

A RUN-TIME ENVIRONMENT 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 SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Can YENİÖL
July, 1989

QA
76-9
.D3
Y39
1989

A RUN-TIME ENVIRONMENT 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
Can Yengül
1989

Can Yengül
tarafından ~~tarafından~~

QA

76.9

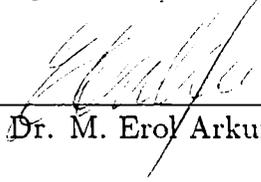
.D3

439

1989

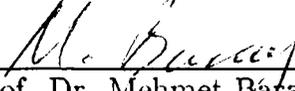
B1856

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.



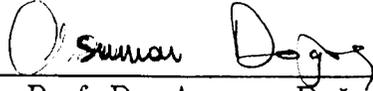
Prof. Dr. M. Ero Arkun(Principal Advisor)

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.



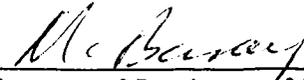
Prof. Dr. Mehmet Baray

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.



Prof. Dr. Asuman Dogac

Approved for the Institute of Engineering and Sciences:



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

ABSTRACT

A RUN-TIME ENVIRONMENT FOR AN OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM

Can Yengül

M.S. in Computer Engineering and Information Sciences

Supervisor: Prof. Dr. M. Erol Arkun

1989

In this thesis, an object-oriented query processor, a database language executer, and the protocols for the system- defined classes are designed and implemented. The designed and implemented database language completely fulfills the requirements of the object-oriented paradigm.

Query processing functions are implemented through the message passing paradigm, which results in a uniform treatment of data manipulation and query processing functions. The run-time environment also supports the implementation of inheritance mechanism, class hierarchy maintenance, instance access and modification, and access to class definitions.

Keywords : object-oriented database, query processor, object- oriented query model, object-oriented language, object, class, instance, message, method, class hierarchy, object identity.

ÖZET

NESNESEL BİR VERİ TABANI İŞLETİM SİSTEMİ İÇİN ÇALIŞMA ORTAMI

Can Yengül

Bilgisayar Mühendisliği ve Enformatik Bilimleri Bölümü

Yüksek Lisans

Tez Yöneticisi: Prof. Dr. M. Erol Arkun

1989

Bu tez çalışmasında nesnel bir sorgulama işlemcisi, nesnel bir veri tabanı dili için çalıştırıcı ve sistemde bulunan sınıflar için gerekli iletişim protokolleri tasarlanmış ve gerçekleştirilmiştir. Geliştirilen veri tabanı dili nesnel yaklaşımın tüm gereklerini yerine getirmektedir.

Sorgu işleme fonksiyonları mesaj yollama yöntemi ile gerçekleştirildiğinden, veri kullanımı ve sorgu işlemleri benzer şekilde ifade edilebilmektedir. Aktarım mekanizması, sınıf sıradüzeninin kurulması, nesne örneklerine ve sınıf tanımlamalarına ulaşım da program çalıştırma ortamı tarafından gerçekleştirilmektedir.

Anahtar sözcükler : nesnel veri tabanı, sorgulama işlemcisi, nesnel sorgulama işlemcisi, nesnel sorgulama modeli, nesnel dil, nesne, sınıf, nesne örneği, mesaj, metot, sınıf sıradüzeni, nesne kimliği

ACKNOWLEDGEMENT

I would like to acknowledge the valuable help, cooperation and encouragement of Prof. Dr. M. Erol Arkun. I would also like to thank Sibel Türkmen with whom I worked together throughout the development of an object-oriented database management system prototype (ODS) for her help and friendly cooperation. I also acknowledge the help and support of Pınar Ayer and appreciate the support of my lovely family.

TABLE OF CONTENTS

1	INTRODUCTION	1
2	THE OBJECT-ORIENTED APPROACH	3
2.1	The Basic Concepts	3
2.2	Basic Characteristics of Object-Oriented Systems	5
2.2.1	Data Abstraction	5
2.2.2	Homogeneity	6
2.2.3	Independence	6
2.2.4	Information Hiding	7
2.2.5	Inheritance	7
2.2.6	Late Binding	8
2.2.7	Message Passing	8
2.2.8	Object Identity	9
2.2.9	Overloading and Genericity	10
2.2.10	Reusability	11
3	THE OBJECT-ORIENTED DATABASES	12

3.1	Object-Oriented Versus Traditional Databases	13
3.2	Advantages of the Object-Oriented Model	14
3.3	Disadvantages of the Object-Oriented Model	15
4	THE ODS PROTOTYPE	17
4.1	An Overview	17
4.2	Implementation of the Classes	19
4.3	The Database Language	35
4.3.1	An Overview	35
4.3.2	Basic Constructs in the Language	38
4.4	The Run-Time Environment	49
4.4.1	The Necessary Structures for the Run-Time Environment	49
4.4.2	The Executor Module	51
4.4.3	The Expression Evaluation Module	54
4.4.4	The Message Passing Module	60
4.4.5	The Object Memory Module	63
4.5	The User Interface	69
4.5.1	The Class Browser	69
4.5.2	The Programming Shell	69
5	QUERIES IN OBJECT-ORIENTED DATABASES	73
5.1	Object-Oriented versus Relational Queries	73
5.2	The ODS Query Model	75

5.2.1	Predicate Construction	75
5.2.2	The ODS Query Language	77
5.3	Object-Oriented Query Processing in other Systems	81
5.3.1	GEMSTONE	81
5.3.2	ORION	82
6	AN APPLICATION WITH ODS	84
6.1	The Example Object-Oriented Database Schema	84
6.2	Example Programs	89
7	CONCLUSIONS	94
	REFERENCES	97
	APPENDICES	101
A	List of Run-Time Errors	101

LIST OF FIGURES

4.1	The Initial Class Hierarchy	18
4.2	A Class Describing Object	23
4.3	An Instance Variable Definition Table Entry	24
4.4	A Method Definition Table Entry	25
4.5	An Argument Definition	25
4.6	A BAG/SET Object	28
4.7	An ARRAY Object	31
4.8	A STRING Object	32
4.9	A BLOCK Object	34
4.10	A User-Defined Object	34
4.11	An Expression Code	57
4.12	An Arithmetic Expression Code	59
4.13	A Message Expression Code	64
4.14	The Class Browser	68
4.15	The Programming Shell	70
4.16	The Run-Time Window	71

4.17 The Read Panel	72
6.1 The Example Hierarchy	85

1. INTRODUCTION

Object-oriented systems are considered to be of significant value in domains such as software engineering, computer graphics and office automation systems. They combine well-known techniques such as modularization and data abstraction, and present a new framework for applications [4, 6, 9, 19].

The notion of objects allows any real world entity to be modelled by an object. Naturally, the closer the constructs in a language are to the entities we deal with in the real world the less difficulty we encounter in translating the real-world problem into a program. The object-oriented approach is a major step in this direction, since working with objects seems more natural than with constructs found in standard languages.

The object-oriented approach is the most promising technique now known for attaining such objectives as extendibility and reusability.

Currently, the complexity of applications such as CAD/CAM, document retrieval and expert systems need more powerful data modelling concepts. It has been believed that object-oriented databases are a step in this direction. They provide more flexible modelling tools than traditional database systems. They also incorporate some of the software engineering methodologies, such as data abstraction, that have proved to be effective in the design of large-scale software systems [14, 19].

Object-oriented databases are emerging to support these complex applications. Developed from the concepts of object-oriented programming, object-oriented databases reduce the semantic gap between complex applications and the data storage supporting those applications.

The basic idea of an object-oriented database is to represent an item in the real world being modelled with a corresponding item in the database. This includes the behaviour of each object as well as its structure. This one-to-one mapping reduces the semantic gap between the real world and the database modelling the real world [30].

In this thesis an object-oriented query processor and a database language executor has been designed and implemented for an object-oriented database management system prototype (ODS) that has been under development since 1987 in Bilkent University [16, 18, 27, 28, 34, 35]. The work also contributed to the design of the database language. Other parts of ODS include a user interface which performs basic schema evolution functions, a compiler and a code generator for the object-oriented database language [34, 35].

In Chapter 2, the basic concepts and characteristics of the object-oriented approach are given. Object-oriented databases, a comparison of object-oriented databases and traditional databases, advantages and disadvantages of object-oriented systems and databases are given in Chapter 3. Chapter 4 discusses the designed and implemented object-oriented database system (ODS) in detail. First, an overview of the system is given. Second, the protocols and the internal representations of objects are explained. Third, the object-oriented database language developed is discussed in detail. Then, the run-time modules of ODS, namely the executor, the expression evaluation, the message passing, the object memory module and their related structures are discussed. Finally, the user-interface module including the Class Browser and the Programming Shell modules are described. In Chapter 5, the basics of object-oriented query processing, a comparison of object-oriented and relational queries, the ODS query model and query language and information about a few database systems that have dealt with the object-oriented query processing are discussed. Chapter 6 presents an example database application developed using ODS. Chapter 6 is the conclusion.

2. THE OBJECT-ORIENTED APPROACH

2.1 The Basic Concepts

Instead of having two types of entity that represent information and its manipulation independently, an object-oriented system has a single type of entity, the object, that represents both. Objects can be manipulated as ordinary data while they describe manipulation, like procedures, as well. The data contained in an object is manipulated through sending a message to the object.

Formally, an object consists of a private memory and a public interface part. The private memory part consists of instance variables capturing the state of the object. The instance variables can contain other objects which have their own states [4, 29, 31].

The set of messages that an object can respond to by executing related pieces of code constitutes its public interface part characterizing the behaviour of the object. The private memory part is only accessible through the public interface part.

When an object receives a message, it determines how to manipulate itself. The object to be manipulated is called the receiver object. A message contains a message name, which is also called a message selector, and possibly some arguments. The message selector describes what the programmer wants to happen, not how it should happen.

Procedures have names too, and they are called by using their names.

Nevertheless, each procedure name corresponds to only one procedure, therefore specifies exactly what should happen. However, a message can be interpreted in different ways by different receivers. Therefore, the receiver of the message determines exactly what will happen, not the message itself.

For each message, there is a procedure-like entity called a method which implements the response when a message is sent to an object. Although methods are procedure-like entities, they can only communicate through messages and can not communicate directly.

Most object-oriented systems make a distinction between the description of an object and the object itself. The description of an object is called a class, since the class can describe a whole set of related objects. Formally, a class can be defined as a description of one or more similar objects. Each object described by a class is called an instance of that class [4, 31].

Inheritance is the concept in object-oriented approach that is used to define objects that are almost like other objects. Inheritance mechanism is important because it makes the declaration of shared specifications possible. It helps to keep programs shorter and more tightly organized. There are two types of inheritance, namely hierarchical inheritance and multiple inheritance. In a hierarchy, a class is defined in terms of a single superclass. A specialized class modifies its superclass with additions and substitutions. The hierarchy of classes is called the class hierarchy. On the other hand, multiple inheritance makes it possible to combine descriptions from several classes. In multiple inheritance the class hierarchy takes the form of a class lattice [4, 31].

2.2 Basic Characteristics of Object-Oriented Systems

Currently, the following notions are associated with the object-oriented approach [25, 26, 29, 40] :

- Data abstraction
- Homogeneity
- Independence
- Information hiding
- Inheritance
- Late binding
- Message passing
- Object identity
- Overloading and genericity
- Reusability

2.2.1 Data Abstraction

The principle that is fundamental to the object-oriented approach is encapsulation or data abstraction. Data abstraction means that one is not interested in the representation of an object but its behaviour [24].

A programmer defines an abstract data type that consists of an internal representation and a set of methods to access and manipulate the data contained in the internal representation. Object-oriented languages enable the programmer to create his own abstract data types. Although languages like C and Pascal support the construction of programmer-defined types, one cannot define operations that are only applicable to that type.

The separation of interface and implementation of a new type makes the types representation independent. It allows the modification of the methods of a class without affecting the other classes that reference the class being modified, because other classes can only communicate with the instances of this class through messages. Therefore, the portability of software increases.

2.2.2 Homogeneity

In order to have a fully object-oriented system, everything should be an object. The degree of homogeneity depends on whether programs and classes are objects, there is a difference between user-defined and system defined objects.

Theoretically, to have everything as an object seems to be attractive. However, this introduces some circularity that has to be broken at some level. For example, assume that some messages are also objects. Then, in order to manipulate a message, one has to use a message, a circularity. Therefore, the degree of homogeneity is a question.

2.2.3 Independence

Object models often encapsulate objects in terms of a set of operations as a visible interface, while hiding the realization of an object. Realization of an object consists of its data structures and implementation of the operations. Since the only mechanism to communicate between objects is through messages, object independence is enforced. Objects have their own control over their own state and the object's methods are the only way to manipulate its state. If an object can access another object's state, then it is clear that the fact that the implementation and the interface are independent is not correct.

2.2.4 Information Hiding

Information hiding reduces the interdependencies between software modules and allows the development of reliable and easily modifiable software systems. The state of a module is represented by a set of instance variables which are private to the module and only a set of local methods are allowed to manipulate these variables. Since the only way to manipulate an internal state of a module is to use messages which constitute the public interface part of a module, the internal data structures and methods can be easily modified without affecting the implementation of other modules.

2.2.5 Inheritance

Inheritance is a reusability mechanism for sharing behaviour between objects. It enables a programmer to create classes and, therefore, objects that are specializations of other objects. Creating a specialization of an existing class is called subclassing. The new class is a subclass of the existing class and the existing class is the superclass of the new class. The subclass inherits instance variables, class variables and methods from its superclass. The subclass may add instance variables, class variables and methods that are appropriate to the more specialized objects. A class may also override or provide additional behaviour to the methods of its superclass.

Class inheritance is an important mechanism which can simplify large pieces of software by using the similarities between certain classes. The key idea of class inheritance is to provide a simple and powerful mechanism for defining new classes that inherit properties from existing classes.

In hierarchical inheritance, a class is defined in terms of a single superclass. This is also called simple or single inheritance. A natural extension is multiple inheritance which increases sharing by making it possible to combine descriptions from several classes [31].

2.2.6 Late Binding

In a typical high-level language such as C, when a function is called, the compiler and the linker generate a subroutine call to a physical address. This is rather efficient, but one has to be careful about the arguments, because a type mismatch may cause severe errors. But specifying some directives, the compiler's type checking facility might catch those type mismatches.

The object-oriented languages relieve the programmer from this problem by automatically calling the appropriate method for a given data structure. The programmer uses generic message selectors, and the system determines the method corresponding to this selector from the class of the receiving object.

Since all references are symbolic, a method can be recompiled without having to recompile all of its callers. In ODS, the same name can be used to identify a method or a message which operate in a similar way in several different classes. For example, each class can implement its own class definition.

