitledNameWithLetters	first - name last - name title letters

Figure 5.6: Instance variable inheritance example

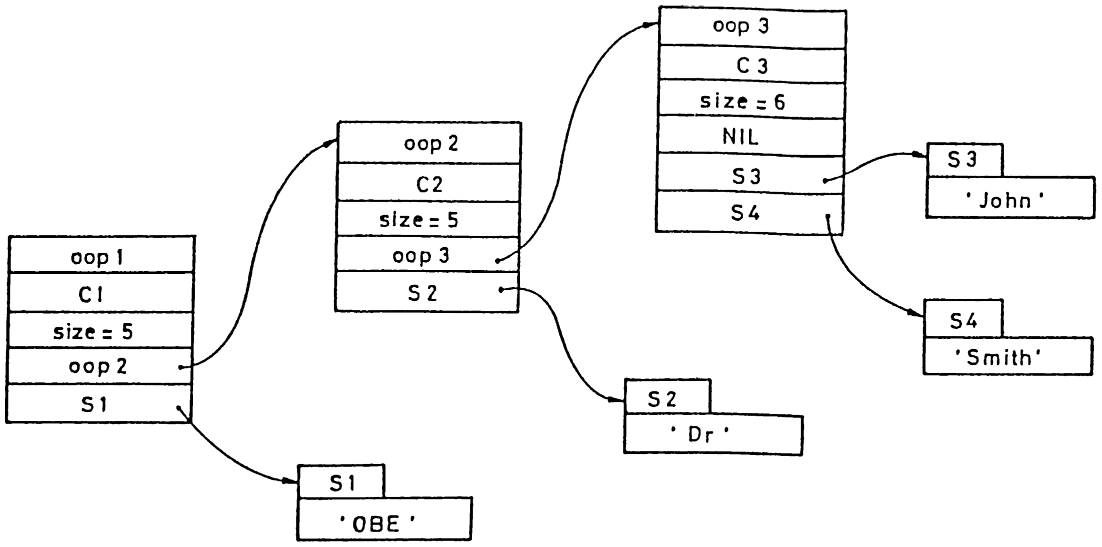

Figure 5.7: Allocated chunks for an object

methods, a default initialization procedure is performed. When a new instance of a class is created, first enough space is allocated to hold the chunks building the instance. That is, if the class has n superclasses up to the class Class, then n chunks will be allocated. Then the fields of these chunks (except fields numbered 0), will be set to NIL. As stated before, the oop NIL represents an undefined object. During this process, the oops and the addresses of the chunks are inserted into the object table.

In some applications, some instances of a class may have some additional behaviour specific to that instance. Note that this is different from the subclass concept. Here, all the instances show the common behaviour, but there can be some exception instances which can perform some specific operations. The model for instance object representation can be extented to specify such instances. The only necessary change will be some additional fields which hold the oops of the methods that perform those specific operations.

The Representation of a Class Defining Object

Every class in the system is represented by a class defining object, or simply class object. The class object contains the information necessary to construct and use its instances. It describes the structure and the behaviour (i.e. methods) of its instances. It also specifies the position of the class in the class hierarchy, thus providing the path to access its superclass chain and its subclasses. Any instance can reference its class's class defining object.

The representation of a class object is different from the representation of an instance object (see figure 5.8). The information in the class object includes the following :

- oop of the class
- name of the class
- number of instances of the class

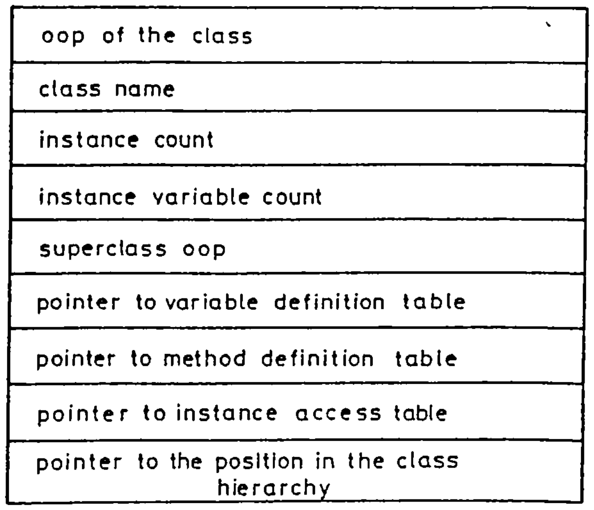
oop of the class
class name
instance count
instance variable count
superclass oop
pointer to variable definition table
pointer to method definition table
pointer to instance access table
pointer to the position in the class hierarchy

Figure 5.8: The format of a class defining object

- number of instance variables
- oop of the superclass
- a pointer to the table which defines the instance variables
- a pointer to the table which defines the methods of the class
- a pointer to the table which provides access to the instances of the class
- a pointer to the position of the class in the class hierarchy

Each class object is associated with an oop and has a name. The name of the class describes the type of its instances. The class name is a simple way for instances to identify themselves. A new class must provide a new class name for itself, because the class names cannot be overridden.

The class object has a field showing the number of instances of the class. This field is used to determine whether the class has any instances or not, and also eliminates the search when the count of instances is required. The number of instance variables is used when allocating space for an instance. The class defining object also specifies the oop of the class object representing its superclass.

The definitions of instance variables of a class are stored in a table called *instance variable definition table*(IVDT). There is a separate IVDT for each

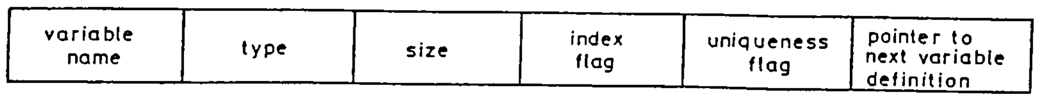

Figure 5.9: An entry of the IVDT of a class

class. The class object provides a pointer to this table in order to interpret the instance variables. The information in this table includes the following :

- name of the instance variable
- type of the instance variable
- size of the instance variable
- index flag
- uniqueness flag
- pointer to the definition of the next instance variable.

The format of an entry in the IVDT is shown in figure 5.9.

The name of the instance variable should be unique among the local instance variable names. If it is the same with an instance variable defined in a class in the superclass chain, then it overrides the definition in that superclass.

The type of the instance variable can be one of the following :

- integer
- character
- derived
- bag
- set
- array

- string
- a class name

If an instance variable has a derived type, then the value of that variable is computed by a method. In the chunk allocated for an instance, the field separated for that variable will contain the oop of this method.

If a class name is specified as the type of an instance variable, then the value for that instance variable will be the oop of an instance of that class.

Size field is for the instance variables whose types are either array or string. The specified size is used to allocate enough space for an object of this type.

The index flag denotes whether there is an index on the instance variable or not and the uniqueness flag denotes whether the values for that instance variable must be unique or not. If the uniqueness flag is set then there cannot be two instances of the class which have the same value for that instance variable.

