MESSAGE PASSING IN 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
Sibel M. Ozelçi
July 1988

MESSAGE PASSING IN 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

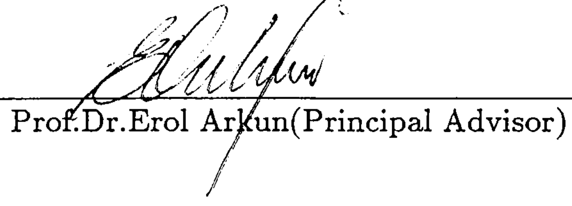
Sibel M. Özelçi

July 1988

Sibel M. Özelçi
tarafından bağışlanmıştır.

QA
76.9
.D3
O23
1988
B 1871

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.

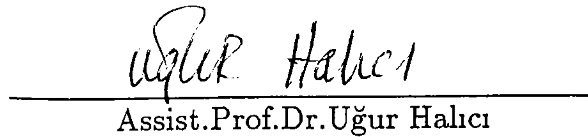


Prof.Dr.Erol Arkun(Principal Advisor)

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

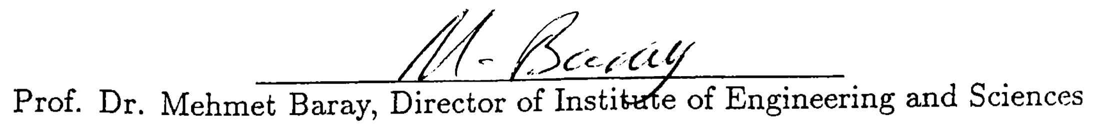
Assist.Prof.Dr.Altay Güvenir

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.



Assist.Prof.Dr.Uğur Halıcı

Approved for the Institute of Engineering and Sciences:



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

ABSTRACT

MESSAGE PASSING IN AN OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM

Sibel M. Özelçi

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

In this thesis, a focused survey on object-oriented database management systems and on object-orientation in general was carried out and a single-user object-oriented database management system prototype was designed and implemented. A command language was defined and a message passing scheme was proposed and implemented. A compiler for the language was developed.

The developed language is computationally complete and aims at solving the impedance mismatch problem. It contains both data definition and data manipulation statements. The statements can be used interactively or in the form of methods. After compilation, the statements are translated into integer codes and these codes are used to perform the necessary operations.Since the developed prototype is a single-user system, the message passing passing scheme does not provide any concurrency control mechanisms and stacks are used to implement message passing and argument handling.

Keywords : object-oriented database management systems, object, class, instance, method, message, message passing, inheritance, class hierarchy, object identity, data abstraction.

ÖZET

NESNESEL BİR VERİ TABANI SİSTEMİNDE MESAJ YOLLAMA

Sibel M. Özelçi
Bilgisayar Mühendisliği ve Enformatik Bilimleri Yüksek Lisans
Tez Yöneticisi: Prof.Dr.Erol Arkun
Temmuz 1988

Bu tez çalışmasında nesnesel yaklaşım ve nesnesel veri tabanı sistemleri üzerinde bir araştırma yapılmıştır. Ayrıca tek kullanıcılı bir nesnesel veri tabanı sistemi prototipi için bir dil geliştirilmiş ve bir mesaj yollama yöntemi önerilmiş ve uygulanmıştır. Geliştirilen dil için bir derleyici yazılmıştır.

Geliştirilen dil hesapsal açıdan tamdır ve empedans uyumsuzluğu problemini çözmeyi amaçlamaktadır. Veri tanımlama ve veri kullanımı için komutlar içerir. Komutlar doğrudan doğruya veya metodlar halinde kullanılabilirler. Derleme sırasında komutlar bazı kodlara çevrilirler ve bu kodlar daha sonra gerekli işlemleri yapmak üzere kullanılırlar. Geliştirilen sistem tek kullanılıcılı olduğundan önerilen mesaj yollama yöntemi verilere aynı anda erişimden doğan problemleri çözümleyecek mekanizmalar içermemektedir. Mesaj yollama ve parametre gönderme yığıt kullanılarak gerçekleştirilmiştir.

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

ACKNOWLEDGEMENT

I gratefully acknowledge the valuable help and advice of my supervisor Professor Erol Arkun without which this thesis could not have been completed and Dr. Nierstrasz for his helpful remarks. I would also like to thank Nihan Kesim and Murat Karaorman with whom we worked together on the project of developing an object-oriented database management system prototype for their help and cooperation and all the other research assistants at Bilkent University who assisted in the preparation of this thesis. I would especially like to thank Levent Alkışlar and Oğuz Gülseren for their support and assistance. I would also like to acknowledge the help and support of my parents and the valuable contribution of Bilkent University and the department of Computer Engineering and Information Science in providing the facilities needed for the completion of the thesis.