The fact that a single message can invoke any one of several methods depending on the receiver object is the most important feature of the object-oriented approach. A lot of control structure, such as if and case statements, are not required because the executor determines the code to be executed according to the receiver object.

However, late binding has the important disadvantage of reduced efficiency. Research continues on improving the efficiency of the late binding approach [12].

2.2.7 Message Passing

In conventional programming languages, active procedures act on passive data that is passed to them via arguments. Object-oriented languages employ a data or object-centered approach to programming. Instead of passing data

to procedures you ask objects to perform operations on themselves. All of the action in object-oriented programming comes from sending messages between objects.

Message sending supports data abstraction. The calling program cannot make any assumptions about the implementation and internal representation of the receiving objects.

2.2.8 Object Identity

Identity is the property of an object that distinguishes it from all others. Most of the programming languages and database languages use variable names to distinguish temporary objects which mixes the addressability and identity. On the other hand, most of the databases use identifier keys to distinguish objects which mixes the data value and identity. However, object-oriented languages offer a different approach for identification which is independent of the address and the data value of an object [19].

If the concept of identity is built into a language, then an object's uniqueness is modelled even though its description is not unique. Use of identifier keys causes several problems because of mixing data value and identity concepts. First, identifier keys are not allowed to change, although they are user-defined data. Second, sometimes any attribute or a set of attributes of an object cannot uniquely determine the object. Then some artifacts have to be introduced. Third, the choice of attribute(s) to use for an identifier key may need to change. Fourth, use of identifier keys causes joins to be used in retrievals instead of the more desirable path traversal.

There is a growing trend to merge programming and database languages into a hybrid environment which includes a language with unified typing and computational identity.

The most powerful technique for supporting identity is through surrogates [19]. Surrogates are system generated globally unique identifiers, completely

independent of any physical location. Surrogates provide full location independence. If surrogates are associated with every object then they also provide full data independence. Each object is associated with a unique identifier which is called an object-oriented pointer (oop).

We may talk about three predicates, namely, identity, shallow equality and deep equality predicates. Given two objects, the identity predicate returns true if their oops are the same. Two objects are shallow-equal if their values are identical. However shallow equality predicate is not recursive. For example, two set objects whose elements have pairwise equal values are not necessarily shallow-equal. Two atomic objects are deep-equal if their values are the same. Two set objects are deep-equal if they have the same cardinality and the elements in their values are pairwise deep-equal. One may implement operators to obtain another object which is shallow-equal or deep-equal to the original object.

2.2.9 Overloading and Genericity

Another feature of object-orientation is operator overloading. Overloading means attaching more than one meaning to a name, such as the name of an operation. Operator overloading describes the useful notion of using the same operator symbol to denote distinct operations on different data types.

For example, the same notation can be used to add two integers, two floating point numbers or an integer and a float.

Another technique which complements overloading is genericity. Genericity allows a module to be defined with generic parameters that represent types. Instances of the module are then produced by supplying different types as actual parameters. This is a definite aid to reusability because just one generic module is defined, instead of a group of modules that differ only in the types of objects they manipulate.

Both overloading and genericity leads to reusability.

2.2.10 Reusability

Encapsulation of procedures, macros and libraries has been exploited for many years to enhance the reusability of software. Object-oriented techniques achieve further reusability through the encapsulation of programs and data [24].

Inheritance enables programmers to create new classes of objects by specifying the differences between a class and an existing class instead of starting from scratch each time. A large amount of code can be reused in this way.

3. THE OBJECT-ORIENTED DATABASES

A database is normally used to maintain a model of some aspect of reality. Traditional data models, such as the relational model, have achieved great efficiency in data storage and retrieval, however it is subject to the limitation of a finite set of data types and the need to normalize data [22].

In contrast, object-oriented languages offer flexible abstract data-typing facilities and the ability to encapsulate data and operations via the message passing paradigm. Combining object-oriented language capabilities and the storage management functions of a traditional data management system yields an object-oriented database system which reduces application development time and increases modelling power [23].

Object-oriented databases are emerging to support complex applications such as CAD/CAM, document retrieval, expert systems and decision support systems. Developed from the concepts of object-oriented programming, object-oriented databases reduce the semantic gap between complex applications and the data storage supporting those applications.

The basic idea of an object-oriented database is to represent an item in the real world being modelled with a corresponding item in the database. This includes modelling the behaviour of each object as well as the object's structure.

3.1 Object-Oriented Versus Traditional Databases

Object-oriented systems emphasize object-independence by encapsulation of individual objects. Objects' contents and the implementation of their operations are hidden from other objects. Interaction with objects is through a well-defined interface [36].

Traditional databases, on the other hand, emphasize data independence by separating the world into two independent parts, namely the data and the applications operating on them.

Traditionally, databases make a very strong distinction between instances and classes. Instances are in the database, whereas class information, i.e., the schema, is stored in the data dictionary.

It is obvious that object-oriented systems need to manage both instances and classes. In object-oriented systems, classes are themselves objects and they can be manipulated as objects.

Databases traditionally provide operations based on selection by contents. This is especially true in relational systems, where all relationships between entities are represented by contents, and all operations are based on contents.

In object-oriented systems object contents are typically encapsulated, i.e., hidden. We are not supposed to know the values of an object's variables. Since objects encapsulate behaviour, they should also be selectable in terms of their behavioural aspects rather than by how the behaviour is implemented.

Databases traditionally have very few classes with a large number of instances per class. The differentiation between entities is represented by attribute contents and not by subdividing or creating extra classes. Classification in object-oriented systems serves a very different function which support instantiation, encapsulation and class inheritance.

Database systems traditionally provide very few generalized types. Therefore, they provide a small number of operations for queries and updates on database objects. The operations are the same regardless of the semantics

of the object involved. Queries and updates on employees, cars, accounts all utilize the same operations. In addition, these operations are simple.

Object-oriented systems require that all objects provide some set of operations which are shared through object classes and inheritance mechanisms. In addition, the methods can be very complex.

Database systems utilize object identifiers internally for implementation purposes. In the relational model tuples do not have a visible identifier. They're identified by their contents, via primary or secondary keys.

In object-oriented systems object identifiers are very important for two reasons. First, identifiers provide a permanent handle for objects that may move in much the same way that file names hide the fact that a file's contents and physical location may change. Second, if an object's contents are properly encapsulated they cannot be expected to provide a means for identification. These identifiers should be purely for identification purposes and should not be related to the physical location of the objects in the database.

Traditional databases allow very little flexibility for evolution of their classes. Schema evolution is very restricted. However, some of the relational systems allow adding new attributes. In object-oriented systems object classes should be able to change to accommodate software evolution [3].

This is obvious, because the classes are also objects as any ordinary entity.

3.2 Advantages of the Object-Oriented Model

An object-oriented model supports modelling of complex objects and relationships directly and organizes classes of data items into an inheritance hierarchy. A single entity is modelled as a single object, not as multiple tuples spread among several relations [22].

One of the characteristics of object-oriented systems, object identity, allows a data object to retain its own identity through arbitrary changes. Two entities which both include the same information can be modelled as two

objects with a shared subobject that contain the common information. Such sharing reduces the update anomalies that exist in the relational data model [10]. We note that referential integrity [10] is directly satisfied in object-oriented data model. One object refers directly to another instead of referring to the name of that object. The reference can not be created if the other object does not exist. Therefore there are no dangling identifiers [22].

Information hiding and data abstraction increase reliability and make applications independent of procedural and representational specifications which are defined in the classes [29].

The class structure speeds application development. Dynamic binding increases flexibility by permitting the addition of new classes of objects without having to modify the existing code. Inheritance mechanism allows code to be reused which reduces the amount of code written by a programmer and increases his productivity.

Building and managing a database schema requires a great effort to maintain consistency between records, fields, relations, data types and values. Therefore, data dictionary facilities have been static in nature. However, building a database schema as an object-oriented hierarchy provides an assistance for automatically describing data representations and transparently maintaining them. Schema descriptions are represented as objects and procedures for adding, modifying and deleting dictionary objects are implemented in methods associated with the schema object.

3.3 Disadvantages of the Object-Oriented Model

Although object-oriented databases are being built and have practical applications, there is no agreement as to a standard data model for object-oriented databases. We do not have the equivalent of the relational algebra for an object-oriented data model [26]. Therefore it is also difficult to decide on a standard query language for objects.

The cost of dynamic binding is also encountered as a disadvantage.

Object-oriented databases provide a database language which include the data definition and data manipulation facilities together with computational aspects. These languages operate on database objects directly. The implementation of these languages are more complex compared to any other procedural language because the semantic gap between these languages and typical hardware is greater.

4. THE ODS PROTOTYPE

4.1 An Overview

ODS supports modelling of complex objects and relationships directly. Any real world entity can be modelled by an object. The state of an object is captured in the instance variables. The domain of an instance variable is not restricted to be a simple data type but can be other entities of arbitrary complexity [34, 35].

ODS represents the behaviour of the real world entities in addition to its structures. The behaviour of an object is encapsulated in the methods. Each object responds to a set of messages which constitutes its public interface part. For each message, there is a corresponding method which implements the message.

Similar objects are grouped into a class. Classes define the internal structure and behaviour of their instances. In ODS both classes and instances are viewed as objects. This allows a uniform treatment of messages. Since classes are objects, they also respond to messages which are called class messages. For example, in order to create an instance of a class, the class message `New` is sent to that class.

Grouping objects into classes helps avoid specification and storage of redundant information. In ODS, a class hierarchy is maintained. The system initially comes with a set of classes which helps the development of applications a great deal. These classes are called system-defined classes.

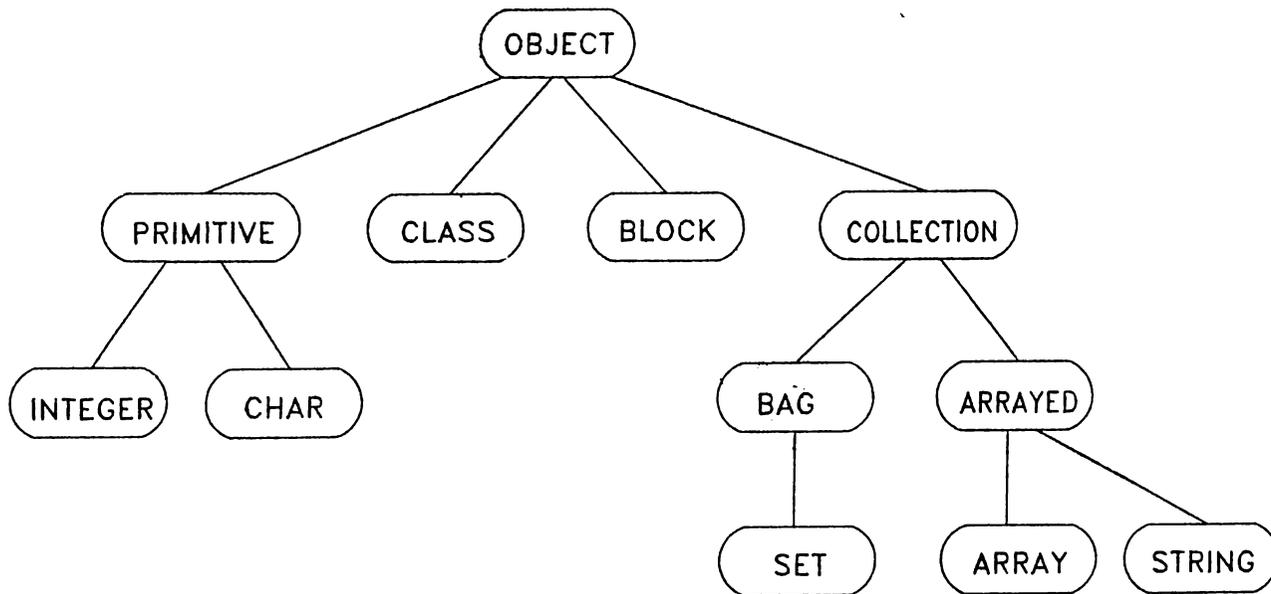


Figure 4.1: The Initial Class Hierarchy

The user may also add new classes to the system which are called user-defined classes. Both the system-defined classes and the user-defined classes are treated uniformly. The system defined-classes are OBJECT, CLASS, PRIMITIVE, CHAR, INTEGER, COLLECTION, BAG, SET, ARRAYED, ARRAY and STRING classes. The PRIMITIVE, COLLECTION and ARRAYED classes are abstract classes which have no instances. The initial class hierarchy is shown in Figure 4.1.

It is possible to create temporary objects which are the instances of BAG/SET, ARRAY, ARRAY/STRING classes. The New method of each of these classes expects an argument specifying the newly created object to be a temporary ('T') or persistent ('P') object. All the user-defined objects are persistent.

ODS supports class variables to reduce redundant storage and specification of objects. Each user-defined class can define a set of class variables that are shared by all instances of the class.

The variables that are supported in ODS are always bound to a specific

class which is desirable for integrity control.

ODS supports identity which is implemented through surrogates which are globally unique system generated identifiers. Objects can be shared through their object-oriented identifiers which is also called an object-oriented pointer (oop). The relationship between objects are represented by object-oriented pointers which automatically satisfy the referential integrity requirements.

In ODS set valued entities are supported directly through the instances of the SET class. A SET object can have arbitrary objects as elements and it needs not be homogeneous. Sets are extensively used in establishing 1:N and N:M relationships and in query processing.

ODS provides a database language which includes data manipulation facilities and computational aspects. The language is strongly typed. Both the language and the database support the same data types and solve the impedance mismatch problem. Queries can also be expressed in this language.

The object-oriented database language does not yet have a data definition capability. The modification of existing classes and addition of new classes are handled by the Class Browser module [34, 35] instead.

The implementation of ODS has been carried out on SUN Workstations¹ running Berkeley UNIX² 4.2 [7, 32, 38] using the C programming language [17]. The internal representations of objects and class hierarchy is taken from the predecessor of ODS [16, 18, 27, 28] which was partially implemented on SUN workstations.

4.2 Implementation of the Classes

In this section, the internal representation of objects and protocols of the system defined classes will be given. The internal representation of the objects differ from each other. The instances of the system-defined classes have different internal representations. The user-defined classes share the same internal

¹SUN Workstation is a registered trademark of SUN Microsystems, Incorporated

²UNIX is a trademark of Bell Laboratories

representation, but this is also different from the internal representation of the system-defined classes. There are classes which are created for a logical grouping of their subclasses which define the common properties of their subclasses but they don't have any instances. These classes are called abstract classes. For example, PRIMITIVE, ARRAYED, COLLECTION classes are abstract classes of the ODS class hierarchy [35].