The instance variable definitions are stored as a linked list. Therefore the last field in the table entry is a pointer to the definition of the next instance variable.

The definitions of the methods which implement the messages to the class are stored in a table called *method definition table* (MDT). Each class has its own MDT. The class defining object provides a pointer to this table in order to access the implementation of the methods. Each entry of the table corresponds to a method defined for the class and contains the following information :

- the method name
- the message name corresponding to the method
- the number of arguments

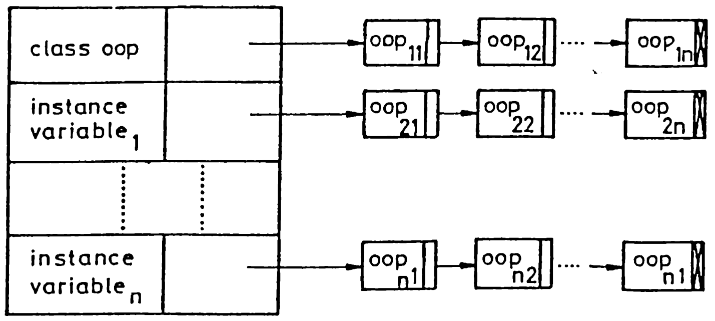

Figure 5.10: The instance access table entry

- a pointer to the list of arguments
- the name of the file that contains the method

When a message is received, the MDT is searched for a match. If there is an entry with the same message name then the corresponding method is invoked. If none is found, then the MDT in the superclass is searched next. The search continues up the superclass chain until a matching message name is found. If again no match is found in any class in the chain, then an error message will be sent to inform the caller.

Each class has an *instance access table* (IAT) which provides access to its instances. IAT keeps a list of the oops of the instances. The access to instances is performed by searching the whole list. The IAT also keeps separate lists of oops for each instance variable. A list for an instance variable contains the oops of all objects that are stored in the field separated for that variable. In a way this list is used as a domain of values for that instance variable. Keeping a list for each instance variable provides an easy way of handling queries that contain existence tests. The format of an IAT is shown in figure 5.10.

The language designed for the prototype system supports some additional information included in the class object. These are :

- number of search keys
- list of search keys
- number of shared variables

- list of shared variables

The variables specified as search keys should already be defined in the class object. The values of the shared variables can be accessed by all the instances of the class.

This completes the description of the class defining object. Each class object is an instance of the system-defined class Class. To create a new class, a message is sent to the class Class. A new class is always created as a subclass of an existing class. If the user creates a new class without specifying the superclass then its superclass will be the class Class by default. The class Class, is the only class that has instances which are also kept in the class hierarchy. Instances of Class have their own instances which can have one of the existing object structures.

The Representation of an Array Object

An Array object is a collection of arbitrary objects that can be accessed by integer indices. The indices range between 0 and the size of the array. When a new array object is created its size must be specified. The objects in the array object may also be an array, thus providing multidimensional arrays. The structure of an Array object is shown in figure 5.11.

The Representation of a String Object

A String object is a collection of character objects which can be accessed by indices. Like Array objects, the size of the String object must be specified. Since it is known that the elements of the String object are characters, they are not stored as encoded oops, but as characters. This saves space and simplifies manipulation of string objects. The format of a string object is shown in figure 5.12.

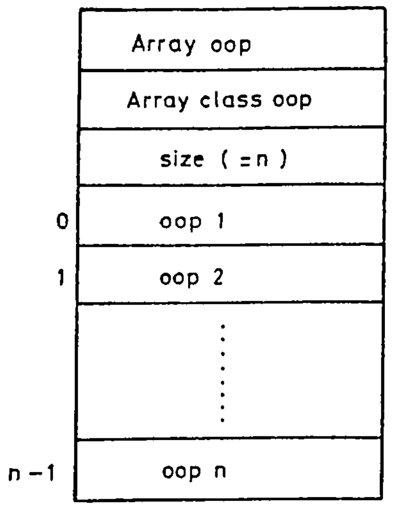

Figure 5.11: An Array Object

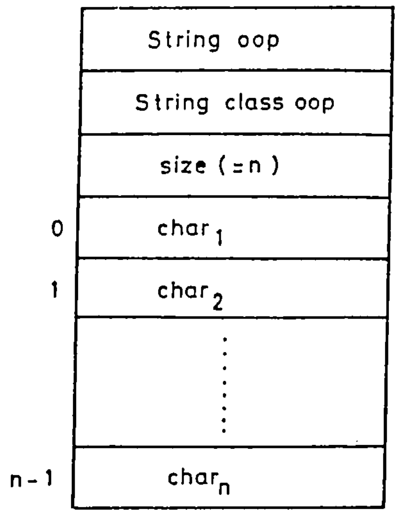

Figure 5.12: A String Object

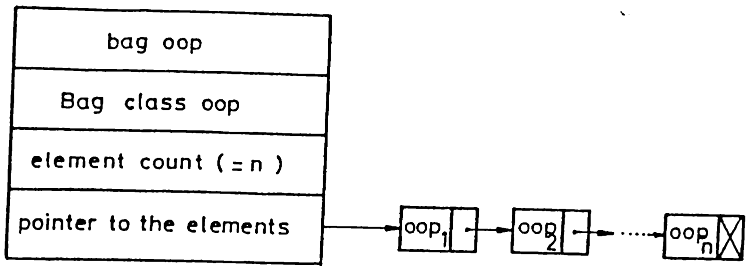

Figure 5.13: A Bag (Set) instance

The Representation of a Method Context Object

A method context object is formed of a header and a body. The header contains the method name, the corresponding message name, the name of the class to which the method belongs, and a list of optional or mandatory arguments of any system defined type or of any class. The method body is formed of a group of batch mode or interactive mode statements. The method and the message name may be the same. All methods are persistent and the code for a method and its compiled form are kept in separate data files.

Representation of BAG and SET Objects

Bag class is a subclass of the abstract class Collection. Its instances are collections of the oops of arbitrary objects. The elements of the instances can also be bag objects. That is nested bag objects are supported. The structure of a bag object is shown in figure 5.13.

The internal structure of a set object is the same as the bag object. The bag and the set objects provide an easy way to represent the facts that are represented by nested relations in the relational model.

For example, suppose there is a Parent class which has instance variables name, age, and children. The variable name is a string object, age is an integer object and children is a collection object. When a new Parent object is created a chunk will be allocated, and in this chunk the field separated for the instance variable children will store the oop of a bag object. The elements of this bag object may be Parent objects as well, or some other object belonging to another class, such as Student. Suppose Student class has instance variables sname, and school which are again string objects. The allocated chunks to represent the Parent "George" is shown in figure 5.14. "George" is 60 years old. He has two children, "Mary" and "Tom"."Tom" is a student at Bilkent. "Mary" is 35 and has a child "John" who is a student at METU.