TABLE OF CONTENTS

1 Introduction 1

2 The Object-Oriented Approach 5

2.1 The Basic Concepts in Object-Orientation 5

2.1.1 Objects and Classes 5

2.1.2 Messages and Methods 9

2.1.3 Inheritance and the Class Lattice 11

2.1.4 Object Types 15

2.1.5 Object Identity 16

2.2 Extensions to the Basic Model 22

2.2.1 Schema Evolution 22

2.2.2 Composite Objects 24

2.2.3 Indexing 26

2.2.4 Temporal Aspects and Version Management 27

2.3 Basic Properties of the Object-Oriented Approach 30

2.3.1 Information Hiding 30

2.3.2 Data Abstraction 31

2.3.3 Data Independence 32

2.3.4 Homogeneity 32

2.3.5 Message Passing 32

2.3.6 Dynamic Binding 33

2.3.7 Inheritance 33

2.3.8 Polymorphism and Overloading 34

2.3.9 Reusability 35

2.3.10 Interactive Interfaces 35

2.3.11 Concurrency 35

2.4 Application Areas of the Object-Oriented Approach 36

2.4.1 Programming Languages 36

2.4.2 Database Management Systems 36

2.4.3 Knowledge Representation 37

2.4.4 CAD/CAM Systems 37

2.4.5 Office Information Systems 37

2.5 Object-Oriented Programming Languages and Some Examples 38

2.5.1 Smalltalk 38

2.5.2 Smallworld 44

2.6 Object-Oriented Database Management Systems and Some Examples 46

2.6.1 GemStone 46

2.6.2 ORION 49

2.6.3 IFO 54

2.7 Conventional versus Object-Oriented Database Management Systems 58

2.8 Advantages and Disadvantages of the Object-Oriented Approach 61

3 The Object-Oriented Database Management System Prototype 63

3.1 An Overview 63

3.2 The Modules of the System 66

3.2.1 Object Memory and Schema Evolution 66

3.2.2 Message Passing 69

3.2.3 Secondary Storage Management and Indexing 71

3.2.4 The User Interface 75

3.3 The Necessary Structures 75

3.3.1 Class Definition Object 76

3.3.2 Method Definition Table 80

3.3.3 Instance Access Table 81

3.3.4 Object Table 81

3.3.5 Class Hierarchy Object 81

4 The Command Language 84

4.1 Data Definition Language 86

4.2 Data Manipulation Statements 90

4.3 Some Examples 115

5 The Message Passing Scheme 122

5.1 The Lexical Analyzer 123

5.2 The Parser 124

5.3 The Code Generator 130

5.4 The Executor Module 130

5.5 The Query Processor 136

6 Open Problems and Future Extensions **137**

7 Conclusion **140**

A List of Basic Routines **142**

A.1 The Lexical Analyzer 142

A.2 The Parser 142

LIST OF FIGURES

2.1 Some example class definitions, message calls and method definitions in Smalltalk 40

2.2 The graphical representation of IFO objects, object reps . . . 55

2.3 An example fragment rep 55

2.4 The representation of specialization and generalization (IS-A) relationships 56

2.5 An extended object rep 57

3.1 The four major modules of the prototype 66

3.2 The format of an allocated object 67

3.3 The format of a class object 68

3.4 The initial class hierarchy and the system defined classes 68

3.5 The abstract view of a variable sized container 73

3.6 The abstract view of a variable sized container with external super-part 74

3.7 A class definition object 77

3.8 Storing an object as contiguous blocks of memory in a linked list 79

3.9 The storage representation of an instance object 80

3.10 The object table 81

3.11 The format of a class hierarchy object 82

3.12 An example class hierarchy and its internal representation . . 83

4.1 The organization of the three classes 116

4.2 The result of executing the *new* statement 119

4.3 The result of executing the *define* statement 119

5.1 The internal representation of production rules 127

5.2 An example for the internal representation of productions . . 128

5.3 An error entry 129

5.4 The record structure of the error file 129

5.5 The internal representation of an example method 133

5.6 The internal representation of an *if* statement 134

5.7 The internal representation of a *while* statement 135

1. INTRODUCTION

Conventional methods such as relational database management systems and programming languages lack a suitable problem solving approach to various data intensive applications such as CAD/CAM applications, office information systems, knowledge base systems, expert systems and knowledge representation. Database systems, programming languages and artificial intelligence already have overlaps in some areas. Databases require better integrated application program interfaces, expert systems must deal with large collections of base facts and programming languages need richer ways to model their data. As a result of all these needs object-oriented programming environments were developed and this approach was extended to other fields. Since then the approach has gained a great deal of importance and popularity. The basic characteristic of object-orientation is that data are active and procedures are passive unlike in traditional data processing methods. In other words, instead of data being sent to procedures, objects which represent real world entities are asked to perform operations on themselves. Every real world entity is modelled as an object.

The basic concept in object-orientation is the object which captures both the state and the behaviour of an entity. The behaviour is represented using methods and messages. Methods are performed when objects are invoked by messages. Messages specify which operation to perform on an object while a method specifies how the operation will be performed. Similar objects constitute a class while the elements of a class constitute its instances. The definitions related to the instances of a class appear only in the corresponding class definition thus eliminating the redundant specification of the information for each instance. The classes in a system can be organized in a class hierarchy or class lattice. A class can be defined as the subclass of another class inheriting the implementation and interface of its superclass. This is known as inheritance. A subclass may modify the definitions it inherits or may add new definitions to them. The class and inheritance concepts increase

modularization and reduce duplication. Building a hierarchy of objects and inheritance also facilitates top-down design.