The system-defined classes have their own message protocols. Each class implements its own methods and messages which constitute the public interface of its instances. The following sections will discuss the internal representations and the message protocols of the classes in the ODS class hierarchy.

The OBJECT Class :

The protocol common to all the objects in the system is provided in the description of the OBJECT class. The OBJECT class does not have any instances. Therefore, it is an abstract class. However it implements several class and instance methods that may be used by its subclasses or the instances of its subclasses. These methods provide a default behaviour to the instances of the subclasses of the OBJECT class. However, they may provide a basis to construct specialized versions of other methods.

The Protocol for the OBJECT Class

`print(new_line) :`

The receiver object is printed to the run-time window. If the new_line argument is TRUE a NEWLINE character is also printed. Since all system-defined classes implement their own print methods, this message is used for only instances of the user-defined classes. The method is recursively defined so that every instance variable of the receiver object is expanded until a primitive object is reached. The message returns TRUE if it successfully completes its operation, FALSE otherwise, to comply with our convention that each method returns a value.

`New()` :

The receiver object of the class messages is always a class object. The system-defined classes define their own `New` methods, therefore this message can only be applied to the user-defined classes. The `New` message creates an instance of the receiver class object by allocating chunks for each of the superclasses of the receiver class and initializes all the instance variables to `NIL`. Finally it returns the oop of the newly created object.

`Getset()` :

As will be presently discussed in the class representing object structure, each class maintains a `SET` object which includes all the oops of the instances of that class. This message returns the oop of the `SET` object that represents the instances of the receiver class.

`remove()` :

The receiver object is logically deleted by marking the status field in its corresponding object-table entry.

There are some other messages for equality checks and copying objects. However, these are not implemented yet. These can be listed as follows :

`shallow_equal(object)` :

Determines if the receiver object and the argument object are shallow equal. Returns `TRUE` or `FALSE`.

`deep_equal(object)` :

Determines if the receiver object and the argument object are deep equal. Returns `TRUE` or `FALSE`.

`Shallow_copy(object) :`

The receiver class creates a new instance and copies the contents of the argument object into the newly created object. Returns the oop of the newly created object.

`Deep_copy(object) :`

The receiver class creates a new instance and for each instance variable of the receiver class a new instance of its domain is created. This continues until a primitive domain is reached. Then, the contents are copied from the argument object to the receiver. The oop of the newly created object is returned.

The CLASS Class :

Each class in the system is an instance of the CLASS class. Both the user-defined classes and the system-defined classes are represented by a class defining object. Each class object describes the structure and the behaviour of the instances of the class it represents. The class describing object has the following information as shown in Figure 4.2 :

- oop of the class
- name of the class which also describes the type of the instances of the class
- oop of the superclass of the class
- oop of the set object which represents the oops of the instances of the class
- instance variable count which is used when allocating space for an instance
- a pointer to the instance variable definition table
- class variable count

Class oop
Class name
Oop of the super class
Instance set oop (oop of the set of the instances)
Instance variable count
Pointer to instance variable definitions
Pointer to class variable definitions
Class variable count
Pointer to instance methods
Pointer to class methods
Pointer to place in hierarchy

Figure 4.2: A Class Describing Object

- a pointer to the class variable definition table
- a pointer to the instance method definition table
- a pointer to the class method definition table
- a pointer to the class hierarchy entry to specify the position of the class in the class hierarchy which provides a path to access the class's superclass chain and its subclasses.

The definitions of the instance variables are stored in an instance variable definition table (IVDT) [18]. The instance variable definition table contains the following information as shown in Figure 4.3:

- name of the instance variable
- type of the instance variable
- size of the instance variable if it is an indexed type
- element type of the instance variable if it is an indexed type
- a pointer to the next instance variable definition table entry

Name
Type
Size
Element type
Pointer to next variable

Figure 4.3: An Instance Variable Definition Table Entry

The definitions of class variables are stored in a class variable definition table (CVDT). The structure of CVDT is nearly the same, but there is an additional entry to store the values of the class variables. Since the value of a class variable is shared among all the instances of a class, the value of the class variable is also kept with its definition.

Both the definitions of the instance methods and the class methods are put into the method definition table (MDT). The method definition table contains the following information as shown in Figure 4.4 :

- a flag indicating whether the corresponding method is implemented as a C function or in the ODS database language
- a pointer to a C function for methods written as C functions
- name of the method
- message selector name of the method
- the name of the file that contains the method
- the number of arguments of the method
- a pointer to the list of argument definitions. Each node contains the following information as shown in

Figure 4.5 :

- type of the argument
- size of the argument if it is an indexed type

C_code
Function pointer
Method Name
Message name
Argument count
Method file name
Pointer to the list of arguments
Pointer to the next method

Figure 4.4: A Method Definition Table Entry

Type
Maximum length
Element Type
Symbol table index
Pointer to the next argument

Figure 4.5: An Argument Definition

- element type of the argument
- the index of the corresponding symbol table entry for the argument. Argument values are put into the oop field of the symbol table using this index during parameter passing operation
- a pointer to the next argument definition

Since the data definition facility is not included in the ODS database language yet, the CLASS class does not implement its methods to perform the definition of new classes and modification of existing classes. These functions are put into a submodule of the system which is called the Class Browser [34, 35] and it will be discussed in section 4.6.1.

The INTEGER Class :

The instances of the INTEGER class has only one state which is the value represented and this never changes. Integers have their values encoded in

their object-oriented pointers which provides efficiency in their manipulation by the system. The object-oriented pointers of the INTEGER objects have their least significant bits 1. The oops of the instances of other classes never have 1 in their least significant bits. For example the integer value 30 is represented by 61. First, the integer value is shifted left one bit. Then, 1 is added to this value to obtain the oop of this value.

The Protocol for the INTEGER Class :

`print(new_line)`

Converts the destination object into its integer format and then prints the integer value to the run-time window. If the `new_line` argument is TRUE, a NEWLINE character is also printed.

`Read(prompt_string)`

Reads an integer value through a read panel which will be discussed in the user interface part. The `prompt_string` argument contains a string which is printed in the read panel to inform the user.

The CHARACTER Class :

Similar to the INTEGER class, characters have their values encoded in their object-oriented pointers. The ASCII values of the characters range between 0 and 255. Multiplying a value in this range by two yields an even integer between 0 and 510. Therefore, these do not overlap with the oops of integer values. For example, the character 'A' (65 ASCII code) is represented as 130.

The Protocol for the CHAR class :

`Read(prompt_string) :`

Reads a character value through a read panel. The prompt string is printed to the read panel to inform the user.

`print(new_line) :`

Prints the receiver CHAR object. If `new_line` argument is TRUE, a NEW-LINE character is also printed.

`ascii() :`

Returns the ASCII value of the receiver CHAR object as an INTEGER object.

`isdigit() :`

Tests if the receiver CHAR object is a digit. Returns TRUE or FALSE.

`isalpha() :`

Tests if the receiver CHAR object is an alphabetic character. Returns TRUE or FALSE.

`isalphanum() :`

Tests if the receiver CHAR object is an alphanumeric character. Returns TRUE or FALSE.

The BAG and SET Classes :

A BAG/SET object contains oops of objects that are instances of either system-defined classes or user-defined classes. The objects contained in a BAG/SET object are not necessarily of the same type, they may belong to arbitrary classes. An element of a BAG/SET object can also be another BAG/SET object.

The internal representation of both the BAG and SET objects as shown in Figure 4.6 are the same and contain the following information :

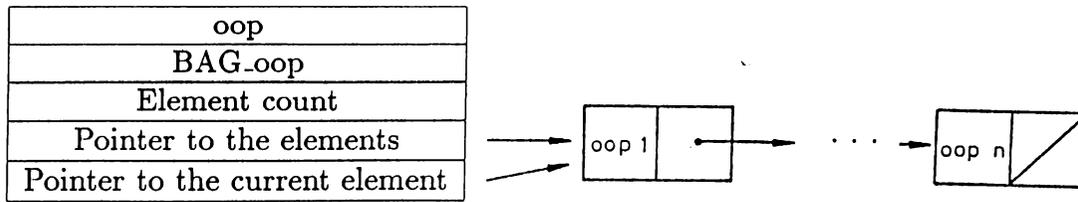


Figure 4.6: A BAG/SET Object

- oop of the BAG/SET object
- oop of the BAG/SET class
- number of elements in the BAG/SET object
- a pointer to a list of elements, each node of the list contains the oop of one of the elements
- a pointer to the current element, used in iterating over the elements.

The difference between a SET and a BAG object is that a SET object does not allow duplication of any of its elements. However, a BAG object allows duplicates.

The Protocol of the BAG Class :

`New(temporary_flag)` :

This message creates a BAG object and returns its oop. The `temporary_flag` specifies if the object will be a temporary ('T') or permanent ('P') object. The newly created BAG object represents an empty bag.

`add(object)` :

The object specified in the argument is inserted into the receiving BAG or SET object. The method implementing the message is a generic function to cover both the SET and BAG objects. If the object to be added to a SET object is a duplicate, the message returns FALSE. Otherwise TRUE.

`print(new_line) :`

Prints the receiving BAG/SET object. Each object included in the receiving object is printed one by one. If `new_line` is TRUE, a NEWLINE character will also be printed.

`include(bag_object) :`

All the elements of the argument BAG object are added to the receiving BAG object. Returns TRUE or FALSE.

`existobj(object) :`

Tests if the object is contained in the receiving BAG/SET object. Returns TRUE or FALSE.

`removeobj(object) :`

Removes the object from the receiving BAG/SET object.

`remove() :`

Removes the receiving BAG/SET object if it is empty.

`isempty() :`

Tests if the receiving BAG/SET object contains any elements. Returns TRUE or FALSE.

`first() :`

Returns the oop of the first element in the receiving BAG/SET object. It sets the `current_element` pointer to the beginning of the list. Returns NIL if the BAG/SET is empty.

`next()` :

Returns the oop of the next element in the element list of the receiving BAG/SET object. Sets the `current_element` pointer to the `next_element` in the list. Returns NIL if there is no next element.

The BAG class implements all the query processing methods as well. These will be discussed in section 5.2.2. These include the methods `retrieve`, `forall`, `forany`, `modify`, `count`, `countu`, `sum` and `sumu`.

The Protocol for the SET class :

Since the SET class is a subclass of the BAG class, it inherits all the methods defined in the BAG class. Some of them are directly applicable to the SET objects, but some of them are generic methods that change their behaviour according to the type of the receiver object. The only message implemented for the SET class is :

`include(set_object)` :

All the elements of the argument set object are added to the receiving SET object.

The ARRAYED Class :

An ARRAY object is a collection of arbitrary objects that can be accessed by integer indices. An ARRAY object can contain other ARRAY objects which can be of arbitrary size which allows the construction of multi-dimensional arrays. The internal representation of an ARRAY object is shown in Figure 4.7.

The protocol for the ARRAY class :

`at(index)` :

Returns the oop of the object whose location in the receiving ARRAY object is specified by the index.

	oop
	ARRAY_oop
	Size
0	<i>oop</i> ₁
1	<i>oop</i> ₂
	.
	.
	.
n-1	<i>oop</i> _n

Figure 4.7: An ARRAY Object

`changeat(index, oop) :`

The second argument `oop` is put into the location specified by the index in the receiving ARRAY object.

`print(new_line) :`

Prints the contents of the receiving ARRAY object to the run-time window. If `new_line` contains TRUE, it also prints a NEWLINE character.

The STRING Class :

A STRING object is a collection of characters which can be accessed by indices. The internal representation of a STRING object is shown in Figure 4.8.

The Protocol for the STRING Class :

`Read(prompt_str) :`

Reads a string through a read panel, creates a new STRING object and returns the oop of this object. The `prompt_str` is printed to inform the user.

	oop
	STRING.oop
	Size
0	<i>char</i> ₁
1	<i>char</i> ₂
	.
	.
	.
n-1	<i>char</i> _n

Figure 4.8: A STRING Object

`print(new_line) :`

Prints the receiving STRING object to the run-time window. If `new_line` is TRUE a NEWLINE character is also printed.

`length() :`

Returns the length of the receiving STRING object as an INTEGER object.

`strcpy(string_object) :`

Copies the contents of the argument string to the receiving STRING object. The length of the receiving STRING object should not be less than the length of the argument string.

`at(index) :`

Returns the character at index in the receiving STRING object.

`changeat(index, char_object) :`

The character specified by `index` in the receiving `STRING` object is replaced by `char_object`.

`strcat(string_object) :`

Concatenates the argument `string` to the receiving `STRING` object. The receiver object should contain enough spaces.

`strcmp(string_object) :`

Compares the receiver and the argument `STRING` objects. If they represent the same string, returns `TRUE`. Otherwise `FALSE`.

The BLOCK Class :

Any block literal that appears in a program or a method is represented by a `BLOCK` object during run-time. A block literal can contain any valid relational or arithmetic expressions including message expressions. Currently, block literals are only used in query processing to formulate selection and projection expressions which are discussed in section 5.2.2 in detail. Figure 4.9 shows the internal representation of a block object which contains the following information :

- oop of the `BLOCK` object
- oop of the `BLOCK` class
- pointer to a block of integer codes that represents the expression contained in the block literal. Each code is augmented with a line number field in order to identify a source line with errors to the user.

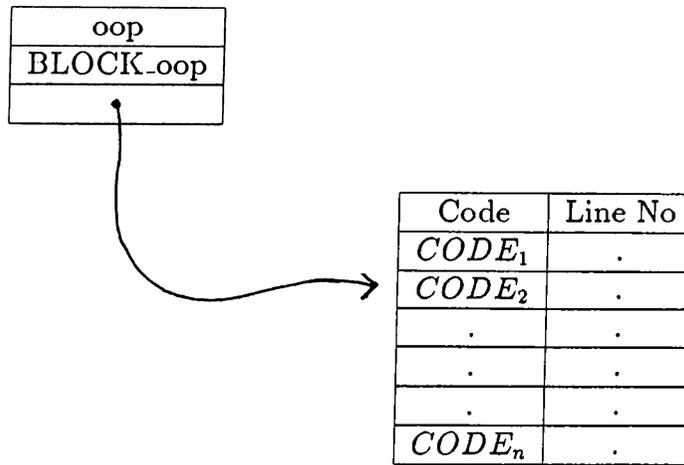


Figure 4.9: A BLOCK Object

oop
Class oop
Size
Oop of the super chunk
Value of the $variable_1$
Value of the $variable_2$
.
.
.
Value of the $variable_n$

Figure 4.10: A User-Defined Object

The User-Defined Classes :

Since these are user-defined classes , their structure and behaviour are defined by the user. The user specifies the names of instance variables and their corresponding domains which information is stored in the class describing object that is an instance of the class CLASS. The internal representation of all user-defined objects are the same as shown in the Figure 4.10.