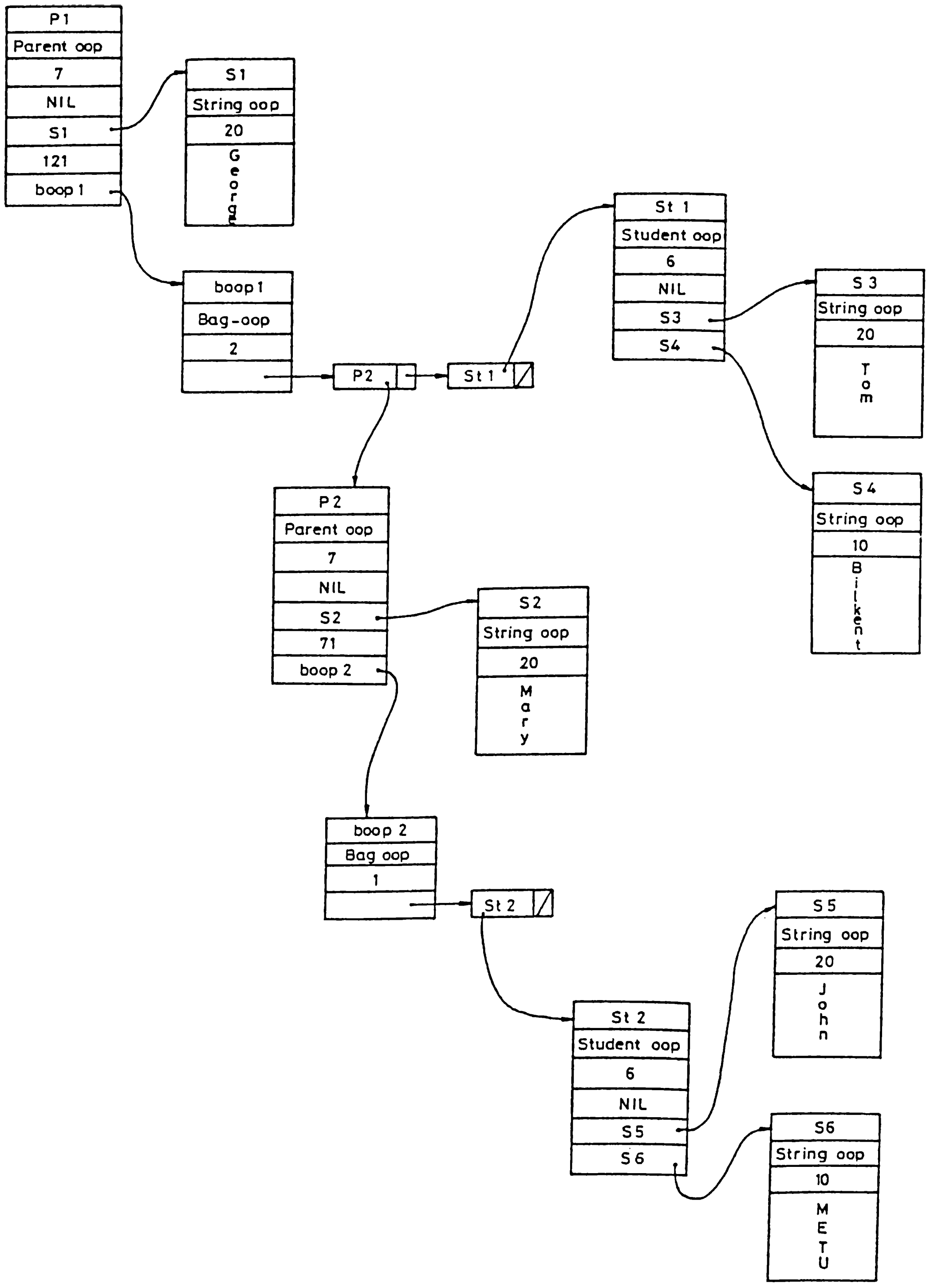

Figure 5.14: Representation of a Parent Object

(George,60,((Mary,35,(John,METU)),(Tom, Bilkent)))

5.1.2 Protocols For All System-Defined Classes

Object memory provides the following :

- space allocation for the objects in the system
- procedures to initialize the system-defined data structures such as the object table and class hierarchy
- procedures for handling the instance variable definition table, method definition table, instance access table
- methods for the system-defined classes
- class modification methodology

A protocol is a list of messages understood by instances of a particular class. The methods that are invoked by these messages are either found in the local method definition table of the class or searched through the superclass chain. Object memory provides the implementation of the methods of the system-defined classes. In the following section, the protocol and the methods for each system-defined class will be given. The protocol for a user-defined class is explicitly specified in its class definition. User-defined classes also inherit the methods defined in the system classes.

Protocol for the Class OBJECT

Everything in the system is an object. The protocol common to all objects in the system is provided in the description of the class Object. In other words any object created in the system can respond to the messages defined by the class Object. These are typically messages that support default behaviour and provide a starting place from which to develop new kinds of objects, either by adding new messages or by modifying the responses to existing messages.

The following method categories constitute the protocol for all objects.

a. Methods for testing functionality :

An object's functionality is determined by its class. To test the functionality of an object, the class Object includes the following methods.

class_of(oop) : returns the oop of the class to which the object pointed by oop belongs.

is< *classname* >**(object)** : returns true if the object oop is an instance of the class denoted by the *classname*.Otherwise returns false.

exists(oop) : returns true if there is an object with the surrogate oop.

b. Methods providing access to parts of objects :

size_of(oop) : returns the size of the chunk specified by the oop.

total_size(oop) : returns the total size of all chunks that make up the object specified by oop.

c. Methods for comparing and copying objects :

There is a protocol for testing the identity of an object and for copying objects. The important comparisons specified in class Object are equivalence and equality testing. Equivalence testing answers whether the two objects are the same object or not, (i.e. they have the same oop or not). Equality test is for determining whether the values of two objects are the same or not. For example, equality of two arrays is checked first by looking at their size and then comparing each of their elements one by one. Since each new class may add new instance variables, the equality testing must be reimplemented in that class. The default implementation of equality test is the same as that of equivalence test. The implementation of equality and equivalence testing have not been completed yet.

In the discussion of object identity it was stated that there were two ways to make copies of an object. These were shallow copy and deep copy. The designed Object Memory only supports deep copying. That is, the copy is another object with its own identity and with values equal to those of the original object. The implementation of deep copying also has not been completed yet.

d. Methods for printing objects :

Since the objects that are supported by the system have different internal representations, they all have separate printing methods. These methods are included into their class definitions. In addition to these methods, the user may specify other methods for printing the instances of newly created classes. The methods supported by the system are the following :

printinteger(oop) : prints the value of the integer object.

printchar(oop) : prints the character represented by the oop.

printstring(oop) : prints the contents of the string pointed by oop.

printarray(oop) : prints the contents of the array pointed by oop.

printBag(oop) : prints the elements of the collection (bag or set).

printClassObject(oop) : print the contents of the class describing object.