In object-orientation, each object is referenced using a value independent and physical location independent surrogate. Surrogates provide data and location independence but unless some kind of indexing is used, one has to perform a sequential search when associative or value-based access is required. The surrogate of an object, that is, its identity remains the same regardless of changes in the object. Objects reference their components by identity and not by value. Therefore data integrity and referential integrity are automatically satisfied and data duplication is reduced.

There is no general definition of an object-oriented system. One approach is to define an object-oriented system as a system which supports data encapsulation and inheritance. Another definition is introduced considering that these two requirements are quite restrictive. This definition states that an object-oriented system is a system that supports data encapsulation and not necessarily inheritance.

Informally, an object-oriented database management system can be defined as follows: a system which is based on a data model that allows the representation of an entity, whatever its complexity and structure, by exactly one object of the database. No decomposition into simpler concepts is necessary. As entities may be composed of subentities which are entities themselves, an object-oriented data model must allow recursively composed objects.

Although record-based models have been successfully applied to a variety of data problems, they have serious limitations. Fundamental problems include the fact that in these models most relationships between data must be represented using record and pointer structures and thus force different kinds of relationships to be represented in the same way. Also, record entries must be from fixed sets of possible values, thus making it difficult to represent situations in which two or more entity types participate in a given role of a relationship. Finally, relational models rely on symbolic identifiers to represent data objects and in that way they add another level of indirection. The problems related to conventional database systems can be solved by combining object-oriented concepts and the storage management functions of a traditional database management system.

Conventional record-oriented database management systems reduce application development time and improve data sharing among applications. However they are subject to the limitations of a finite set of data types and the need to normalize data. In contrast, object-oriented systems offer flexible abstract data-typing facilities and the ability to encapsulate data and operations with the message metaphor. In addition, they reduce application development efforts. Object-oriented database management systems support more direct modelling and require less encoding compared to other data models and they capture more information semantics. Also, one can easily represent models which can not be represented using normalized relations, thus keeping the semantic gap as small as possible and representing most of the problem semantics in the database itself. Another point is that, object-oriented systems aim at solving the impedance mismatch problem seen in conventional database systems in which there are separate languages for data definition and data manipulation by providing a unified language supporting both functions. Lastly, object-oriented database systems allow nested (non-first normal form or N1NF) relations, can capture the temporal aspect of the data and can handle multiple versions.

The major advantages of the object-oriented approach are versatility, flexibility, reusability, implementation independence and increased programmer productivity. Also, since duplication and redundancy are reduced data integrity is automatically satisfied. The main disadvantages are the relatively poor performance and the complexity of implementing such a system. This is due to the lack of a theoretical model and other basic standards for object-oriented systems. In addition, object-oriented systems require a new and different approach to problem-solving.

The main problem areas of the object-oriented approach and object-oriented database management systems are version control, manipulation of composite or dependent objects, schema evolution and handling conflicts in the case of multiple inheritance. The use of object identity requires a sequential search during associative access unless some kind of mapping or indexing is provided, thus degrading system performance. Index handling in object-oriented database management systems is a very important research area. Another problem associated with the object identity concept is the preservation of object identity consistency. Other open problems related to object-oriented database management systems include garbage collection, storage management and especially the storage of variable-size or very large objects and clustering. Also, there is a great demand for a theoretical model and some standards for the object-oriented approach.

The developed object-oriented database management system prototype consists of four major modules which are object memory and schema evolution; message passing; secondary storage management, indexing and the user interface. Object memory handles the representation, access and manipulation of the objects in the system. The schema evolution module supports some basic modifications to the class hierarchy. The message passing module is built on top of the object memory and schema evolution module and forms the basis for the user interface module. It includes the definition and support of the designed command language and error handling in addition to message passing. It consists of five submodules which are the lexical analyzer, parser, code generator, executor module and query processor. The designed language aims at solving the impedance mismatch problem. The secondary storage management and indexing module handles persistent objects by storing and retrieving them from secondary storage files and the indexing facility provides B-tree structures for efficient execution of value-based queries. The user interface module is also object-oriented and supports three types of users, namely, the developer/maintainer, the domain specialist and the end-user.

The prototype has been implemented on Sun workstations running under Berkeley Unix[1] 4.2 and the C programming language. The system is single-user and all objects are persistent and passive. Simple inheritance is supported resulting in a class lattice in the form of a tree. Authorization, concurrent access to data,composite objects and versions are not supported.

In this thesis, a focused survey on object-orientation and object-oriented database management systems is presented and the design and implementation of a single-user object-oriented database management system prototype is described with an emphasis on the message passing module. A command language is defined and a message passing scheme is proposed and implemented.