The instance variables are put into contiguous memory locations each 32 bits wide. Each user-defined object starts with a header, that contains the oop of the object, oop of the object's class, and number of words allocated.

4.3 The Database Language

4.3.1 An Overview

In conventional systems, the emphasis is much more on programs than data. In traditional programming languages data that exist for the life time of the program are treated differently than the data that persist after execution. The data structures supported in files are usually not as rich as the data structures supported in main memory, which necessitates user-generated encodings to write structured values to files. In the database area, data manipulation languages do not provide arbitrary computational facilities which results in the requirement of an interface to a programming language. The interface can be in one of two forms :

- One language can be embedded into another.
- Procedure calls to the database system from within the programming language

Since we have two languages, the so called "impedance mismatch" problem arises [2, 8]. There might be two kinds of mismatches:

1. One mismatch is conceptual, the programming language and the data manipulation language might support different programming paradigms. One might be a procedural language while the other might be declarative.
2. The other mismatch is structural, the languages might not support the same data types, which results in a structure reflected at the interface.

For example, one can access a relational database using SQL [10] from COBOL. However, COBOL can operate only at the tuple level. Therefore, the relational structure is lost [8].

The consequences of the impedance mismatch are [2]:

1. More code has to be written because the programmer has to handle data conversion and binding of variables.
2. This code is hard to write because it is not related to the problem that programmer is solving but it is related to the system deficiencies.
3. It makes the programmer to decide on which environment to solve the problems : in the programming language or in the data manipulation language.
4. It reduces the performance of the system because it introduces a lot of unnecessary communication between the programming language and the database system.

To sum up, the impedance mismatch is a major deficiency of existing systems.

The object-orientation is a promising approach for solving the impedance mismatch problems, because encapsulation embodies data and programs in the same object. Programs become part of the database.

We can list three approaches for the choice of the language in an object-oriented data model :

- New language approach
- Existing language approach
- Multilanguage approach

In the new language approach, we define a new language which is specifically designed for this task. It has to add the features of a programming language, such as I/O.

In the existing language approach, we choose an existing programming language such as C or Pascal. The language can be chosen according to its

popularity or suitability. Then, the language chosen is connected to the data model.

In the multilanguage approach, the user is allowed to write methods in a set of existing programming languages.

Each of these three approaches have its advantages and disadvantages [2]. There is no optimal way to choose between these approaches. However, it will mainly depend on the type of objective assigned to a system in terms of research and development, whether one is technology driven or market driven.

The object-oriented database management prototype ODS is developed for pure research purposes to investigate and analyze the issues of object-oriented approach. Therefore we have chosen the new language approach. Although designing a new language is a long task and it may meet user resistance, the connection between the programming language and the data model is smooth and natural because the language is designed for that purpose.

ODS comes with a built-in class hierarchy that contains the system-defined classes. However, the user can extend the class hierarchy with user-defined classes. Therefore, during the execution of the system, the type system of ODS might be extended or modified. When a new class is added, the methods that belong to that class are compiled and linked to the system.

In order to solve the problem of dynamically adding new classes and new methods to the system, we have chosen the new language approach and decided to develop our own run-time environment. The modification of the existing methods also require dynamic compilation and linking. Therefore, it is decided that developing our own environment is the right choice.

The designed and implemented object-oriented database language is strongly typed. A user programming within the prototype cannot only define methods for user-defined classes, but also write programs for manipulating classes.

4.3.2 Basic Constructs in the Language

The designed and implemented object-oriented database language supports the following constructs :

- expressions
- assignment statement
- conditional constructs
- looping constructs
- declarations
- blocking
- return statement

These constructs are used to develop programs and methods for the database. The user can modify or retrieve the instances of classes in the system or perform any other programming activity by writing programs. This section explains the basic constructs of the database language of ODS at an introductory level for understanding the run-time environment and the query language. The formal grammar of the language can be found in [34].

Expressions

An expression is a sequence of characters that describes an object called the value of the expression. Expressions in ODS database language are used to invoke operations on objects and structure and manipulate values. Every expression has a value which is typed. Therefore every expression is also typed. There are six types of expressions in database language of ODS :

1) Literals

They describe certain constant objects such as numbers, characters or character strings.

2) Variable Expressions

They describe the accessible variables.

3) Message Expressions

They describe messages to receivers. The value of a message expression is determined by the method that the message invokes.

4) Arithmetic Expressions

They contain any number of arithmetic operators and numeric literals and variable names. They may also contain message expressions.

5) Relational Expressions

Relational expressions contain arithmetic expressions and relational operators. Any non-zero value is considered TRUE; FALSE otherwise.

6) Logical Expressions

They connect relational expressions via logical operators and evaluate to either TRUE or FALSE.

Literals

Five kinds of objects can be referred to by literal expressions. Since the value of a literal expression is always the same object, these expressions are also called literal constants. The five kinds of literals are :

1) Numbers

Currently only integer numbers are supported in the ODS database language. The literal representation of a number is a sequence of digits that may be preceded by a minus sign, for example,

- 5
- -3

2) Characters

Characters are objects that represent the individual symbols of an alphabet. A character literal expression consists of a character enclosed in single quotes, for example,

- 'A'
- '\$'
- 'a'

3) Strings

Strings are objects that represent sequences of characters. The literal representation of a string is a sequence of characters delimited by double quotes, for example,

- "Any questions ?"
- "ODS System "

4) Class Names

Class names represent the class objects in any expression. The literal representation of a class name is a sequence of capital letters, for example,

- PERSON
- INTEGER

5) Block Expressions

A block expression contains any valid arithmetic, relational, logical or message expression. It is delimited by square brackets, and it may take a value of any type depending on the expression, for example,

- [i + 1]
- [person getage() > 30]

Variables

A variable name is a simple identifier, a sequence of letters and/or digits beginning with a letter, for example,

- person
- Count
- student1a

There are three kinds of variables that are available for a method. The instance variables and temporary variables are required to have lower-case initial letters, class variables are required to have upper-case initial letters.

1) Instance Variables

They exist for the lifetime of an object.

2) Temporary Variables

They are created for a specific activity and are available for the duration of that activity.

3) Class Variables

These are shared by all instances of a class and by its subclasses unless overridden.

One can only use temporary variables in a program. Since, programs are not executed due to a message call there is no object, *O*, that the instance variables of *O* or class variables of the class of *O* can be used. Since the ODS database language is strongly typed, all the variables, i.e. instance variables, temporary variables and class variables are typed.

Current values of instance variables of an object represent the object's current state. An object has one variable corresponding to each instance variable name in its class definition [13].

Instance variables are also typed and may take values compatible with their types. The domain of an instance variable can be any class that is defined in the class hierarchy. It may either be a user-defined class or a system-defined class. Thus, the construction of nested objects is allowed.

When a new instance is created by sending the message *New* to a class, a new set of locations for instance variables is created. The default *New* message in the definition of the *OBJECT* class initializes all the instance variables to *NIL*. But each class can define its own *New* method to initialize its instances appropriately.

The ODS database language allows programmers to declare variables local to a program or a method. Instance variables represent the current state of an object while temporary variables represent a transition state to carry out some activity [13]. Temporary variables are created whenever a message invokes a method or a user explicitly states a program execution and they are discarded at the end of the execution of the program or the method.

Each class may define zero or more class variables which are accessible by its instances. The value of a class variable is independent of which instance is using the method in which the name of the class variable appears. On the other hand, the value of instance variables and temporary variables depend on the instance that receives the message.

Message Expressions

In ODS, sending a message involves :

- 1) Identifying the object to which the message is sent, the receiver object
- 2) Identifying the arguments of the message
- 3) Specifying the desired operation to be performed via a message selector
- 4) Accepting the single object that is returned as the message answer.

The formal syntax of a message expression is as follows :

<receiverObject>

< *messageSelector*₁ > (< *arg*₁₁ > , ... , < *arg*_{1n} >)

·
·

< *messageSelector*_n > (< *arg*_{n1} > , ... , < *arg*_{nn} >)

The receiver object may be referenced either by a variable or a class constant which denotes its corresponding class object. The message selector is a simple identifier. However, an instance message selector starts with a lower-case letter whereas a class message selector starts with a capital letter. Each message can have several arguments which can be any valid expression defined in the ODS database language. The result of the first message becomes the receiver object of the second message and this goes on

until the last message. The result of the last message becomes the result of the whole message expression.

Here are some example message expressions in ODS database language :

- Get the age of a person. `person getage()`.
- Change the name of a company that the person works.
`person getcompany() setname(STRING Read("Name : "))`.
- Retrieve all person names that are above 25 years old.
`PERSON Getset() retrieve(%p, [%p getage() > 25]
[%p getname()])`.

Arithmetic Expressions

Arithmetic expressions can contain constants, variables, messages and arithmetic operators. Valid arithmetic operators are,

- plus (+)
- minus (-)
- divide (/)
- unary minus (-)
- multiply (*)

The operands of these operators are limited to be of type INTEGER. Any other type of object is not acceptable and causes an error. For example,

- `i + 1`
- `person getage() + 1`

Relational Expressions

Relational expressions are used to compare integer values via relational operators. They evaluate to either true or false. Valid relational operators are,

- Greater ($>$)
- Greater than or equal to ($>=$)
- Less than ($<$)
- Less than or equal to ($<=$)
- Equal ($=$)

The operands of a relational operator are limited to be of type INTEGER. The operands can be any valid arithmetic expressions. Any value other than zero is considered as TRUE, zero is considered to be FALSE. Therefore any valid arithmetic expression can be used in place of a relational expression. For example,

- 0
- 1
- person ismale().
- $i + 1$
- $i + 1 >= 100$

are all relational expressions.

Logical Expressions

It is possible to express more complex conditions through the use of logical operators. The operand(s) of a logical expression can be any relational expression(s). Valid Boolean operators are,

- Boolean And (and)
- Boolean Or (or)
- Boolean Not (not)

Some example logical expressions are :

- $i + j > 100$ and $j < 20$
- $\text{not} (i > j)$

The Assignment Statement

Like all other programming languages, the ODS database language has an assignment statement. One can change the value of a temporary variable, an instance variable or a class variable using the assignment statement. The lefthand side of the assignment operator (:=) contains a variable name. Its righthand side contains an expression. For example,

- `person := PERSON New().`
- `i := j.`
- `i := i + 1.`

The Conditional Constructs

The language currently supports only the `if_then_else` construct. The `if_then_else` statement is the same as the ones in conventional programming languages and it has no type. It is used to control the flow of execution in a method or program. The condition part of this statement contains a valid relational expression. For example,

```
if person ismale() then
    malecnt := malecnt + 1.
else femalecnt := femalecnt + 1.
```

The Looping Constructs

Currently, only the while statement is supported in the language. It is used to repeat a block of statements as long as the condition holds. The condition part can be any valid relational expression. For example,

```
while i < 10 do
    i := i + 1.
```

Declarations

The language allows a user to declare local variables in a program or method. The variables are also typed. For example,

```
INTEGER i.
PERSON person.
```

The Blocking Construct

It is possible to block several statements together between the keywords BEGIN and END. They are very useful in if.then.else and while statements. For example,

```
begin
    i := i + 1.
    j := j + 1.
end.
```

The Return Statement

This statement is very similar to the return statement in the C programming language. It can be used in a method but cannot be used in programs. Since each message returns an object as its result, each method contains a return

statement. The argument of the return statement can be any valid expression.
For example,

- `return(1).`
- `return("OK").`
- `return(self.getage() + 1).`

4.4 The Run-Time Environment

4.4.1 The Necessary Structures for the Run-Time Environment

This section describes the data structures associated with the run-time environment and its submodules. The following structures are needed to perform methods and programs [18] :

- Symbol Table
- Reference Table
- Activation Record
- Method Return Structure
- Expression Evaluation Stack
- Environment Stack

The Symbol Table

The compiler of the ODS database language creates a symbol table for each method or program compiled. The symbol table contains an entry for each literal or local variable that appears in a method or program.

The symbol table contains the following information :

- name of a literal or local variable
- type of a literal or local variable
- maximum length of a local variable if it is an indexed type
- type of the elements of an indexed variable
- a flag to determine redundant variable declarations.

- oop of the literal object or the object referenced by the variable

The Reference Table

A reference table is created by the system for each method/program. The reference table contains an entry for each instance/class variable and instance/class message selector that appears in the corresponding program or method. A reference table entry contains the following information :

- name of a variable or message
- type of a variable or message, the value of a variable/message type can be one of the following :
 - INSTANCE.MESSAGE
 - CLASS.MESSAGE
 - INSTANCE.VARIABLE
 - CLASS.VARIABLE

The Activation Record

The activation record is used to implement message passing and each method or program is represented by an activation record. An activation record contains the following information:

- a program counter
- a pointer to the beginning of the executable code, each entry in the executable code contains an integer code and a line number of the corresponding source code
- a pointer to the symbol table of the method/program
- a pointer to the reference table of the method/program
- a pointer to the method return structure

- name of the message that caused the creation of the activation record
- the oop of the class object of the method
- the oop of the receiver object

The Method Return Structure

Each method returns a method return structure at the end of its execution. The structure contains the following information :

- an error flag
- a flag indicating whether the method returned a value or not
- oop of the returned object

The Expression Evaluation Stack

The expression evaluation stack is used to evaluate expressions, namely, arithmetic expressions, relational expressions, logical expressions and message expressions. All expressions are converted into their equivalent postfix form in order to facilitate expression evaluation by use of a stack.

The Environment Stack

Each program execution request or a message call is represented by an activation record. Each time a message call is issued, the executor creates and initializes a new activation record and pushes it into the environment stack. At the end of the execution of a program or a method its corresponding activation record is popped from the environment stack. The use of the environment stack and activation records solves the return address handling problem and allows recursion.

4.4.2 The Executor Module

The ODS programs are compiled into an intermediate language. This language has a set of primitive code blocks, such as expressions, if_then_else and

while_wend blocks. Theoretically, users can write code in this language too, but they will not be as comfortable as using the compiler provided. The job of the executor is to scan the intermediate code and execute machine code to make the language perform. Actually, this situation is very similar to a microprocessor scanning machine instructions and executing microinstructions. In addition, the Executor has also a program counter for each executable block which is the equivalent of the instruction pointer of the microprocessor [12].

The executor module is a submodule of the run-time environment of ODS. The operations specified in the code generated by the compiler are performed by the executor module. The message passing operation is also performed through this module.