Protocol for the Class INTEGER

The class Integer is a subclass of the class Primitive Type. As it is stated before, the values of integer objects are encoded in their oops. For example, the integer value encoded in the oop 00..01100011, is found by shifting the oop one bit to the right. Thus the value 49 is obtained as the value of the integer object. In class Integer, methods for encoding and decoding of an integer object and printing and reading an integer object are provided. The addition, subtraction, multiplication and division operations are automatically supported by the system.

Protocol for the Class CHARACTER

The second subclass of the class Primitive Type is the class Character. Each character occupies four bytes since its code is encoded in its oop. This is of course a waste, but it provides homogeneity in the representation of instance objects.

Methods for instances of this class support accessing the ASCII value and the digit value (i.e. the oop of the character) of the instances and testing the types of a character. Type testing methods are the following :

isDigit(char) : returns true if the character is a digit.

isAlpha(char) : returns true if the character is a letter.

isAlphaNumeric(char) : returns true if the character is a letter or a digit.

Protocol for the Class COLLECTION and Its Subclasses

A collection represents a group of objects. The objects are called the elements of the collection. For example, a String object is a collection whose elements are characters. Also a set object is a collection. Its elements are arbitrary objects. They can be sets as well. The class Collection is the superclass of all collection classes. It is an abstract class. That is, it provides the methods common to all its subclasses, but has no instances.

Collections can be with ordered or unordered elements. The subclasses Bag and Set are collections with unordered elements. The difference between these two classes is that Bags allow duplicate elements, but Sets do not allow duplication. Arrays and Strings have ordered elements. The order is specified externally when the elements are added.

Collections support the following operations :

- Addition of a single element or several elements.
- Deletion of a single element or several elements.
- Testing whether a collection is empty or not.
- Testing whether a particular element is included or not.
- Determine the number of times a particular element occurs in the collection.
- Update a particular element in the collection.

A single element or several elements can be added or removed from a collection. It is also possible to test whether a collection is empty or not. Testing whether it includes a particular element is performed by equivalence testing. But, equality testing will also be provided in the later development stages of the prototype. It is also possible to determine the number of times a particular element occurs in the collection again by equivalence testing.

The methods for adding, removing and testing elements are as follows :

addtobag(aBag,oop) : adds the object pointed by oop to the collection aBag.

addall(aBag,bag_oop) : adds all the elements of the collection pointed by bag_oop to the collection aBag.

rem_element(aBag,oop) : removes the object pointed by oop from the collection aBag. If it is not found in the elements of the collection, then an error is reported.

rem_aCollection(aBag,bag_oop) : removes each element of the collection pointed by bag_oop from the collection aBag.

isEmptyBag(bag_oop) : returns true if the collection has no elements otherwise returns false.

includes(aBag,oop) : returns true if the object pointed by oop is one of the elements of aBag.

count_of(aBag,oop) : returns the number of elements that are equal to the object pointed by oop.

In addition all collections can return the number of their elements.

Protocol for the Class ARRAYED COLLECTION and Its Subclasses

Arrayed Collection is a subclass of Collection. It represents a collection of elements with a fixed range of integers as external keys. Arrayed Collection has two subclasses : Array and String.

The class Arrayed Collection provides the methods for creating, accessing, copying and enumerating elements of a collection. It is possible to determine the first and last elements of the collection.

The class Arrayed Collection is an abstract class. Since the internal data structures of its subclasses are different, it does not provide sufficient representation for storing elements, so it is not possible to provide all the implementations in Arrayed Collection class. Because of this, no instances of this class are created.

Array is a subclass of the class Arrayed Collection. It is concrete (i.e. not abstract) in the sense that it provides a representation for storing elements and implementations of the messages not implementable in its superclass. The methods provided in the class Array are as follows :

value_at (oop,index) : returns the value of the element whose location is denoted by index in the array pointed by oop. If index is greater than the size of the array an error is reported.

change_value_at (oop,index,newvalue) : change the value of the element whose location is denoted by index by the newvalue. The index should be less than or equal to the size of the array, otherwise an error is reported.

String is a subclass of the class Arrayed Collection. It adds methods for creating a copy of another String object, concatenating two string objects, comparing two string objects. The methods provided by the class String are as follows :

length(oop) : returns the length of the string pointed by oop.

char_at(oop,index) : returns the character at the location index in the string pointed by oop. Index should be less than the size of the string, otherwise an error is reported.

change_char_at(oop,index,newchar) : changes the value of the element at location index by the new character value in the string pointed by oop.

stringcopy(oop1,oop2) : copies the second string pointed by oop2 into the first string pointed by oop1.

stringcompare(oop1,oop2) : compares the contents of the two strings pointed by oop1 and oop2, and returns true if they are the same, otherwise returns false.

stringcat(oop1,oop2) : if the size of the first string pointed by oop1 is enough to hold the second string pointed by oop2 then the second string is appended to the end of the first string. Otherwise an error is reported.

Protocol for the class CLASS

The class Class provides the facilities needed to describe new classes. It is a subclass of class Object. Creating a new class involves specifying names for instance variables, methods, and the class itself and also compiling methods. A new class is specified by creating a subclass of another class. Every object is an instance of a class and every class is an instance of the class Class.

The methods of Class support the behaviour common to all class objects. The functionality of all classes are determined by the following four categories of messages :

- messages for creating new classes.
- messages for accessing the existing classes.
- messages for testing the definition of classes.
- messages for enumerating subclasses and instances.

a. Creating new classes :

Messages for creating new classes invoke methods that create a method definition table, an instance variable definition table, an instance access table and a link to the class hierarchy.

The methods in class description are stored in the method definition table. The keys in this table are message names. Using this table, the compiled form of methods can be accessed. The protocol for creating the method definition table supports compiling methods and also supports accessing both the compiled and noncompiled (source) versions of the method.

The class Class also supports the creation of new instances by the message new. The default implementation is to allocate enough space for the object and initialize all instance variables to NIL. However the message new can be overridden in the method table of a class in order to supply special initialization behaviour. The purpose of any special initialization is to guarantee that an instance is created with variables that are themselves appropriate instances.

The protocol for creating classes includes methods for placing the class within the hierarchy of classes in the system. Since this hierarchy is in the form of a tree and only addition to the leaves of this tree is allowed, placing the class in the hierarchy only needs setting the superclass and adding the new class to the subclass list of the superclass.

b. Accessing the classes :

Messages for accessing the existing classes invoke the methods that access the contents of the method definition and instance variable definition tables, the instances of the classes, and the class hierarchy.