The thesis begins with a general introduction of the object-oriented approach. After the basic concepts, properties and application areas of the approach are introduced, the limitations of conventional database management systems and the advantages introduced by object-oriented databases are explained. Following a survey on some available object-oriented programming languages and database management systems, the developed object-oriented database management system prototype is presented. After a detailed description of the language developed and the message passing scheme applied, some open problems and future extensions to the system are listed.

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

2. THE OBJECT-ORIENTED APPROACH

2.1 THE BASIC CONCEPTS IN OBJECT-ORIENTATION

Object management refers to a set of run time issues such as object naming, persistence, concurrency, distribution, version control, security etc. Objects reside in a workspace which may be local and private or distributed and shared. Persistence methods must deal with local failures to resolve inconsistency problems [42].

2.1.1 OBJECTS AND CLASSES

In object-oriented programming, all conceptual entities are modelled as *objects*[50]. Programs are based on objects which are record-like data structures.An integer, string, aircraft or a submarine is an object. Objects are entities that combine the properties of procedures and data since they perform computations and save local state. The uniform use of objects in object-oriented systems contrasts with the use of separate procedures and data in conventional systems.

Each object is considered to have two parts: the *private part* and the *public interface part*[6]. The public interface part is used to communicate with other objects; and the private part specifies the internal implementation of the object. The private part can only be accessed through the public interface part. These two parts, together, capture both the state and the behaviour of the entity. The state of the object is represented using a collection of *instance variables*. Each instance variable is an object and therefore has its own private memory. A primitive object such as an integer or character has no instance variables. It only has a value which itself is an object. A default value may be specified for instance variables. In that case, such an

instance variable is called a default value variable. If the value for such an instance variable is not specified for an object, the associated default value will be taken as the value. A derived instance variable is one whose value is dependent on other information that is contained in the state of the object. It is not possible to set the value of a derived instance variable. The value of a derived instance variable is computed using a derivation function.

It is often desirable not to require that an instance variable's value belongs to a particular class, that is, not to bind the possible values of an instance variable to any single class. This means that, two different instances of the same class may reference objects from two different classes through the same instance variable. However, for the purpose of preserving data integrity, it is desirable to bind the domain, that is, the data type of an instance variable to a specific class and therefore implicitly to all subclasses of the class. Some object-oriented systems such as Smalltalk and GemStone are typeless allowing instance variables to take any value while others such as Hybrid and the developed object-oriented database management system prototype are strongly-typed requiring that each instance variable must be assigned to a domain from which it may take values.

Before creating an object, it must be described. After it has been described, this description can be used to create a whole set of objects. Such an object description is called a class and any object created using this description is called an instance of the class. Thus, objects with similar implementations and interfaces constitute a *class*; and the members of a class constitute its *instances*. Classes are used as [7]:

- generators of new objects
- descriptions of the representation of their instances
- descriptions of the message protocols of their instances
- a means for implementing differential programming
- repositories for methods for receiving messages
- a way of dynamically updating many objects at the same time
- set of all instances of a class

The class provides all the information necessary to construct and use objects of a particular kind, its instances. It is sufficient to know the messages

defined for a class and their input and output arguments, to create an instance of that class. Each instance of a class has its own copy of instance variables. Each class of objects is associated with a particular set of procedure-like operations called *methods*; and methods are performed when objects are invoked by *messages.*

A class may be associated with some class variables. The value of a class variable is shared by all instances of a class. Class variables and default value variables reduce storage and specification of objects.

Each instance has a single class while a class may have any number of instances. Allowing an object to belong to more than one class results in lower performance and a large increase in system complexity. This is because the structure of an instance object is variable; since it can belong to a number of classes, its instance variables cannot be determined beforehand and the identifiers for all classes to which an instance belongs must be stored with each instance. Only by examining the content of an instance object and determining the classes to which it belongs, it will be possible to determine its instance variables and methods.

The class concept reduces storage and duplication. It also provides conceptual simplicity.

There are two approaches to instantiation. In *static instantiation*, the object is instantiated at compile time and the object remains in the system through program execution. *Dynamic binding* requires run time support for allocation and for explicit deallocation or garbage collection [43].

Classes are used to describe the common properties of related objects, its instances. This class-instance approach has some complications resulting from the interaction of message look-up with the role of classes, which gives rise to the need of metaclasses and the use of classes for several different functions. One of the problems is the need to create a separate class for each object that has a distinct message protocol. If classes are treated as objects, to allow different classes to understand different initialization messages, each class itself must be an instance of a different class, namely, a *metaclass.* Another problem is that when designing a class the user must move to the abstract level of the class, write a class definition and then instantiate it and test it. To solve these problems associated with classes and metaclasses, *prototypes* are used [7]. A prototype is a standard example instance and new

objects are created by copying and modifying prototypes instead of instantiating classes. Also, prototypes are useful to avoid a proliferation of object classes in systems where objects evolve rapidly and display more differences than similarities (analogy and deviation). The difference between prototyping and instantiation is seen in terms of applicable inheritance mechanisms.