The executor module depends heavily on other modules of the run-time environment :

- the message passing module
- the expression evaluation module
- the object memory module

Each of the above modules is designed to cooperate with the executor module and some of them have several submodules which will be discussed in detail.

The executor is designed to execute a compiled code which is the output of the compilation of either a method or a program. The executor module is invoked by the user interface module whenever the RUN button is pressed after a program is selected for execution. The RUN button is in the programming shell window. The user interface module passes the filename of the program that is to be executed by the executor module. First, the necessary files related with the program to be executed are tested for existence. If any of these files is missing, the executor rejects the request by issuing an error message.

The executor expects the presence of the following files in order to execute a program or method in C :

- C.com : Contains the integer codes that are generated by the compiler.
- C.sym : Contains the symbol table entries of the symbolic code in C.
- C.ref : Contains the reference table entries of the symbolic code in C.
- Cblock1 : Contains the code generated for the expression in the first block if any. These files are numbered from 1 to n, where n is the number of block expressions in the file.

To solve the return address handling problem, the Executor uses a structure called an activation record [1]. Each time a program is to be executed, the Executor creates and initializes an activation record of the currently executing program or method. When a program is to be executed, the environment stack is initially always empty. The program may cause new activation records to be created due to message calls in the program and they are also pushed into the environment stack.

Since the introduction of the Algol 60 programming language, a run-time stack with activation records has been the primary mechanism to implement block structured programming languages. An activation record represents the local variable storage associated with the activation of a procedure. It also contains some control information that is used to access a global variable or to return to the calling environment. Since most of the modern programming languages enforce the following semantic requirements, a simple LIFO (Last In First Out) stack is sufficient for managing the activation records :

- Return from a procedure P can not occur until all the procedures invoked from P have returned.
- Pointers to data stored in activation records are not allowed.

The management of a LIFO stack is quite efficient for the implementation of an activation record stack, in ODS it is called the environment stack. Recursion is possible due to the use of activation records [39].

The activation record of a program is initialized as follows. The program counter is set to zero. Executable_code, symbol_table and reference_table entries are initialized using F.com, F.sym, F.ref files respectively. The rest of the fields of an activation record are used in case of message calls which will be discussed in the message passing module.

The code generator creates an object-oriented pointer for each character or integer literal and puts them to the corresponding symbol table entry. However, it is not possible to allocate an oop for literals of type string, class and block expression at compile time. The code generator allocates a symbol table entry for each occurrence of these literals by filling the name and type fields of the entry, and leaving the oop field undefined. For such symbol table entries, the code generator puts the literals into the name field and constant types such as stringcons, blockcons and classcons into the type field.

After each activation record creation, the executor module scans the symbol table to see if there is such a constant which will be converted to a temporary object during a specific task initiated by a program execute request or a message call. These constants are converted to temporary objects because the literals are available in the code as long as the code exists.

For each constant type, the executor creates a temporary instance of the corresponding class and assigns their oops to the oop fields of the symbol table entries. This allows the uniform treatment of string objects and string literals, class objects and class literals. The literals do not have any special treatment.

4.4.3 The Expression Evaluation Module

The primitive operations used in expression evaluation are variable value retrieval, operator application and message passing.

Variable Value Retrieval Primitive

Since there are three kinds of variables in the language, this operation can also be subdivided into three. The temporary variables are defined in

the symbol table related to a method or a program in which the variable appears. When the `SYMBOL_TABLE` code is fetched, the expression evaluation module gets the index of the variable from the next entry in the executable code and retrieves the oop field of the symbol table entry specified by this index. The value retrieved is pushed into the expression evaluation stack.

The same steps apply to the processing of literal references, because they are also put into the symbol table and referenced by the `SYMBOL_TABLE` code.

All undeclared variable names and their types are entered into the reference table. These are either instance variable names or class variable names. All message names of type class or instance are also put into the reference table. The code generator outputs the `REFERENCE_TABLE` code and an index for each reference to these variable and message names into the executable code.

The expression evaluation module invokes the appropriate routines of the object memory module to access the class/instance variables when it fetches the `REFERENCE_TABLE` code and the index. Since class and instance variables are subject to inheritance, the object memory module performs the necessary tasks related with this issue as discussed in section 4.5.

At this stage an existence test is performed for each reference to a class or instance variable. In case of non-existence the execution terminates with a proper error message.

Operator Application Primitive

When the `ARITHMETIC_OPERATOR` or `RELATIONAL_OPERATOR` code is encountered in the executable code, the Executor performs the operator specified by the next code in the executable code. If the operator is a unary operator such as unary minus or logical not, only one operand is popped from the expression evaluation stack and the value obtained by the application of the operator is pushed onto the stack. If the operator is binary, two operands are popped and the result is pushed onto the stack. The expression evaluation module assumes that the code generated for any

expression is in postfix notation.

The operands of both relational and arithmetic operators are limited to type INTEGER. However, other classes such as CHAR or STRING may implement their own methods to compare their instances.

For example, let's examine how the following arithmetic expression is executed :

$$a * (b + c)$$

Assume that a, b and c are temporary variables. Then, for each temporary variable, there is a corresponding entry in the symbol table of the method/program where these variables are declared. Executable code segment and symbol table entries for the above expression and variables are shown in Figure 4.11.

Since the expression contains only temporary variables, the code contains only references to the symbol table which is specified by SYMBOL_TABLE code. The arithmetic operator codes are preceded by ARITHMETIC_OPERATOR code. After the executor recognizes an arithmetic expression, it calls the expression evaluation module and assumes that the result of the expression is put onto the expression evaluation stack.

The arithmetic expression evaluation module starts scanning the executable code and performs the primitives under discussion. In this particular example expression, the oop fields of the symbol table entries indexed by 1, 2 and 3 are pushed into the expression evaluation stack in the order they are referenced in the executable code. Fetching the ARITHMETIC_OPERATOR causes the specified operation to be performed. When the evaluator module encounters the first operation the stack contains 5, 2 and 3 where 3 is on the top of the stack. First, PLUS operator is applied by popping two operands and pushing the result 5. Then, the TIMES operator causes 5 and 5 to be multiplied and 25 to be pushed on the stack. The EXPRESSION_END code informs the evaluator module that the job is finished.

Note that each time an arithmetic operator is applied, the types of the

ARITHMETIC_EXPRESSION
SYMBOL_TABLE
1
SYMBOL_TABLE
2
SYMBOL_TABLE
3
ARITHMETIC_OPERATOR
PLUS
ARITHMETIC_OPERATOR
TIMES
EXPRESSION_END

Executable Code Fragment

	NAME	TYPE	OOP
1	a	INTEGER	11
2	b	INTEGER	5
3	c	INTEGER	7

Symbol Table Entries

Figure 4.11: An Expression Code

operands are tested and if they are not of `INTEGER` type an error message is generated and the execution terminates.

Message Passing Primitive

The expression evaluation module calls the message passing module whenever it encounters `MESSAGE_BEGIN` code in the executable code. This module pops the destination object from the expression evaluation stack and executes the corresponding method. Each message call returns a value which is pushed again onto the expression evaluation stack through the method return structure.

In ODS, each method returns a value which can be used in an arithmetic expression like the value of a variable or a constant. The value returned can be ignored whenever required.

Since the message passing module pushes onto the stack the return value which comes from the method that corresponds to the message being serviced, the message passing primitive is very similar to the variable value retrieval primitive. The arithmetic expressions in ODS can contain any valid combination of operators, messages, variables and constants.

The following expression which contains a message and an integer constant is represented in integer codes as shown in Figure 4.12.

```
person getage() + 2.
```

Note the constant 2 in the expression is also inserted into the symbol table. The name of the message, here `getage`, is put into the reference table and because it is an instance message its type is set as `INSTANCE_MESSAGE` as in Figure 4.12. The reference to the reference table is made by `REFERENCE_TABLE` code.

As we discussed earlier, the expression evaluation module has three primitives and when it is invoked it starts performing these three primitives. In the example in Figure 4.12, first the value of the `person` variable is pushed into the expression evaluation stack. The value is retrieved from the oop

EXPRESSION-BEGIN
SYMBOL.TABLE
1
MESSAGE-BEGIN
REFERENCE.TABLE
1
0
MESSAGE-END
SYMBOL.TABLE
2
ARITHMETIC.OPERATOR
PLUS
EXPRESSION-END

Executable Code Fragment

	NAME	TYPE	OOP
1	person	PERSON	1200
2		INTEGER	2

Symbol Table Entries

	NAME	TYPE
1	getage	INSTANCE.MESSAGE

Reference Table Entry

Figure 4.12: An Arithmetic Expression Code

field of the first entry of the symbol table. Then, as MESSAGE_BEGIN code will be fetched, the message passing primitive will be performed. Next, the constant 2 will be pushed into the stack. The oop contained in the person variable will receive the message and the value returned from the method will be one of the operands of the PLUS operator.

4.4.4 The Message Passing Module

This module is called from the expression evaluation module whenever a MESSAGE_BEGIN code is fetched from the executable code. The message passing module assumes that the oop of the object which is the receiver of the message is on top of the expression evaluation stack.

First, the oop of the receiver object is popped from the stack and its owner class is determined. Then, the name of the message and its type is retrieved from the reference table entry that is specified by the reference table index. At this point a conflict test is performed to see if a class message is received by an instance object or vice versa. If such a case occurs the execution terminates with a proper error message. Next, the message passing module requests the corresponding method definition table entry (method definition) from the object memory module by specifying the class that is the owner of the receiver object, the name of the message and the type of the message, i.e. instance or class message. However, if the super flag is set by the expression evaluation module, the class that is the superclass of the class which is the owner of the receiver object is passed to the object memory module instead of the owner class of the receiver object. In order to find the superclass of the owner class of the receiver object, the message passing module calls the object memory module.

If the requested method is not found, the object memory module informs the message passing module about this fact, returns the definition of the method corresponding to the message otherwise. In case of non-existence of the method, the execution terminates.

Then, the module compares the number of arguments in the method definition with the number of arguments actually used by the user which is the fourth argument in a MESSAGE_BEGIN ... MESSAGE_END block. If these do not match, the execution is terminated and an error message is issued.

If the number of actual arguments, and the number of arguments in the method definition matches, the module tests the flag in the method definition which indicates whether the method is written in C or in the ODS database language. If the method is implemented by a C function, the arguments are pushed into the argument passing stack. In the ODS database language, an argument of a message can be any valid expression which may be a simple variable or a constant, a sequence of messages and can contain arithmetic operators or a combination of all of these. For each argument, the code generator generates an EXPRESSION_BEGIN ... EXPRESSION_END block and the message passing module calls the expression evaluation module to obtain the value of each argument. The argument expressions can contain messages which results in a new call to the message passing module. Notice the recursive nature of the expression evaluation module and message passing module through one another. The expression evaluation module invokes the message passing module which later may cause the invocation of the expression evaluation module, and this may continue like this.

All the system-defined methods are C functions written in the C programming language and they are expected to pop the correct number of arguments in a correct order. Otherwise, the argument stack becomes inconsistent and the system might fail. The system-defined methods are built-in in ODS and they are tested for such inconsistencies. The user can not add new C function methods to the system during run-time. Because this requires a dynamic linking facility which is very much dependent on the operating system environment and a highly complex task. It also reduces the compatibility of the system with other environments.

If another C function method is to be added to the existing system-defined classes or a new system-defined class is to be added to the system, the whole system must be compiled and linked after a few minor additions to the source

code.

The message passing module evaluates each argument expression, determines the type of the result of the expression and compares it with the type of the formal argument which is defined in the method definition. Any mismatch terminates the execution with an error message.

The function pointer field of the method definition contains a pointer to a C function which is executed for each message call. The message passing module calls this function through its pointer in the function pointer field of the method definition. If an error is detected during the execution of a C function, it informs the message passing module to terminate the execution. Otherwise, the value returned from the C function is pushed onto the expression evaluation stack.

If the method is implemented through the ODS database language, the process of parameter passing and method execution is different. The message passing module does not create an activation record for the C function methods. However, for each message call whose method is implemented in the ODS database language, an activation record is prepared and pushed into the stack. As in the case of an activation record created for a program execution, the program counter is set to zero, executable code, symbol table, and reference table fields of the activation record are initialized from their corresponding files. The number of arguments and the values of the argument expressions are placed in the argument count and parameter begin fields, respectively. The message selector which caused the creation of a new activation record is placed in the message name field. The oop of the receiver object and the oop of its class are placed in the oop and coop fields, respectively.

The number of arguments and their types which are obtained after the evaluation of the argument expressions are tested with the information in the corresponding method definition structure. Any conflict results in the termination of the execution and an error message is printed.

The message passing module pushes the newly created activation record

into the environment stack and it becomes the current activation record. Then, the executor module is called to execute the statements of the method. Therefore, the executor module is recursive through the message passing module. On the other hand, the message passing module is also recursive, because the executor module might call the message passing module again.

The code generated for the following message expression which modifies the location of the company that the person works through a read panel is shown in Figure 4.13.

```
person getcompany() setlocation(STRING Read("Location : ")).
```

In the example above, the messages are cascaded and the output of the getcompany message becomes the receiver object of the setlocation message. The output of the setlocation message is ignored. The parameters of the messages can be any valid expressions which may also contain messages.

4.4.5 The Object Memory Module

The object memory module provides an interface to the objects in the system [18]. Each object is associated with a unique identifier (oop - object-oriented pointer) as discussed earlier. All the run-time modules communicate about objects through their oops.