The messages that access the contents of a method definition table, distinguishes the selectors specified in the class's local method definition table and the selectors that are inherited from the superclass chain.

Similarly, the instance variables that are locally defined and those that are inherited can also be distinguished by the supported methods. Methods that provide access to instances and variables can be listed as follows :

allInstances(oop) : returns a set of all (direct) instances of the class pointed by the oop.

Instance_count(oop) : returns the number of instances of the class pointed by oop.

LocalInstVarNames(oop) : returns all the instance variables that are defined in the instance variable definition table of the class pointed by oop.

InheritedInstVarNames(oop) : returns the set of all the instance variables that are inherited from the superclass chain of the class pointed by oop.

allInstVarNames(oop) : returns all the instance variables that are inherited from the superclass chain and that are locally defined in the class pointed by oop.

The protocol for accessing the class hierarchy includes messages for obtaining the set of superclasses and subclasses of a class. The methods to execute these messages are as follows :

subclasses(oop) : returns the set of the direct subclasses of the class oop.

allSubclasses(oop) : returns the set of all the subclasses of the class oop.

super_of(oop) : returns the immediate superclass of the class oop.

allSuperclasses(oop) : returns all superclasses of the class oop, including class Object.

c. Testing the definition of a class:

Testing protocol provides the messages needed to find out information about the structure of a class and the form of its instances. It includes messages that test how its variables are stored, the number of the instance variables, the types of the instance variables, and the ability to respond to particular messages.

The methods provided in the class Class to test the definition of a class are the following :

includesSelector(oop,selector) : returns true if the selector exists in the local method definition table of class oop, otherwise returns false.

canUnderstand(oop,selector) : returns true if the instances of the class oop can respond to the message selector.

name_of(oop) : returns the name of the class oop.

coop_of(classname) : returns the oop of the class whose name is given.

type_of_field(coop,var) : returns the type (class) of the instance variable var defined in class coop.

existsvarname(oop,name) : returns true if the class oop defines the instance variable name in its instance variable definition table.

d. Enumerating instances and subclasses :

The messages in this category support access to each of the instances and subclasses of a class. The following are the methods provided by the enumerating protocol.

isInstance_of(coop,oop) : returns true if the oop is a instance of the class coop.

first_instance_of(oop) : returns the oop of the first instance of the class oop.

next_instance_of(oop) : returns the oop of the next instance of the class oop.

subclass_of(oop) : returns the first subclass in the subclass list of the class oop.

next_subclass_of(oop) : returns the next subclass in the list of subclasses of class oop.

5.1.3 Schema Evolution Methodology

One of the important requirements of database applications is the schema evolution, that is the ability to change the database schema dynamically. Existing conventional database systems support only a few types of changes to the schema. For example SQL/DS only allows the dynamic creation and deletion of relations and the addition of new columns in a relation [8].

In object-oriented databases there can be changes to the class definitions or to the structure of the class hierarchy. The types of changes required include

- creation and deletion of a class
- alteration of the IS-A relationship between classes
- addition and deletion of instance variables and methods

The most important point in designing a class modification methodology is how to bring existing objects in line with a modified class. There are two approaches to overcome this problem : *screening* and *conversion* [33].

Screening approach defers modifying the objects. The representation of objects are corrected as they are used. But this will cause a long degragation of performance. The other approach, conversion, changes all instances of the class to the new class definition. In this approach much time can be consumed at the time a class is modified. It is also important to ensure that the class's methods agree with the new definition.

In both approaches, the aim is to maintain a consistent database. To achieve this some rules must be satisfied at the end of each modification. First of all, classes and subclasses must form a hierarchy, that is there cannot be disconnected components. Also, every class must inherit every instance variable defined in its superclass. The representation of an inherited variable must be the same with that of the superclass. Another rule is that there cannot be dangling references. There must not be references to non-existing objects.

Possible class modifications can be listed as follows :

- Renaming an instance variable name : The name of an existing instance variable may be changed; but renaming is not allowed if the new name is already defined in the class and if the instance variable is inherited from the superclass. The renaming is propagated to all subclasses. Further, if the renaming fails in any subclass then the operation is not allowed.
- Adding an instance variable : All instances of a class may have an additional instance variable defined. Adding an instance variable is not allowed if another instance variable already has the name. If in any subclass of the modified class the instance variable is already present the operation is disallowed. If the instance variable is not defined in a subclass, then the modification is propagated to that subclass. All instances of the class to which the instance variable has been added gain a value of NIL.
- Removing an instance variable : All instances of a class may have an existing instance variable removed. An instance variable may not be removed if it is inherited from a superclass. All instances of the modified

class are re-written to remove the instance variable. The modification is propagated to subclasses.

- Removing a class : A class may not be removed if it has any instances. First the instances of the class must be removed. The problem is that, there can be instances which refer to the instances of the removed class (dangling reference problem).
- Adding a class : To add a leaf node requires the name of the superclass to be specified. To add an interior node to the hierarchy, the new class's name, its superclass, and its subclasses are specified. The subclasses must be immediate subclasses of the given superclass.

In the prototype, the conversion approach has been selected. That is, every time a schema change occurs, the object structures and methods are changed accordingly. The design of the schema evolution has not been completed yet. Currently, the allowed changes are the following.

A new class can be added to the hierarchy as a leaf node. The name of the new class must be different from all the existing classes. To add it into the hierarchy its superclass's name must be given. The new class is added to the end of the subclass list of the given superclass.

Deletion of a class is allowed if again it is a leaf node. If it has instances then they must be deleted first. The deletion of the instances is performed by removing their oops from the instance access table of the class and changing the status field of the entry in the object table to the deleted state. Thus references to that object will see that the object is deleted, and they will decrement the reference count of the object and change their reference to NIL.

Addition of an instance variable is allowed when the conditions specified above are satisfied. When a new instance variable is added to a class definition, all the instances of that class is restructured. That is, a new space is allocated to each object, the old values are all copied to the new space and the new field is set to NIL. The setting of the new field to an appropriate value must be done explicitly by the new methods.

Deletion of an instance variable also requires restructuring of objects. One problem is with the methods. The methods using those instance variables must be either removed or changed accordingly.

5.2 Message Passing Module

The message passing module is built on top of the object memory and schema evolution module [31] [30]. It includes the definition and support of the designed command language and error handling in addition to message passing. It consists of five submodules which are the lexical analyzer, parser, code generator, executor module and the query processor.

The command language of the object-oriented database management system prototype is designed to provide unification so it captures both the data definition and data manipulation language aspects. The language can be used both interactively, that is, command by command or in the batch mode, that is, in the form of methods.