In the prototype model, an object consists of state and behaviour as in the class model. The state of an object is represented by a set of named *fields*. There are two components of an object's behaviour. The first component is a *method dictionary* and the second is a *protocol* that describes the set of messages the object declares that it can understand, the protocols required of the arguments to the messages and the protocols of the results returned by the messages. There may be several methods for receiving a given message. Similarly, one can send messages to an object asking it for information, asking it to change its state or asking it to change its behaviour. The only way to make a new object is to make a complete copy of an existing object including the state and the behaviour. Once the copy is made there is no relation between the original and the copy. Creating new objects by copying eliminates the need for metaclasses. The model handles object creation, manipulation and representation. The problems with this model are:

- There is no classification of objects, either by message protocol or by representation
- There is no way of updating a whole group of objects at a time

Constraints are used to express the inheritance relations among objects. There are two messages available for creating new objects: *copy* and *descendant*. The *copy* method makes a complete copy of the receiver and returns it. The second method makes a copy of the receiver and also creates a descendant relationship between the two objects.

The class and prototype model can be integrated and used to eliminate the need for metaclasses.

There are various ways an object can be stored in secondary storage. The two basic approaches are [56]:

- decomposing an object into its fields and representing each field as a binary relation
- storing objects by grouping all the fields of one object together on disk

The binary relation representation is better for associative access. It is not very good if all fields of an object is to be accessed.

For the object-based storage scheme, it is easy to access all fields for an object but associative access has lots of problems:

- many disk blocks must be read even with indexing
- data can be clustered only on one field
- redundancy and update problems

In the hybrid organization, binary relations are used on disk to speed associative access, with an object-based representation used in main memory to aid manipulations on single objects.

2.1.2 MESSAGES AND METHODS

A message is a request for an object to access, modify, or return part of its private part. It is like an indirect procedure or function call. Objects provide methods as a part of their definition. Messages completely define the semantics of an object. Methods describe how to carry out the necessary operations and a message specifies which method is desired but not how that operation is performed. The set of messages to which an object can respond is called its *interface*. Methods are not visible from outside the object. Objects communicate with one another through messages. A crucial property of an object is that its private memory can be manipulated only by its own operations and the messages are the only way to invoke an object's operations.

When a message is sent to an object, a message look-up is performed to determine the method associated with that message [50]. Generally, the message look-up starts from the class of the object which received the message. If the associated method is found, it is executed and the search is complete. If it is not found, the search continues in the superclass of that class. This look-up procedure searches the class lattice or hierarchy until the method is found or the root class is reached, in which case an error occurs. *Pseudo variables* [45] used in message calls as the receiver alter this message look-up procedure. In some object-oriented systems, the message itself is an object that the receiver processes as it wishes.

Pseudo variables are similar to other variables syntactically, but they are different semantically in that they cannot be assigned a new value during any particular invocation of a method. Two important pseudo variables are *self* and *super*. They both refer to the object that received the message currently being processed. They differ in the way message look-up is performed. When *self* is sent a message, the message look-up algorithm is identical to the way a look-up is performed when the message is sent externally, starting in the object's class. When *super* is sent a message, the look-up is performed starting in the superclass of the class in which the method that is currently executing is found. This pseudo variable gives objects a controlled way of accessing superclass methods. The *self* call allows the implementation of recursive methods and the *super* call is used to make incremental additions to an inherited method. The new behaviour added to the method may precede, follow or surround the call to *super*. An example from Smalltalk is:

```
initialize

    super initialize
    x ← y ← 0.
```

In the example, the initialization method for a class uses the initialize method of its superclass and also initializes the variables x and y to zero. The '←' operator is an assignment operator.

Message passing can be implemented as function calls or in a concurrent system as remote procedure calls [50]. Methods are equivalent to functions when there are no other methods associated with the message selector. Another implementation technique is based on actors [3] which are persistent, message passing processes [38]. In this approach, objects are in a way implemented on top of actors.

Messages and methods add data abstraction and polymorphism to the object-oriented model. A system is said to support data abstraction when it has a mechanism for bundling together all operations on a data type. The purpose of data abstraction is to change the underlying implementation without changing other parts [45]. Object-oriented systems support this idea since a class defines all the messages and methods, that is operations, that apply to its instances. Polymorphism refers to the capability of different classes of objects to respond to the same protocol.

In concurrent environments message passing could be either synchronous

or aynchronous [38]. In asynchronous message passing, the message is put on a queue and the sender is free to work on another task. In synchronous message passing, the sender is blocked until the message is delivered. In some systems the sender is blocked until the receiver sends a response. The problem with asynchronous message passing is infinite size buffers while synchronous message passing limits the amount of concurrency through blocking.