Object Memory Data Structures

Object memory uses an object table to map the oops of the objects into their physical locations in the main memory. Each reference to an object is indirected through the object table to find its physical location [18]. This indirection allows objects to be moved freely in memory without modifying the references. The object table contains the following information :

- object-oriented pointer
- status of the object (whether the object is deleted or not)

EXPRESSION-BEGIN
SYMBOL-TABLE
1
MESSAGE-BEGIN
REFERENCE-TABLE
1
0
MESSAGE-END
MESSAGE-BEGIN
REFERENCE-TABLE
2
1
EXPRESSION-BEGIN
SYMBOL-TABLE
2
MESSAGE-BEGIN
REFERENCE-TABLE
3
1
SYMBOL-TABLE
3
MESSAGE-END
EXPRESSION-END
MESSAGE-END
EXPRESSION-END

Executable Code Fragment

	NAME	TYPE	OOP
1	person	PERSON	1500
2	STRING	classcons	
3	location	stringcons	

Symbol Table Entries

	NAME	TYPE
1	getcompany	INSTANCE.METHOD
2	setlocation	INSTANCE.METHOD
3	Read	CLASS.METHOD

Reference Table Entries

Figure 4.13: A Message Expression Code

- reference count shows the number other objects that refer to the object
- physical memory address of the object

The object memory is also responsible for the maintenance and update of the class hierarchy which is used to determine the superclass of a class, the subclasses of a class and the siblings of a class. The class hierarchy is maintained as a tree and each node in the tree has the following information :

- class oop of the class that the node represents
- a pointer to the parent node which represents the superclass of the current node
- a pointer to a list of sibling nodes of the current node
- a pointer to a list of child nodes which represents the subclasses of the current class

Functions of the Object Memory Module

The object memory module provides the following fundamental functions to the run-time modules :

- Determine an object's class (type) and size
- Access and change the value of an instance variable
- Access the type of an instance variable
- Access and change the value of a class variable
- Access the type of a class variable
- Access the definition of an instance/class method
- Create a new object

Determine an Object's Class

Since type checking is performed during run-time, the executor module, the expression evaluation module and the message passing module frequently require the determination of the type of an object. The object memory performs several tests to determine an object's type, i.e. its class. First, the oop of the object is tested to be an odd integer which means it is of type INTEGER. Then, the oop is tested to be an integer in the range of 0 and 510 which means that the object is of type CHAR. If neither of these tests succeeds, the object is tested to be a class object, which means that it is of type CLASS. If all these tests fail, the object memory determines the type of the object by looking its second field. Note that all the objects other than CHAR, INTEGER and CLASS objects, store their class's oop in their second field.

Access and Modify the Value of an Instance Variable

The object memory is responsible for accessing and modifying the value of an instance variable of an object. To perform this task the name of the instance variable and the oop of the object for which the particular value of the instance variable will be manipulated are required.

The object memory first tests if the mentioned instance variable exists. If it does not exist an error status is returned to the caller. Since instance variables are subject to inheritance, the class of the object and the superclasses are searched for the instance variable under consideration. The superclasses are determined using the class hierarchy structure. After determining the class which defines the instance variable, the chunk of the object which corresponds to that class and the offset of the instance variable in that chunk are established. Finally, the value of the instance variable is either retrieved or modified.

Access the Type of an Instance Variable

The object memory provides access to the type of an instance variable. The calling module passes the name of the instance variable and the class oop of the class which should define or inherit the instance variable. The type of

the instance variable is returned after possibly examining the superclasses of the class which is passed as an argument.

Access and Change the Value of a Class Variable

The value of a class variable might be modified or retrieved through the object memory module. The name of the class variable and the oop of the class that defines or inherits the class variable should be passed. The superclass chain of the specified class might be examined if the variable is not defined in the specified class. If the module cannot find the variable definition, it informs the caller about this situation. Since space for the class variables are together with their class definitions, this process does not involve dealing with chunks of objects.

Access the Type of a Class Variable

This is very similar to the class variable value access and modification process discussed above. The type of the class variable is returned after possibly examining superclasses of the specified class.

Access the Definition of an Instance/Class Method

The message passing module requests the definition of a method that corresponds to a message selector for each message call. The message passing module passes the oop of the class of the receiver object, the name of the message selector and the message type (class message or instance message) to the object memory module. First, the MDT of the receiver's class is searched for an entry with a matching message selector. If none is found, the MDT of the class's superclass is searched next. The search continues upwards along the superclass chain [13] of the specified class until a matching method is found. If there is no method matching the selector, the message passing module is so informed.

Create a New Object

Creation of an instance of a class is also performed by the object memory module. Each system-defined class implements its own New method that

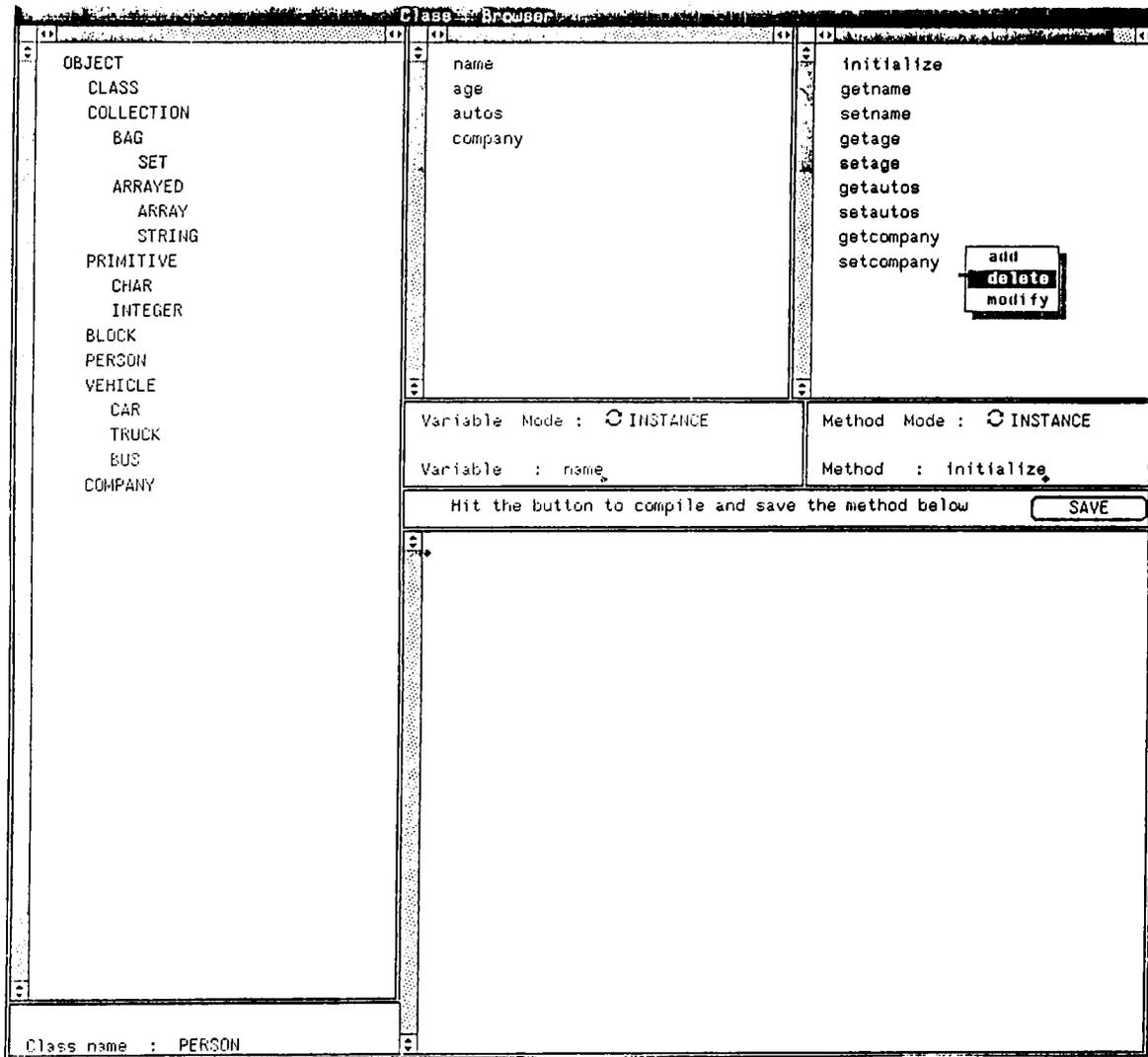


Figure 4.14: The Class Browser

creates the instances of that particular class. The New class method defined in the OBJECT class is implemented for the creation of the instances of user-defined classes. The method New allocates space for the instance of the class, creates an object-pointer for the object just created, inserts it to the object table and inserts the oop to the class's class representing set.

4.5 The User Interface

In this section, the user interface of ODS will be presented [34, 35]. There are two basic modules which are related to the user interface : the Class Browser and the Programming Shell. The user interface is supported through pop-up menus, icons, and windows which are facilities provided by the SUN Windows System [33].

4.5.1 The Class Browser

Since, the database language of ODS does not include data definition capabilities, the Class Browser is responsible for the definition of new classes and modification of existing ones. It is possible to add a class, add or delete an instance/class variable, modify an instance/class variable, add or delete a class/instance method, modify a class/instance method through the Class Browser. The Class Browser is shown in Figure 4.14.

4.5.2 The Programming Shell

The Programming Shell allows programs to be edited, compiled, listed, and executed. The Class Browser is also initiated from the Programming Shell window. Part of the Programming Shell window is a UNIX C Shell. Therefore, one can initiate other programs, communicate with other users in the system, and compile programs in this shell. The Programming Shell is shown in Figure 4.15.

The functions of the Programming Shell are initiated through selecting their corresponding buttons. When the RUN button is selected, the executor module is invoked and the run-time window appears as shown in Figure 4.16. The CONTINUE button in the run-time window is used to start the execution. The same program can be reexecuted after each execution by selecting this button again. The QUIT button returns back to the Programming Shell.

The output of programs or methods are directed to the run-time window.

```
PROGRAMMING SHELL
LIST  EDIT  EDIT NEW  DELETE  COMPILE  RUN  CLASS EDIT  QUIT
<101>ls -l *.prog
-rw-r--r-- 1 yengul 235 Jun 13 11:31 company.prog
-rw-r--r-- 1 yengul  8 Jun 28 12:36 dataentry.prog
-rw-r--r-- 1 yengul 411 May 22 13:29 dene1.prog
-rw-r--r-- 1 yengul 361 May 25 12:27 dene2.prog
-rw-r--r-- 1 yengul 352 Jun 13 11:30 person.prog
-rw-r--r-- 1 yengul 158 May 31 12:29 query1.prog
-rw-r--r-- 1 yengul  76 May  4 13:38 query2.prog
-rw-r--r-- 1 yengul 157 May  4 13:38 query3.prog
-rw-r--r-- 1 yengul 241 Jun 28 12:45 queryall.prog
-rw-r--r-- 1 yengul 254 Jun 14 13:26 queryany.prog
-rw-r--r-- 1 yengul  98 May 31 12:49 querycount.prog
-rw-r--r-- 1 yengul 187 Jun 14 14:30 querysum.prog
-rw-r--r-- 1 yengul  77 May 31 12:48 queryx1.prog
-rw-r--r-- 1 yengul 102 May 31 12:48 queryx2.prog
<102>
```

Figure 4.15: The Programming Shell

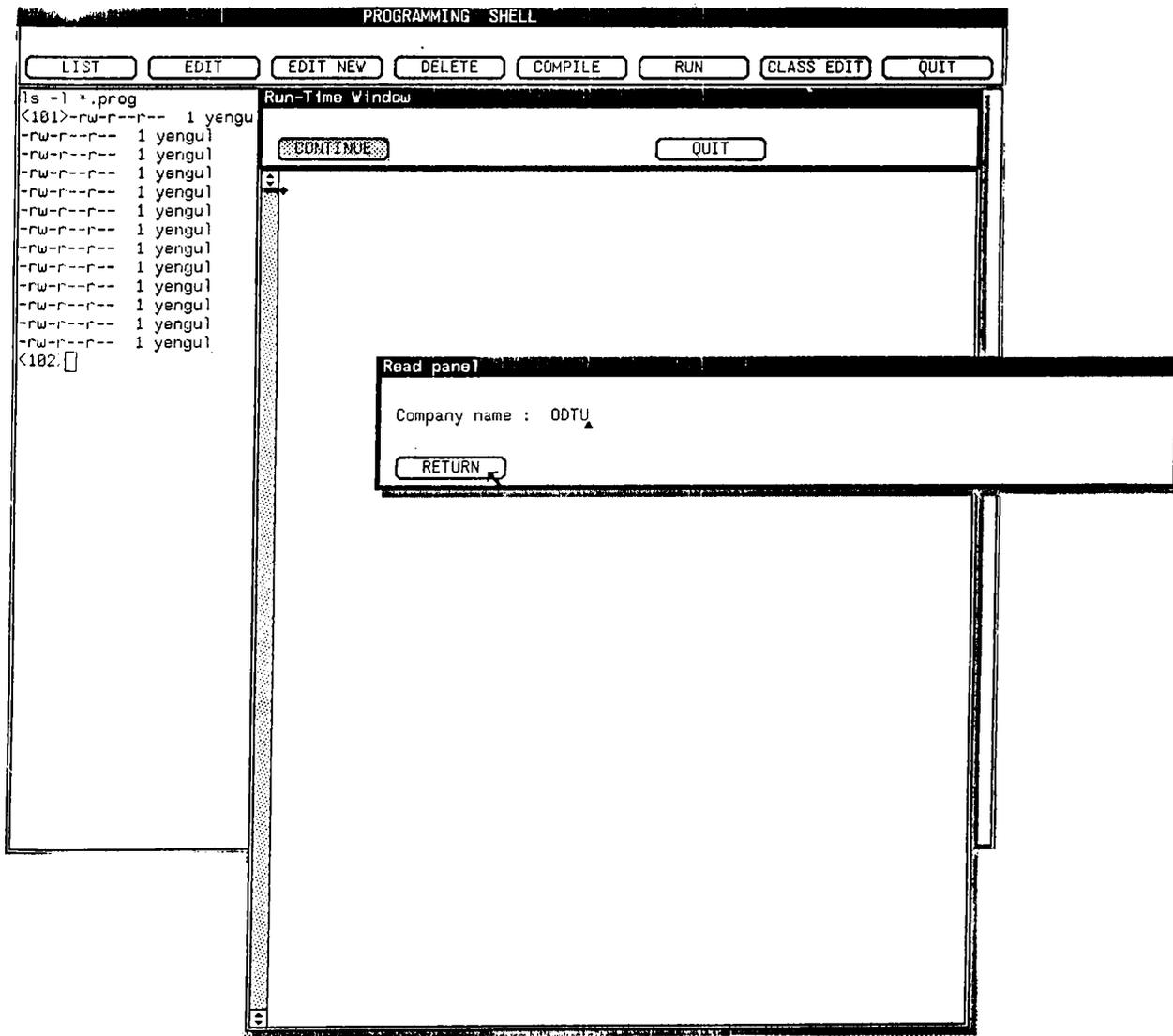


Figure 4.17: The Read Panel

On the other hand, input operations are performed through the read panel which appears for each input request as shown in Figure 4.17.

5. QUERIES IN OBJECT-ORIENTED DATABASES

Conventional object-oriented programming language systems do not support the notion of predicate-based queries. Applications must navigate from an object to others referenced by it through the object-oriented pointers of the referenced objects embedded in the former object. However, the need for predicate-based queries in large databases is obvious [5].