The commands can be classified into two major groups: interactive mode statements and batch mode statements. The interactive mode statements can be further classified as follows:

1. Definition statements: One can define a new class, a new method for a class or a new instance of a class.
2. Schema evolution statements:
   - Additions to a class definition
   - Deletions from a class definition
   - Modifications to a class definition
   - Renaming operations
   - Additional changes to a class definition
   - Changes in the class hierarchy

3. Query statements: These are for accessing and manipulating objects. They include statements for retrieving instances and class information, index manipulation, object duplication, equality checks and method manipulation.

The batch mode statements may only be used in methods and provide iteration, conditional execution, declarations, assignments and message calls. There are two types of message calls. These are the system calls which are implemented as C function calls and actual message calls which are executed by the executor module and which have the following format:

$< \text{destination} >< \text{message name} > [<\text{argument list} >]$

The argument list field is optional. System defined data types are integers, characters, arrays, strings, sets and collections. In addition to these, a variable may be declared to be an instance of a class by specifying the class name.

A method is created using a method definition statement. Methods are accessed through the method definition table.

The lexical analyzer, parser and code generator form the compiler for the command language. Every time a new method is created or a method is modified and a compile method statement is executed or each time a message is invoked and the compiled form of the corresponding method is not available, these subroutines are invoked. At the end of the code generation phase, the interactive statement or the method is converted into a set of integer codes and stored in a file. The executor module takes the generated integer codes as input and performs the corresponding operations using a structure called an *activation record.* During the execution phase, the interactive statements are considered as methods with no arguments for the class Object.

Each message returns a fixed size and fixed structure block. This block contains an error flag, a flag indicating whether a value is returned or not, returned value type, the address of the memory location containing the returned value and for indexed return values the maximum length and the element type.

An activation record contains the following information:

- the class name of the method (this is needed for message calls with self or super as destination classes)
- a pointer to the return block
- the name of the file containing the method
- the program counter
- the condition register
- the accumulator
- symbol table pointer- The symbol table contains the name, type, maximum length, element type, usage flag and address of temporary variables.
- reference table pointer- This table holds the message names, class and instance variable names used in the method.
- argument count
- a pointer to the list of arguments- Each node of the list contains the address of the argument and an index identifying the argument.

Each occurrence of a literal in a method is converted into an index for the reference or symbol table. Each activation record has its own program counter, accumulator, condition register, symbol table and reference table. There is a global expression evaluation stack used by all methods.

Activation records are created whenever a message call is executed. The previous activation record is pushed on to the *activation stack*. Whenever a return from a message invocation is performed, an entry is popped from the stack and it becomes the current activation record. This solves the parameter passing and the return address handling problems.

The query processor handles various associative retrieval queries using the routines provided by the object memory and the indexing modules.

Error handling is performed at all stages. Each time an error occurs, an error code is generated and the corresponding error message is retrieved from the system error file and displayed or written to a file.

5.3 Secondary Storage Management Module

Efficient storage and retrieval of objects in secondary storage constitutes an integral and important part of the prototype implementation.

The memory system is composed of system defined and user defined persistent objects and temporary objects. The secondary storage module is responsible for managing the transfer of objects between main memory and disk while making sure that the object identity remains unchanged throughout its internal (secondary storage) and external (main memory) representation [31]. The issue of being able to propose a uniform method for handling objects of very different sizes is also very important.

Major access problems are incurred due to the nonnormalized nature of objects and objects being variable-sized. The storage structure and the addressing mechanism should provide fast access to entire complex objects and to their components at the same time. This demands efficient ways of clustering the objects and thus eliminating frequent diskhead motions and single object transfers.

The objects of the main memory data model are mapped to disk objects (called containers) each of which can be viewed as a segment with proper definitions of its class instance variables and super objects. The main objective is to hold together individual chunks of an object contiguously on disk which also happens to be the clustering preference of the prototype. It is assumed that retrieving a chunk into main memory would most likely reference to other chunks of the same object due to inheritance and thus retrieving a complex object in its entirety is important for eliminating single chunk disk retrievals. Another partitioning approach such as storing all instances of a class together for clustering would satisfy queries requiring the search of all objects of a class. It is up to the application to determine which access pattern would be more suited. However clustering could be achieved by only one preference and the system's default is storing the objects with its super objects. Another clustering scheme is implemented so as to cluster together collection objects that are values of nested-type instance variables of an object. Objects are stored in disk as a byte stream using Unix low-level file services .

The secondary storage module is flexible to be able to do certain conceptual level to physical level transformations for efficiency and performance, yet for this reason the container objects know information about the form of objects that are contained in them.

A container is recursively defined as a variable sized segment in disk which contains an object's instance variables' values and either the container of its super object's instance or a reference to that container. Resizing a container is possible in two ways; by reorganization or by using an overflow file to keep overflown instance variables. Accessing subcontainers in a container is possible via an oop-to-container-address conversion. However once one is in the root container one can make use of physical contiguity and skip the address mapping. If a super object can not be contained in a container due to multiple referencing or identity assignments, then a slot containing the oop of that object is used in resolving the reference . The storage manager guarantees that instance values of an object can be found in exactly one container and other references to that object will be redirected to point to this container.

In order to provide alternate access paths to objects, based on the values of their instance variables (i.e. to provide associative access to objects) an indexing module is implemented.

Indexing is performed on classes and automatic index maintenance is provided by the system. An index is specified by a path name which is a string of the form $A_1 ... A_n$ where $A_i \in$ user defined classes and A_i is a subclass of

A_{i+1} for $i = 1..n-1$ and there does not exist any i such that A_i = CLASS class and the indexed instance variable is among the instance variables of A_n. Indexing a path $A_1...A_n$ on the instance variable V will associate the oops of the objects found at class A_1 with the value of V in the corresponding super object.

Multi-level indexing is performed by indexing each link along the variable path rather than maintaining a single index for the whole path. This allows the query processor to take advantage of more efficient access patterns even if indexes are not specified.

5.4 The User Interface Module

The User Interface of the designed prototype is also object-oriented and the user is navigated by a pop-up menu driven system to the operations he/she desires to perform. The User Interface provides three different environments corresponding to three groups of users: (i) developer/maintainer , (ii) domain specialist, (iii) end-user .

The first environment contains the tools for doing schema changes such as defining new classes, instance variables, updating existing ones, editing methods and customized applications in the prototype's command language.

The second environment contains tools for creating, updating new instances of classes , invoking methods of objects, and doing operational maintenance.

The third environment is for running only customized applications and thus interacting with the database in a controlled manner.

5.5 Current State of The Implementation

Currently the implementation of the modules have not been completed yet. After their completion, the modules will be combined together to build the whole system. As far as the object memory module is concerned, the current state is explained in this thesis. All the listed methods in section 5.1.2 are implemented. Some of the remaining important methods relate to equality checks. These methods will be used in the processing of the associative queries. Also, in some applications it may be necessary to make an instance of a class an instance of another class. For example it may be required to make an existing student an employee. The object memory module must support functions to perform such operations.