2.1.3 INHERITANCE AND THE CLASS LATTICE

A typical application may create and reference a large number of objects. If every object is to carry its own instance variable names and its own methods, the amount of information to be specified and stored can become unmanageably large. The class concept provides modularization and conceptual simplicity as well as reducing duplication, since all the messages, methods and instance variables shared by the instances only appear in the corresponding class definition. Another such tool is *inheritance*, in which a class can be defined as a *subclass* of another class inheriting the implementation and definition of its *superclass*. Thus, all classes in the system form a *class hierarchy*: a directed acyclic graph in which an edge between two nodes represents the IS-A relationship, that is, the child node is a specialization of the parent node and the parent node is a generalization of the child node [6]. The parent node is called the superclass of the child and the child node is called the subclass of the parent. Classes participate in the inheritance hierarchy directly whereas instances participate indirectly through their classes. A class needs to inherit properties only from its immediate superclass. So, by induction, a class inherits properties from every class in its superclass chain. A subclass may modify the definitions and implementations it inherits from its superclasses or may add new ones. Methods or definitions are overridden if a new method is provided for the old method's selector or a variable is redefined. Adding new behaviour to existing methods is usually done through the pseudo variable *super*.

Inheritance enables programmers to create new classes of objects by specifying the differences between a new class and an existing class. Thus a large amount of code can be reused through the sharing of behaviour between objects. Inheritance also facilitates top-down design. The inheritance and the class concepts avoid the specification and storage of some redundant information. They also provide information hiding. In a way, inheritance is a conceptual structuring mechanism.

There are many forms of inheritance depending on what is inherited and when and how the inheritance takes place. The related issues are [42]

- Does inheritance occur dynamically or statically?
- Are classes or instances clients of inheritance?
- What properties can be inherited?
- Which inherited properties are visible to the client?
- Can inherited properties be overwritten or suppressed?
- How are conflicts resolved?

Class inheritance reflects the similarity between object classes and is static inheritance. In partial inheritance, some properties are inherited and others are suppressed. It is convenient for code sharing but may create a messy hierarchy.

In dynamic inheritance [42], objects alter their behaviour in the course of normal interactions between objects. Dynamic inheritance occurs within the object model as opposed to schema evolution. It can be classified as follows:

- part inheritance_ An object explicitly changes its behaviour by accepting new parts from other objects. It is the exchange of values between objects. An object that modifies an instance also changes its behaviour, though limited by the object's class. If one considers instance variables and methods as values, an object may dynamically inherit new instance variables and methods from other objects.
- scope inheritance_ It occurs indirectly through changes in the environment. An object's behaviour is determined by its environment and acquaintances. The behaviour of an object changes when its environment changes.

Dynamic inheritance is possible with systems supporting prototypes.

Inheritance can be considered in four categories [47].

- *Type theory inheritance* is related to the similarity of the data structure between a subclass and a superclass. The structure of a subclass contains all the instance variables of its superclass and may include its own instance variables.

- *External interface inheritance* is the similarity of the externally visible interface provided by a class and its superclass. The class is able to provide all the external interface of its superclass and may specialize its superclass by providing its own interface as well.

- *Code sharing* and *reusability* is related to the property that if a subclass redefines the methods of its superclass, it can use the methods as provided by its superclass but can build upon them its own methods. Thus more complex routines can be built out of simpler ones by reusing but not duplicating the code.

- *Polymorphism* is related to operator overloading and allows a concrete operation to inherit its definition and properties from a generic operation.

There are two types of inheritance, namely *simple inheritance* and *multiple inheritance* [6] [50]. In simple inheritance, a class may have only one superclass forming a class hierarchy restricted to being a tree while in multiple inheritance, a class may have more than one superclass inheriting the definition and properties of all its superclasses and forming a lattice structure as the class hierarchy. Multiple inheritance simplifies data modelling and often requires fewer classes to be specified than with simple inheritance. However it introduces name conflicts, that is, the problem of two or more classes having instance variables or methods with the same name. The conflict may be between a class and its superclass or between the superclasses of a class. The name conflict problem between a class and its superclass may also be seen in simple inheritance and is solved by giving priority to the class. To solve the conflict problems in multiple inheritance, a conflict resolution scheme must be used. Either all instance variable or method names of superclasses must be distinct or a priority order for the superclasses should be specified. The default conflict resolution scheme provided by most systems chooses the property of the first superclass in the list of immediate superclasses when a conflict occurs. The problem with this approach is that the scheme depends on the permutation of the superclasses of a class. To overcome this problem, some systems allow users to explicitly change the permutation at any time.

There are two basic problems related to inheritance relationships between objects, that is, IS-A relationships. These are [57]:

1. the confusion between the inheritance of behaviour and the inheritance of representation.

2. the lack of any requirements for semantic relationships between a named operation on a type and a replaced operation with the same name on a subtype.

To distinguish between inheritance of behaviour and inheritance of representation, some other relationships can be introduced [57].

- The *behaves-like* relation. If a class B behaves-like a class A, B must have at least the behaviour of A. B may add new behaviour (properties, operations and constraints) but all A's behaviour must be supported by B. This relation does not have the side-effect of creating instances of superclasses which is seen in IS-A relations. The behaves-like relation could be implemented using IS-A relations by adding a new class which specifies the behaviour to be shared but has a null representation and which cannot be instantiated as a superclass of the two related classes. This requires schema evolution, the dynamic modificaton of the class lattice, and is troublesome. The behaves-like relation allows the user to retain the old structure while achieving the desired behaviour.
- The *subsumes* relation. The aim in this relation is for a subclass to access the representation of its subtypes. Subsumes guarantees that a subclass has at least the specification of its superclasses but it adds the ability for the subclass to access any state that is available in the superclass instance. This in a way loses some of the data abstraction seen in object-oriented systems.