5.1 Object-Oriented versus Relational Queries

The process of retrieving nested objects is similar to the relational query evaluation. One can model a class as a relation and each attribute of the class becomes an attribute of the relation. Since in the object-oriented data model every object has a unique identifier we have to append a unique-identifier (UID) field to satisfy the uniqueness property of tuples. Then the retrieval of an instance of a class *C* which has a complex attribute *A* whose domain is the class *D* is similar to the relational join of a tuple of a relation *C* with a tuple of a relation *D* where the join columns are column *A* of the relation *C* and the UID column of the relation *D* [5].

Although there are similarities between object-oriented and relational query evaluation there are also some fundamental differences.

In relational databases, one can formulate a query that may involve one or more relations. The result of a query against a single relation *R*, is another relation *S* whose attributes form a subset of the attributes of the relation *R*.

On the other hand, the result of a query against say n relations, R_1, R_2, \dots, R_n is another relation S which consists of a subset of tuples of the cross product of the n relations. Again, the attributes of S form a subset of the attributes of these n relations. In relational query evaluation, logically all predicates on a single relation and all predicates which correlate columns of pairs of relations are applied to this cross product and columns are projected from the resulting set of tuples of the cross product.

In object-oriented databases, however, the result of a query involving a single class C can either be a subset of the instances in C , or a subset of the instances of another class which is the domain of an instance variable of the class C or a class that can be reached from the class C . Therefore, the query must return either one or all attributes of a class. Formally, for a class with m attributes, the query can not result with a set of objects with k attributes where $1 < k < m$ holds. This is mainly due to the fact that every object should belong to a class and the result of a query is a set of objects.

If a query which is formulated against a BAG/SET of objects whose elements are instances of a class C retrieves all the attributes of the class C , the result of the query is a subset of the objects which are the instances of the class C . If the query retrieves only one attribute of the class C , the result of the query is a set of objects which belong to the domain class of the selected attribute. Note that, for any other subset of the attributes other than any one or all attributes there is no class defined that the resulting objects can belong.

In object-oriented databases a query which involves n classes, considers n classes involved in the construction of nested objects. But, there is no mechanism for correlating the instances of these n classes on the basis of user-defined predicates between pairs of classes. In the relational model, the result of a query which requires a join of n relations is obtained by using user-defined predicates between pairs of relations. Here we conclude that there's no physical or logical links between tuples of different relations [5]. On the other hand, in object-oriented databases, a nested object is constructed through explicit logical links which are the object-oriented pointers of the

contained objects.

In a relational database, there are several independent relations, whereas an object-oriented database consists of a hierarchy of classes. This fact introduces interdependencies between classes. The first consequence of this interdependency is that a query formulated against the instances of a class *C* may either fetch only the instances of the class *C* or in addition to the instances of the class *C*, the result may include instances of subclasses of the class *C*. The second consequence is that the domain *D* of an attribute of a class is the class *D* and all subclasses of *D* [5].

5.2 The ODS Query Model

A query may be formulated against an object-oriented schema which will fetch instances of a class or elements of a collection which will satisfy a search criterion. A search criterion may be formulated using predicates on the attributes of the destination objects.

5.2.1 Predicate Construction

In conventional object-oriented languages, we do not see any predicate-based query processing functions. But in a database application, there is a great need for predicate-based queries. One may write a query to retrieve a subset of the instances of a class by specifying predicates involving the attributes of the class and/or attributes of classes that are referenced in the instances of the subject class [5]. In the literature, we see two approaches for predicate formulation:

- Path Expression Approach
- Message-Based Approach

Path Expression Approach

In the path expression approach [21, 22], a predicate is formulated using instance variables of the subject class and the variables that are contained in any class that is referenced by the subject class. For example, to retrieve the name of the company that Ali works, where Ali is an instance of the PERSON class which is the subject class here, the following path expression can be written:

Ali.company.name

First, the company instance variable of PERSON object Ali is extracted. The company instance variable contains an oop of an instance of the COMPANY class which is the referenced class from the PERSON class. Next, the name instance variable of the company object that is owned by Ali is accessed. But, the path expression approach seems to violate the data encapsulation requirements. It is stated that the attributes of any object is not visible outside the object and the only way to communicate with objects is through messages. Therefore, this approach violates the data encapsulation property of object-oriented systems [21, 22, 23, 26].

Message-Based Approach

Since messages are the only means to communicate with objects, their use in predicate formulation is desirable [5]. To retrieve or update the value of any instance variable, one has to send a corresponding message to the object. Therefore, for each instance variable that is allowed to be accessed, there has to be messages, hence methods, in the class definition of the object for retrieval and update of the value of the instance variable.

The ODS system automatically defines methods and the corresponding messages for setting and getting the values of instance and class variables. The message names are obtained by prefixing the corresponding variable names with the word get or set. One may restrict the set of variables that are accessible by the user by deleting the access methods for these variables. The methods corresponding to the name instance variable of the PERSON class were presented in Section 6. The path expression example considered

above can be rewritten using messages as follows:

```
Ali getcompany() getname()
```

First the `getcompany` message is sent to `ali` and the value of the instance variable `company` in `ali` object is retrieved. Then `getname` message is sent to the retrieved `company` object to retrieve the name of the company.

In ODS predicates are formulated using the message-based approach.

5.2.2 The ODS Query Language

All the functions of the Query Processor are implemented through messages [34]. The messages that are designed and implemented in the prototype for query formulation are

- `retrieve`
- `modify`
- `forany`
- `forall`
- `sum`
- `sumu`
- `count`
- `countu`

which operate on the collections `bag` or `set` of objects.

For each class in the system, there is a `SET` object associated with that class which contains oops of all the instances of that class. Queries are formulated against these `SET` objects, hence the classes through an indirection, and the result of each query is put into another `BAG` object in case the query

specifies a retrieval operation [22]. Queries can also be formulated against any BAG or SET object which does not correspond to any of the classes.

The formal syntax of the retrieve message is as follows:

```
receiverObject
    retrieve(iterationVariable
            selectionBlock
            projectionBlock).
```

The receiver object can be of either SET or BAG type. Iteration variable deviates from the syntax for a variable by being preceded by a % sign. This variable is used to iterate over the elements of valid collection objects.

The first block is used for selection and the second block is used for projection. For each instantiation of the iteration variable, the selectionBlock, which is a block of code that contains a relational expression, is evaluated to the value of either TRUE or FALSE. The projection block either contains the iteration variable itself or a message expression involving the iteration variable. If it contains only the iteration variable, then all the attributes of the objects which are subject to the query are projected. On the other hand, if it contains a message expression, one of the attributes of the object is projected. The result of the evaluation of a projection block is put into the resulting BAG object.

To formulate a query against the instances of a class C, one has to obtain the corresponding set object that contains its instances. This is accomplished by sending the message Getset() which is defined as a class method in the OBJECT class. All the examples in this section refer to the object-oriented database schema discussed in section 6.

Example Query 1 :

```
Retrieve the names of persons that are older than 25.
```

```
PERSON Getset() retrieve(%p
                        [%p getage() > 25]
                        [%p getname()]).
```

In the above query, %p iteration variable assumes the instances of the PERSON class one by one. For each instantiation the first block is evaluated, that is the age of the instantiated person object is retrieved and compared to 25. If the result is True, then the object is selected and the projection is applied. The second block contains the projection expression. Here the names of person objects are projected. The result of the projection operation is put into the resulting bag. Therefore, the result of this query is a bag of oops of names.

One may update or delete elements of a bag or a set using the modify message. Consider the following queries that increment the ages of all persons by one and delete persons that are older than a hundred.

Example Query 2 :

Increment the ages of all persons by one.

```
PERSON Getset() modify(%p
                        [TRUE]
                        [%p setage(%p getage() + 1)]).
```

Example Query 3 :

Delete persons older than 100.

```
PERSON Getset() modify(%p
                        [%p getage() > 100]
                        [%p remove()]).
```

Sum and count messages are similar to the select message except that they return the sum and count of the elements of the resulting bag. Sumu

and countu messages eliminate the duplicate elements in the resulting bag before proceeding.

Example Query 4 :

Get the number of persons that are older than 25.

```
PERSON Getset() count(%p
                    [%p getage() > 25] [%p]).
```

Forany and forall messages are also similar to the retrieve message but forany/forall returns True if any/all of the elements of the destination bag satisfy the relational expression block.

Example Query 5 :

Retrieve the persons that have at least a MERCEDES vehicle and work in MICROSOFT Inc.

```
PERSON Getset()
  retrieve(%p
    [%p getvehicles()
      forany(%a
        [%a getname() strcmp("MERCEDES")])
        and
        %p getcompany()
        getname() strcmp("MICROSOFT Inc")]
    [%p] ).
```

Example Query 6 :

Retrieve the names of persons who have only red vehicles.

```
PERSON Getset()
```

```
    retrieve(%p  
            [%p getvehicles()  
            forall(%a  
                  [%a getcolor() = "Red"])]  
            [%p getname()]).
```

Queries can be used anywhere in the methods where a message is appropriate. Query functions are implemented by the message passing paradigm and their corresponding definitions are in the BAG class. This solves the impedance mismatch problem between the programming and query languages [22].

5.3 Object-Oriented Query Processing in other Systems

5.3.1 GEMSTONE

Gemstone provides a limited query sublanguage. The sublanguage only supports selection on collections which are the instances of SET or BAG classes. Selection conditions are conjunctions of comparisons, where the comparisons are between path expressions and other path expressions or literals.

An associative query is a variation on a select expression. Suppose an Employee object also has a worksIn instance variable whose domain is Department class. The following query will make use of all indexes available on the paths specified in the conditions. OPAL differentiates between selection expression and associative queries by the use of brackets and braces. Braces indicate an associative query whereas brackets indicate a selection expression. This distinction introduces an impedance mismatch between OPAL and its query sublanguage [21, 22, 23].

```

Emps select :
    {anEmp | anEmp.name.last = 'Jones'
      &
      anEmp.salary >
      anEmp.worksIn.manager.salary}

```

The OPAL language allows the use of path expressions which violates the data encapsulation and independence properties of the object-oriented approach. The select message is similar to the retrieve message in ODS. However, ODS not only supports selection but also projection on objects. It also provides additional messages which are quite useful in expressing queries. However, ODS does not support indexes and therefore cannot utilize them in query processing yet.

5.3.2 ORION

The query model of the ORION [2, 3, 4, 5] is very similar to the ODS query model. In fact, ORION has affected a great deal the implementation of query processing methodology in object-oriented databases.

In ORION, one uses predicates to formulate the search criteria in queries which are expressed in terms of messages. The use of a path-expression is not allowed which violates the data encapsulation requirement. However, the ORION query language supports only selection but does not provide projection on objects. For example, to retrieve vehicles which are blue and manufactured by the Ford motor company, the formulation is as follows :

```
(Vehicle select :V
  (:V Color = "blue")
  and
  (:V Manufacturer Name
    = "Ford Motor Company")))
```

The result of the query above is a set of Vehicle objects. In addition to the select message, ORION implements some and all messages which are similar to forany and forall messages of ODS. As discussed earlier, ODS implements modify, sum, sumu, count and countu messages as well which do not have counterparts in ORION.

6. AN APPLICATION WITH ODS

In this section an example database application developed in ODS is presented. First the classes involved in the application with their instance variables and protocols are explained. Then, some application programs on the example object-oriented database schema are given to show the data manipulation and computational aspects as well as the query processing capabilities of the ODS object-oriented database language.

6.1 The Example Object-Oriented Database Schema

The example schema includes six classes, namely, PERSON, COMPANY, VEHICLE, BUS, TRUCK and CAR. The relationships between these classes are given in the example hierarchy shown in Figure 6.1. The figure also shows the instance variables of each class in the system.

In addition to the user-defined methods, each class implements methods to retrieve and modify the values of its instance variables. These methods are named by prefixing the respective instance variable names by get and set strings. Examples of these methods are given in the protocol of the PERSON class for the age instance variable only. Each class implements methods to retrieve and modify the values of its instance variables, but we omit the listings here.

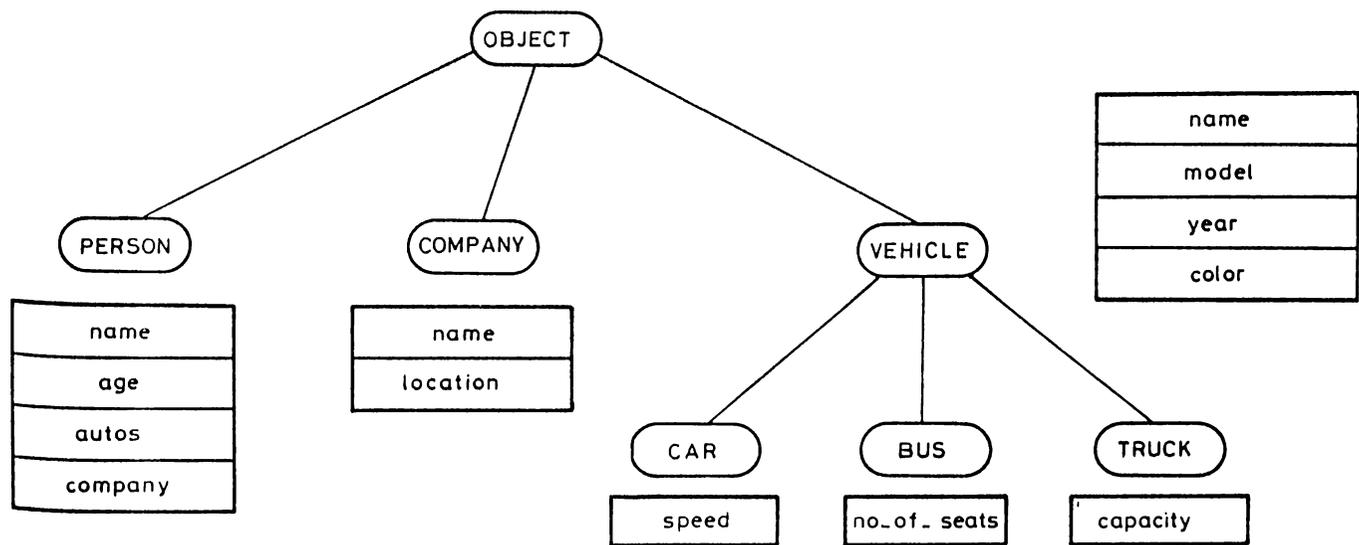


Figure 6.1: The Example Hierarchy

The Protocol of the PERSON Class :

PERSONgetagei.met : Returns the age of a person.

```

getage ()
begin
    return(age).
end
  
```

PERSONsetagei.met : Sets the age of a PERSON object.

```

setage(INTEGER s)
begin
    age := s.
    return(1).
end
  