Also, reference counting and garbage collection are not performed in the system. Whenever an object is deleted the status field of the object table entry corresponding to the object is set to the deleted state.

Currently the source code lines of the major components of the prototype are of the following sizes:

- Object Memory Module .... 2700 lines
- Message Passing Module .... 4000 lines
- Secondary Storage Management Module ... 1000 lines

Future work will be in several directions. First the prototype needs to be extended to handle very large number of objects. Currently all the objects stay in the main memory. The object memory module and the secondary storage management module must provide functions to call the objects into memory as they are needed. Secondly, as it is stated before, object memory should provide schema evolution. Currently the allowed operations are very restricted. The implementation of the module will be complete when all the schema evolution functions are added to the system.

6. THE RESEARCH ISSUES OF OBJECT-ORIENTED DATABASE SYSTEMS

The object-oriented approach has its advantages and problem areas but especially for data-intensive applications, it is a very promising and active research area.

Typical VLSI/CAD applications are often concerned with object versions, that is multiple representations of the same semantic entity to account for different stages, different times of validity. Object-oriented database systems therefore need mechanisms to deal with versions.

A temporal extension to a data model provides historical access for users. Historical access is usually not provided in database systems. Temporal extensions of data models are an active research topic, but the results of that research have not reached commercial systems yet. The object-oriented data model has the potential to capture the history of database states as a part of the data model.

The object-oriented data model is sufficient to represent a collection of related objects. However it does not capture the IS-PART-OF relationship between objects. One object simply references but does not own other objects. A composite object is an object with a hierarchy of exclusive component objects. Many applications require the ability to define and manipulate composite objects. The object-oriented data model can be enhanced to model composite objects.

When manipulating objects comprising large bulks of data, transactions may become much longer than usual. New concepts are therefore needed to accommodate long-duration transactions, and in addition the concepts for recovery and consistency control and their relationship to the transaction concept have to be reconsidered.

Protection mechanisms have to be based on the notion of object which is the natural unit of access control in this framework.

For databases containing large numbers of data, archiving may become a major issue. Again, objects and their versions, if any, form the natural unit for this activity.

Other research areas for object-oriented database management systems are indexing and schema evolution. The indexing problem is introduced by the use of a location and value independent surrogate to reference an object. The problem arises during the value based access of objects. Schema evolution has many problems to be solved. There have been several research groups working on this area [33] [3].

7. CONCLUSIONS

The object-oriented database management system prototype developed and implemented at Bilkent University supports the basic object-oriented concepts such as object identity, classes, inheritance and message passing. This prototype system is only a starting point to develop a more functional object-oriented database management system. The system will be enhanced to support more advanced facilities.

The implemented prototype is a single-user system. It may be extended to support multiple users. This requires the addition of the transaction concept, authorization control, concurrency control and data integrity checks.

The system allows basic schema evolution functions such as adding a new class to the system, adding a new instance to a class, deleting an existing class and deleting an instance of a class. The system may be extended to support all schema evolution functions.

The class hierarchy may be extended to support multiple inheritance. To achieve this, each class will point to a superclass list instead of pointing to a single superclass. Of course, necessary checks should be made in order to have a directed acyclic graph of classes. Also the representation of the instance objects needs to be modified to point multiple super chunks.

One problem of the system relating to performance is that, the instances of a class are accessed by searching the list of oops in the instance access table. The organization structure of this table may be changed to provide a faster access.

The survey that has been done in this thesis showed that object-oriented approach is very suitable for database applications which need representations of complex data. The approach introduces powerful concepts for modeling the application semantics, thus allowing to keep the semantic gap between the data and the representation of data as small as possible. Object-oriented database systems offer a different kind of modeling which causes reconsidering a number of database issue, such as version management, recovery and consistency control, security and archiving.

However it appears that it is more difficult to design a high quality object-oriented database management system. Appropriate design methodologies and tools that support them have to be developed. An object-oriented database management system must provide specialized storage structures for complex objects and specialized access paths for them. In general the problems related to object-oriented approach are performance, indexing and efficient storage management.

One last point is that, there is a need for a common data model for object-oriented databases. Such a data model will serve as a unifying force analogous to the relational model. Also, it will provide a compromise between programming language objects and database objects so that applications can easily share objects.

A. LIST OF BASIC ROUTINES

InitializeTables()
search_hierarchy(class_name,ptr)
initialize_ObjectTable()
insert_into_ObjectTable(oop,address)
search_ObjectTable(oop)
insert_into_hierarchy(class,super_place)
insert_into_IAT(head,oop)
search_IAT(entry,oop)
delete_IAT(entry,oop)

new_surrogate(flag)
getlocation(oop)
getoop(oop,field_index)
findchunk(object,class_oop)
change_value(oop,field_index,value)
class_of(oop)
size_of(oop)
total_size(oop)
field_count(oop)
delete(oop)
exists(oop)

isClass(oop)
field_offset(class_oop,variable_name)
type_of_field(class_oop,field_index)

```
definition_of(class_name)
name_of(class_oop)
coop_of(class_name)
anew(class_oop)
instance_count(class_oop)
isInstanceof(class_oop,oop)
first_instance_of(class_oop)
next_instance_of(class_oop)
super_of(class_oop)
place_inHierarchy(class_name)
subclass_of(class_oop)
next_subclass_of(class_oop)
inheritsvarname(class_oop,variable_name)
exists_varname(variable_name,class)
anewInstance(class_oop)
Initialize_Instance(oop)
PrintInstance(oop)
PrintallInstances(class_name)
PrintClassObject(class_oop)
subclasses(class_name)
allsubclasses(class_name)
LocalInstanceVariableNames(class_oop)
InheritedVariableNames(class_name)
allInstanceVariableNames(class_oop)
allInstances(class_name)

anewBag(class_oop,flag)
add2bag(aBag,oop)
addall(aBag,bag_oop)
add2set(aSet,oop)
includes(bag_oop,oop)
Union(bag1,bag2)
remove_element(aBag,oop)
remove_aCollection(aBag,bag_oop)
```

```
isEmptyBag(bag_oop)
isBag(oop)
isSet(oop)
PrintBag(bag_oop)
PrintElement(oop)

newchar(aCharacter)
isChar(oop)
asciivalue(oop)
digitvalue(oop)
isDigit(char)
isAlpha(char)
isAlphaNumeric(char)
printchar(oop)
readchar()

isInteger(oop)
integervalue(oop)
newinteger(value)
printinteger(oop)
readinteger()

anewArrayedCollection(class_oop,size,flag)
isString(oop)
length(oop)
stringcopy(oop1,oop2)
char_at(oop,index)
stringcat(oop1,oop2)
stringcompare(oop1,oop2)
printstring(oop)
readstring(size)
change_char_at(oop,index,value)
isArray(oop)
value_at(oop,index)
```

printarray(oop)

readarray(size)

change_value_at(oop,index,value)

REFERENCES

[1] Abiteboul, S., and R. Hull, *IFO: A Formal Semantic Database Model*, Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1984, pp.119-132.