Another problem with object-oriented systems is that operation refinement or operation redefinition is not based on any semantic properties of the operations involved. The aim of the IS-A relation is to induce a subclass relationship among the classes but with operation refinement one may end up with two classes related by an IS-A relationship but with completely different behaviour.

An approach to adding some semantics to the operation refinement problem is to allow an operation Op_1 on B to refine another operation Op_2 on a superclass A if and only if Op_1 behaves-like Op_2. B inherits all operations defined on A that are not refined by an object in B. In order for an operation to be a subtype of another operation type, it must have at least the behaviour of its supertype [57].

The *refines* relation is used to relate operation types. If B behaves-like A

and an operation Op_2 on B refines an operation Op_1 on A then an invocation of Op_1 on B will cause Op_2 to be invoked. Op_1 on A may only be refined once on a given subclass of A. Property refinement is also possible.

The discussion up to now was based on the class model. The prototype model also supports inheritance through some independent inheritance constraints. These are [57]:

- *inherits-field-names*($Object_1$,$Object_2$)
- *inherits-behaviour*($Object_1$,$Object_2$)
- *inherits-protocol*($Object_1$,$Object_2$)

A *descendant* constraint is defined to be the conjunction of the three inheritance constraints. These constraints can be used to support multiple inheritance but the name conflict problem still exists as in the class model.

2.1.4 OBJECT TYPES

An *object type* [42] is the same as an object class but when using typed objects whether they are manipulated in a consistent way must be checked statically. With static type checking there is no need to protect objects from unexpected messages. In object-oriented systems, with polymorphic operations and dynamic binding, some types may be equivalent or included in other types. The declared types of variables and arguments serve as specifications for valid bindings and invocations. One type conforms to another if some subset of its interface is identical to that of the second, that is, the first is a subtype of the second. They are equivalent if they conform to one another.

The difference between object classes and types can be interpreted as viewing the second as specifications. In the presence of dynamic binding, it is generally impossible to statically determine the class of a variable, but with the appropriate type rules, type checking can be performed. If dynamic binding is not supported then an object type will uniquely determine an object class. Type information can be useful for generic classes.

Class hierarchies are not the same as type hierarchies but they may overlap. Two classes may be equivalent as types, though neither inherits anything from the other.

2.1.5 OBJECT IDENTITY

Identity [28] is a property of an object that distinguishes it from other objects. Object identity provides the ability to distinguish objects from one another regardless of their content, location or addressability and the ability to share objects. This supports the modelling of arbitrarily complex and dynamic objects using versions which is a very important necessity in programming languages and database systems.

Consistency can be defined in terms of object identity. A consistent system must have the following two properties [28]:

a) Unique identifier assumption_ No two distinct objects may have the same identifiers, that is, the identifier functionally determines the type and the value of an object.

b) No dangling identifier assumption_ For each identifier in the system there is an object with that identifier.

The dangling identifier problem may be seen when an object is deleted. In most systems, a reference count representing the number of references to an object is kept for each object [28]. This reference count is updated whenever a reference to the associated object is added or removed. When the reference count of an object goes down to zero, the object is no longer referenced so it may be removed and garbage collection is applied. This is important for preserving the consistency of the system by avoiding dangling identities. This property is especially essential for temporal data.

WEAK SUPPORT OF IDENTITY VS STRONG SUPPORT OF IDENTITY

There are basically two dimensions involved in the support of identity. These are the representation dimension and the temporal dimension [28].

The representation dimension distinguishes languages based on whether they represent the identity of an object by its value, by a user-defined name or built into the language. Using values to distinguish objects provides a weak support of identity whereas built-in support of identity provides strong identity. A language providing a strong support of identity in the representation dimension must maintain its representation of identity during updates, use

identity in the semantics of its operators and provide operators to manipulate identity.

The temporal dimension distinguishes languages based on whether they preserve their representation of identity within a single program or transaction, between transactions or between structural reorganizations such as schema reorganization. If a language preserves the identities during only a single program or transaction, that language is said to support weak identity. The strongest support of identity in the temporal dimension is the preservation of identities through structural reorganizations. A language supporting stronger identity in the temporal dimension requires more robust implementation techniques to preserve its representation of identity.

Strong identity in the representation dimension is important for both temporary and persistent objects. Strong identity in the temporal dimension is important for persistent objects. For hybrid languages, which merge programming languages and database functionality, a strong identity in both dimensions is important as a result of the need for a uniform treatment of all objects because their status may change between temporary and persistent.

IDENTITY IN PROGRAMMING LANGUAGES

Most general-purpose programming languages are built based on temporal objects and a file system which is not part of the language is used to support persistent objects. In most languages weak identity is supported for temporal data.