```

PERSONinitializei.met : Initializes a PERSON object.

```
initialize()
CAR      car.
BUS      bus.
TRUCK    truck.
STRING   companyname[80].
CHAR     answer.
CHAR     vehicletype.
begin
    name := STRING Read("Name").
    age  := INTEGER Read("Age").
    answer := CHAR
                Read("Does she/he have a vehicle ? (Y/N)").
    autos := SET New('P').
    # The user may enter several vehicles #
    while (answer = 'Y') do
    begin
        vehicletype := CHAR
                        Read("(T)ruck, (C)ar, (B)us ? ").
        # A person may have several different vehicles#
        if (vehicletype = 'C') then
        begin
            car := CAR New().
            car initialize().
            autos add(car).
        end.
        else if (vehicletype = 'B') then
        begin
            bus := BUS New().
            bus initialize().
            autos add(bus).
        end.
    else
```

```

        begin
            truck := TRUCK New().
            truck initialize().
            autos add(truck).
        end.
        answer := CHAR Read("Another vehicle ? (Y/N)").
    end.
    companyname := STRING Read("Company Name : ").

    # Get the oop of the COMPANY object that the person#
    # works for#
    company :=
        COMPANY Getset()
            retrieve(%c,
                [%c getname() strcmp(companyname) = 1],
                [%c])
            first().
    return(1).
end

```

The Protocol of the COMPANY Class :

COMPANYinitializei.met : Initializes a COMPANY object by setting its name and location instance variables through the read panel.

```

initialize()
begin
    name := STRING Read("Company name : ").
    location := STRING Read("Location : ").
    return(1).
end

```

The Protocol of the VEHICLE Class :

VEHICLEinitializei.met : Initializes a VEHICLE object by setting its name, model, year and color instance variables.

```

initialize()
begin
    name := STRING Read("Name of the vehicle ? ").
    model := STRING Read("Model ? ").
    year := INTEGER Read("Year ? ").
    color := STRING Read("Color ? ").
    return(1).
end

```

The Protocol of the BUS Class :

BUSinitializei.met : Initializes a BUS object by setting its seatno instance variable. Inherited instance variables are initialized by the initialize method in the superclass of the BUS class (VEHICLE class) which is activated through the pseudo variable super.

```

initialize()
begin
    seatno := INTEGER Read("Seats : ").
    super initialize().
    return(1).
end

```

The Protocol of the CAR Class :

CARinitializei.met : Its function is similar to the initialize method in the BUS class.

```

initialize()
begin
    super initialize().
    speed := INTEGER Read("Speed : ").
    return(1).
end

```

The Protocol of the TRUCK Class :

TRUCKinitializei.met : Its function is similar to the initialize method in the BUS class.

```
initialize()
begin
    super initialize().
    capacity := INTEGER Read("Capacity ? ").
    return(1).
end
```

6.2 Example Programs

1) company.prog

The company information is accepted from the user in this program. The user is able to enter as many companies as he wants. All the company objects are listed to the run-time window at the end of the program as shown in Figure 4.16.

```
program
CHAR    answer.
COMPANY company.
begin
    answer := 'Y'.
    while (answer = 'Y') do
        begin
            company := COMPANY New().
            company initialize().
            answer := CHAR Read("Another company ? (Y/N)").
        end.
    COMPANY Getset() print(1).
end
```

2) person.prog

This program is used for entering information on a person into the database. At the end of the program all person objects are listed to the run-time window.

```
program
PERSON  p1.
INTEGER i.
CHAR    answer.
BAG     baggy.
begin
    answer := 'Y'.
    # Allows several persons to be added #
    while (answer = 'Y') do
    begin
        p1 := PERSON New().
        p1 initialize().
        answer := CHAR Read("Another person ? (Y/N)").
    end.
    PERSON Getset() print(1).
end
```

3) queryall.prog

This program tests if all persons are older than 18. The forall query message is used.

```
program
STRING  s[80].
begin
    if (PERSON Getset()
        forall(%p, [%p getage() > 18]) = 1) then
    begin
```

```

        s := "Everyone is older than 18 .".
        s print (1).
    end.
    else
        begin
            s := "There is a person below 18".
            s print(1).
        end.
    end
end

```

4) queryany.prog

Test if any of the persons is older than 100.

```

program
STRING  s[80].
begin
    if (PERSON Getset()
        forany(%p, [%p getage() > 100]) = 1) then
    begin
        s :=
            "There is at least a person older than 100".
        s print(1).
    end.
    else
        begin
            s:= "There is nobody older than 100".
            s print(1).
        end.
    end
end

```

5) querycount.prog

Retrieve the number of persons that are older than 100.

```
program
begin
PERSON Getset()
    count(%p, [%p getage() > 100], [%p getage()])
    print(1).
end
```

6) querysum.prog

Retrieve the total of ages of persons older than 10.

```
program
STRING s[80].
begin
    s := "The total of ages of people older than 10 : ".
    s print(0).
    PERSON Getset()
        sum(%p, [%p getage() > 10],
            [%p getage]).
end
```

7) queryx1.prog

Print all the person objects to the run-time window.

```
program
begin
PERSON Getset()
    retrieve(%p, [1], [%p]) print(1).
end
```

8) queryx2.prog

Print the person object whose name is Can.

```
program
begin
PERSON Getset()
    retrieve(%p, [%p getname() strcmp("Can")],
            [%p print(1)]).
end
```

7. CONCLUSIONS

The greatest strength of the object-oriented approach is that it allows the modelling of a real world entity whatever its complexity is by a single object. This increases the maintainability and understandability of complex systems such as CAD/CAM, document retrieval systems, programming languages and databases.

Most of the advantages of object-oriented systems are due to the basic characteristics of these systems which are data abstraction, homogeneity, independence, information hiding, inheritance, late binding, message passing, object identity, overloading and reusability.

Information hiding and data abstraction increases the reliability, integrity and security of the systems. The inheritance mechanism enforces reusability. The dynamic binding mechanism allows new classes to be added dynamically, thus supporting extendible data typing facilities. The ability to represent the behaviour together with the structure of an entity makes it easier to enforce the data integrity constraints.

Independence of objects appears to be an important concept. It guarantees a higher degree of security and allows an efficient data integrity enforcement, since the state of an object can only be accessed by its methods.

One of the application areas of the object-oriented approach is object-oriented database management systems. Object-oriented database management systems are shown to be suitable for applications which require the modelling of complex data. They introduce powerful concepts for modelling real world entities by narrowing the semantic gap between the data and its

representation in the database. The concept of object identity reduces the update anomalies and automatically satisfies the referential integrity, because references are only through object identifiers, but not through data values.

In spite of its advantages, there are some disadvantages of object-oriented database systems. They have relatively poor performance and a lack of a theoretical data model. We do not, for instance, have any equivalent of relational algebra.

An object-oriented database management system prototype (ODS) has been under development since 1987 at Bilkent University. ODS supports all the basic characteristics of the object-oriented approach. In this thesis an object-oriented query processor, a database language executor and the protocols for the system-defined classes have been designed and implemented.

ODS has an object-oriented database language which is strongly typed and provides data manipulation and computational facilities. The language can be extended to support a data definition facility which is currently provided by the Class Browser of ODS. All type checking operations are performed at run-time. The compiler of the language can be modified to support some of the type checking operations at compile-time for performance reasons.

ODS supports an object-oriented query model which is similar to that of ORION's but provides a projection operation in addition to a selection operation. Since the query functions are implemented using the message passing paradigm, the formulation of queries in the object-oriented database language does not lead to any impedance mismatch problems. In addition, the query model does not violate the data encapsulation principle, since it uses messages in the construction of selection and projection expressions.

ODS is developed as a single-user memory-based system. It can be modified to support multiple users. However, this introduces several concepts such as concurrency control, transactions and authorization.

ODS supports simple inheritance which could be extended to support multiple inheritance.

The system can be extended to support interactive query processing and to provide an object displayer which enhances the user friendliness of the system.

An object browser that will deal with class instances could be developed. Such a browser may be integrated with the interactive query processor and object display modules.

A secondary storage management facility is an essential part of a database management system. Currently, the ODS only stores(retrieves) objects including the class definitions at the end(beginning) of a session. Secondary storage management facilities need to be added to the system. Indexing should also be introduced to improve the performance of query processing in ODS.

REFERENCES

- [1] Aho, A.V., J.D. Ullman, *Principles of Compiler Design*, Addison Wesley, 1977.
- [2] Bancilhon, F., and, D. Maier, "Multilanguage Object-Oriented Systems: New Answer to Old Database Problems", *Programming of Future Generation Computers II*, April 1988.
- [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. et al., "Data Model Issues for Object-Oriented Applications", *ACM Transactions on Office Information Systems*, Vol.5, No.1, Jan.1987, pp.3-26.
- [5] Banerjee, J., W. Kim,, and, K. C. Kim, *Queries in Object- Oriented Databases*, MCC Technical Report, 1987
- [6] Casasis, E.,"An Object-Oriented System Implementing KNOs", *Proceedings of the Conference on Office Information Systems*, Palo Alto, March 1988, pp. 284-290
- [7] *Commands Reference Manual*, Sun Microsystems Inc., 1986.
- [8] Copeland, G., and D. Maier, "Making Smalltalk a Database System", *Proc.ACM SIGACT / SIGMOD International Conference on the Management of Data*, 1985.

- [9] Cox, Brad J., *Object-oriented Programming An Evolutionary Approach*, Addison-Wesley, 1986.
- [10] Date, C.J., *An Introduction to Database Systems*, Fourth Edition, Vol.1, Addison Wesley, 1986.
- [11] Diederich, J., and J. Milton, "Experimental Prototyping in Smalltalk", *IEEE Software*, May 1987, pp.50-64.
- [12] Duff C., "Designing an Efficient Language", *BYTE Magazine*, August 1986
- [13] Goldberg, A., and D. Robson, *Smalltalk-80 :The Language and Its Implementation*, Addison-Wesley, 1983.
- [14] 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, Jan.1987, pp.70-95.
- [15] Kaehler, T., and D. Patterson, "A Small Taste of Smalltalk", *BYTE Magazine*, August 1986, pp.145-159.
- [16] Karaorman, M., *Secondary Storage Management in an Object-Oriented Database Management System*, M.S. Thesis, Bilkent University, Ankara, July 1988.
- [17] Kernighan, B. W., D. M. Ritchie, *The C Programming Language*, Prentice Hall,1978.
- [18] Kesim, N., *An object Memory for an Object-Oriented Database Management System*, M.S. Thesis, Bilkent University, Ankara, July 1988.
- [19] Khoshafian, S.N., and G.P. Copeland, "Object Identity", *ACM OOP-SLA '86 Proceedings*, Sept.1986.
- [20] Kim , W., H. Chou, and, J. Banerjee, "Operations and Implementations of Complex Objects", *IEEE Transactions on Software Engineering*, Vol. 14, No. 7, July 1988.
- [21] Maier, D., and J. Stein, "Indexing in an Object-Oriented DBMS", *Proc. of the Workshop on Object-Oriented Database Systems*, Sept.1986.

- [22] 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.
- [23] Maier, D., and J. Stein, "Development and Implementation of an Object-Oriented DBMS", *Research Directions in Object-Oriented Programming*, Shriver, B., and P. Wegner Eds, 1987
- [24] Meyer, B., "Reusability : The Case for Object-Oriented Design", *IEEE Trans. Software Eng.*, March 1987, pp. 50-65
- [25] Nierstrasz, O.M., "What is the 'Object' in Object-Oriented Programming ?", *Objects and Things*, ed. D.Tsichritzis, Centre Universitaire D'Informatique, Universite de Geneve, March 1987, pp.1-13.
- [26] Nierstrasz, O.M., "A Survey of Object-oriented Concepts", *Active Object Environments*, ed. D.Tsichritzis, Centre Universitaire D'Informatique, Universite de Geneve, July 1988.
- [27] Özelçi, S. M., *Message Passing in an Object-Oriented Database Management System*, M.S. Thesis, Bilkent University, Ankara, July 1988.
- [28] Özelçi, S.M., N. Kesim, M. Karaorman, E. Arkun, "An Experimental Object-oriented Database Management System Prototype", *Proc. of the Third International Symposium on Computer and Information Sciences*, October 1988, Çeşme, Turkey.
- [29] Pascoe, G.A., "Elements of Object-Oriented Programming", *BYTE Magazine*, August 1986, pp.139-144.
- [30] Peterson, R., "Object-Oriented Database Design", *AI Expert*, March 1987, pp. 26-31
- [31] Stefik, M., and D.G. Bobrow, "Object-Oriented Programming: Themes and Variations", *AI Magazine*, Jan. 1986, pp. 40-62.
- [32] *Sun System Overview*, Sun Microsystems Inc., 1986.
- [33] *SunView Programmers Guide*, Sun Microsystems Inc., 1986.

- [34] Türkmen, S., *Data Definition and Manipulation Languages for an Object-Oriented Database Management System (ODS)*, M.S. Thesis, Bilkent University, Ankara, July 1989.
- [35] Türkmen, S., C. Yengül, E. Arkun, "An Object-Oriented Database System Prototype", to appear in the *Proc. of the Fourth International Symposium on Computer and Information Sciences*, October 1989, Çeşme, Turkey.
- [36] Tsicritzis, D. C., O. M. Nierstratz, "Fitting Round Objects Into Square Databases", *Proceedings ECOOP 88*, Oslo, Springer-Verlag
- [37] Ullman, J.D., *Principles of Database Systems*, Computer Science Press, 1982.
- [38] *UNIX Interface Reference Manual*, Sun Microsystems Inc., 1986
- [39] Zeigler S., N. Allegre, D. Coor, R. Johnson, J. Morris, "The Intel 432 Ada Programming Environment", *COMPSCON S'81*, 1981, pp.405-410
- [40] Zaniolo C. et al., "Object-Oriented Database Systems and Knowledge Systems", *1st International Workshop on Expert Database Systems*, 1985, pp.1-17.

A. List of Run-Time Errors

The run-time module generates several error messages to inform the user about problems. Each time an error is detected, the name of a code currently being executed, the line number of the currently executing statement and other additional information such as message or variable name together with the error message are printed. The list of the error messages are as follows :

- Invalid class message
- Invalid instance message
- No of parameters does not match
- Parameter types are not compatible
- Return statement is missing in the method body
- Duplication of an element of the SET object
- Object does not exist in the BAG/SET object
- ARRAY size too large
- STRING size too large
- Index out of range in the ARRAY object
- Index out of range in the STRING object
- String overflow in string concatenation
- Operand type is not an integer in arithmetic expression

- Type mismatch in assignment
- Instance variable not found
- Class variable not found
- Object table overflow
- Symbol table not found
- Reference table not found
- Executable code not found