[2] Abiteboul, S., and R. Hull, *IFO: A Formal Semantic Database Model*, Technical Report, University of Southern California, April 1984.

[3] Banerjee, J., H.J. Kim, W.Kim, and H.F. Korth, *Schema Evolution in Object-Oriented Persistent Databases*, Proc.of the 6th Advanced Database Symposium (Tokyo,Japan,Aug.) Information Processing Society of Japan's Special Interest Group on Database Systems, 1986, pp.23-31.

[4] Banerjee, J., H. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou, H. Kim, *Data Model Issues for Object-Oriented Applications*, ACM Transactions on Office Information Systems, Vol.5, No.1, January 1987, pp. 3-26.

[5] Beech, D., *Groundwork for an Object Database Model*, Research Directions in Object-Oriented Programming, ed. B. Shriver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp. 317-354.

[6] Christian, K., *The Unix Operating System*, John Wiley and sons, 1983.

[7] Copeland, G., and D. Maier, *Making Smalltalk a Database System*, Proc. ACM SIGACT/SIGMOD International Conference on the Management of Data, 1985.

[8] Date, C.J., *An Introduction to Database Systems*, Addison Wesley, Vol.1, 1986.

[9] Date, C.J., *An Introduction to Database Systems*, Addison Wesley, Vol.2, 1983, pp. 181-210.

[10] Diederich, J., and J. Milton, *Experimental Prototyping in Smalltalk*, IEEE Software, May 1987, pp.50-64.

[11] Fishman, D.H. et al., *Iris: An Object-Oriented Database Management System*, ACM Transactions on Office Information Systems, Vol.5, No.1, January 1987, pp. 48-69.

[12] Goldberg, A., and D. Robson, *Smalltalk-80:The Language and Its Implementation*, Addison-Wesley, 1983.

[13] Hailpern, B., and V. Nguyen, *A Model for Object-Based Inheritance*, Research Directions in Object-Oriented Programming, ed. B. Shriver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp. 147-164.

[14] Hammer, M. and D. McLeod, *Database Description with SDM:A Semantic Database Model*, ACM Transactions on Database Systems, Vol. 6, No. 3, September 1981, pp. 351-386.

[15] Hornick, M.F., and S.B. Zdonik, *A Shared, Segmented Memory System for an Object-Oriented Database*, ACM Transactions on Office Information Systems, Vol. 5, No. 1, January 1987, pp. 70-95.

[16] Karaorman, M., *Secondary Storage Management in an Object-Oriented Database Management System*, M.S. Thesis, Bilkent University, Ankara, July 1988.

[17] Kelley, A., and I. Pohl, *A Book on C*, The Benjamin/Cummings Publishing Company Inc., 1984.

[18] Kernighan, B.W., and D.M. Ritchie, *The C Programming Language*, Prentice Hall, 1978.

[19] Khoshafian, S.N., and G.P. Copeland, *Object Identity*, ACM OOPSLA'86 Proceedings, September 1986.

[20] Kulgarni, K.G., and M.P. Atkinson, *EFDM: Extended Functional Data Model*, The Computer Journal, Vol.29, No.1, 1986, pp. 38-46.

[21] Laff, M.R., and B. Hailpern, *SW2-An Object-Based Programming Environment*, Proc. of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1985, pp. 1-11.

[22] Lyngbaek, P., and V. Vianu, *Mapping a Semantic Database Model to the Relational Model*, ACM, 1987.

[23] Maier, D., and J. Stein, *Indexing in an Object-Oriented DBMS*, Proc. of the Workshop on Object-Oriented Database Systems, September 1986.

[24] Maier, D., A. Otis, and A. Purdy, *Object-oriented Database Development at Servio Logic*, Database Engineering, IEEE, Vol.8, No.4 ,December 1985.

[25] Maier, D., J. Stein, A. Otis, and A. Purdy, *Development of an Object-Oriented DBMS*, ACM Conference on Object-Oriented Programming Systems, Languages and Applications, 1986.

[26] Manola, F., and U. Dayal, *PDM: An Object-Oriented Data Model*, Intl. Workshop on Object-Oriented Database Systems, Pasific Grove, CA, September 1987.

[27] Nierstrasz, O.M., *What is the 'Object' in Object-Oriented Programming?*, Objects and Things, ed. D. Tsichritzis, Centre Universitaire D'Informatique, Université de Genève, March 1987, pp. 1-13.

[28] Nierstrasz, O.M., *A Survey of Object-Oriented Concepts*, Active Object Environments, ed. D. Tsichritzis, Centre Universitaire D'Informatique, Université de Genève, July 1988.

[29] Nierstrasz, O.M., *Hybrid - A Language for Programming with Active Objects*, Objects and Things, ed. D. Tsichritzis, Centre Universitaire D'Informatique, Université de Genève, March 1987, pp. 14-24.

[30] Özelçi, S., *Message Passing in an Object-Oriented Database Management System*, M.S. Thesis, Bilkent University, Ankara, July 1988.

[31] Özelçi, S., N. Kesim, M. Karaorman, E. Arkun, *An Experimental Object-Oriented Database Management System Prototype*, To appear in The

Third International Symposium on Computer and Information Sciences, October 1988,Çeşme, İzmir.

[32] Pascoe, G.A., *Elements of Object-Oriented Programming*, Byte, August 1986, pp. 139-144.

[33] Penney, D.J., and J. Stein, *Class Modification in the GemStone Object-Oriented DBMS*, ACM OOPSLA'87 Proceedings, 1987.

[34] Purdy, A., B. Schuchardt, and D. Maier, *Integrating an Object Server with Other Worlds*, ACM Transactions on Office Information Systems, Vol.5, No.1, January 1987, pp. 27-47.

[35] Shriver, B., and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, MIT Press Series in Computer Systems, 1987.

[36] Skarra, A.H., and S.B. Zdonik, *Type Evolution in an Object-Oriented Database*, Research Directions in Object-Oriented Programming, ed. B. Shriver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp. 393-415.

[37] Stefik, M., and D.G. Bobrow, *Object-Oriented Programming:Themes and Variations*, AI Magazine, January 1986, pp. 40-62.

[38] *Sun System Overview*, Sun Microsystems Inc., 1986.

[39] Synder, A., *Inheritance and The Development of Encapsulated Software Components*, Research Directions in Object-Oriented Programming, ed. B. Shriver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp. 164-188.

[40] Ullman, J.D., *Principles of Database Systems*, Computer Science Press, 1982.

[41] Zaniolo, C., et al., *Object-Oriented Database Systems and Knowledge Systems*, 1st International Workshop on Expert Database Systems, 1985, pp.1-17.