Programming languages differ in the way they support identity in the representation dimension. Most languages use variable names as identities [28]. The actual binding of a variable to its name could be static, that is, at compile time or dynamic, that is, at run time. This approach confuses addressability and identity. Addressability is external to the object and provides a way of accessing an object within a specific environment and thus is environment dependent whereas identity is internal to an object and provides a way to represent an object uniquely and independently of how it is accessed. There are other limitations to this approach. One important problem is that a single object may be accessed in different ways and bound to different variables without a way of finding out whether they refer to the same object or not. To solve this problem operators for manipulating identity must be added to the language.

IDENTITY IN DATABASE LANGUAGES

Database languages must support strong identity in both the temporal and representation dimensions [28].

In relational database systems, a subset of attributes that uniquely determine a tuple, that is, an identifier key is used to represent the identity of an object. The identifier key is unique for all objects in the relation. Using identifier keys to represent object identity mixes the concepts of value and identity and thus introduces many problems. These problems can be listed as follows:

- Identifier keys are not allowed to change even though they are user-defined descriptive data. If an identifier key is allowed to be modified, this will cause integrity problems, discontinuity in identity and update problems in all objects that refer to it.
- Identifier keys cannot provide identity for every object in the relational model. Each attribute or subset of attributes cannot have identity.
- The choice of which subset of attributes to use as an identifier key may change.
- The use of identifier keys causes joins to be used in retrievals instead of path expressions. Path expressions [34] [36] which are used in object-oriented systems are much simpler.

With built-in object identity no joins are needed during retrievals. However using path expressions requires unique attribute names since nested names are used. Also, one may have some ambiguous paths. On the other hand, with built-in identity the insertion and deletion anomalies seen in relational systems and the need to normalize data are eliminated.

Using the notion of built-in identity in the language, the system may support strong identity in both the representation and temporal dimension. Strong support of identity in the temporal dimension is very important for representing the temporal aspects of the data since a single retrieval may involve multiple versions of an object. This requires the database system to provide a continuous and consistent notion of identity throughout the life of an object independently of its data or structure which may be modified. This value and structure independent identity can be used to link versions of an object and thus to support a temporal data model.

In some cases, the physical description of an object may not be stored in a single location and may be partitioned or replicated. Some reasons for this can be listed as follows [28]:

1. Some parts of the object may be shared by other objects as a consequence of the class hierarchy and inheritance. If each part is duplicated for each object, this will cause redundancy and some consistency problems.

2. For data recovery issues, some parts of the object may be replicated. In order to obtain maximum recoverability, the copies should be stored on separate media.

3. Some parts of the object may be physically partitioned based on the frequency of use together in order to improve performance.

4. In a temporal data model that supports versions, the most recent version may be kept separately from the other versions for faster access.

Using a value and location independent surrogate [14] [53] as object identity provides a way to relate the separately stored replicates or parts of an object.

PROGRAMMING LANGUAGE AND DATABASE SYSTEM HYBRIDS

Database systems and programming languages support different typing, computation and identity aspects. The data types supported in databases differ from those supported by programming languages. Programming languages are rich in manipulation capability while database systems include search and simple update capabilities. Most application programming languages are procedural whereas data manipulation languages are declarative as being declarative provides more opportunities for using indices and planning secondary storage access. Database systems support a stronger notion of identity compared to programming languages. These important differences introduce the impedance mismatch problem [28] [56] especially at the interface between the two systems. Much of the meta information in either system is reflected back at the interface and it must be defined redundantly in both languages. In addition, transformations must be defined whenever data or operations need to pass through the interface.

The solution to the impedance mismatch problem is unifying database systems and programming languages, that is, merging programming and database languages into a hybrid environment which includes a language with unified typing and computation [28] [56]. The aim is to obtain a language with a uniform treatment of types, computation and identity. Data instances of any type should be capable of being temporary or persistent. Any computation should apply uniformly to either temporary or persistent data, although computations which result in state changes of shared persistent data should be enveloped by a transaction. All types should employ the same notion of identity. One approach is to make programming language data types persistent. In these languages the file system is extended to support the same data types as in the language and type checking is done when file objects are imported into the system. A second approach is to combine programming and database language data types and database transactions.

OBJECT IDENTITY OPERATORS

Systems which support the concept of object identity must provide some operators for dealing with object identity. These include operators for checking if two objects are equal or identical, copying operators for deep-copy and shallow-copy of objects and an assignment operator. Shallow-copy and deep-copy operators indicate the degree of copying vs. sharing [20] [28] [31] [36].

Two objects are identical if they reference identical objects and they both have the same identity, that is, if they are actually the same object. One can differentiate between two types of equality, namely, shallow-equality and deep-equality. Two objects are shallow-equal if their values are identical. While checking if two objects are shallow-equal, the values of the components of the object are considered. On the other hand, when checking for deep-equality objects are recursively traversed comparing equality of corresponding components. Two atomic objects are deep-equal if they have the same value. Shallow-equality and deep-equality are the same for atomic objects. Two non-atomic objects are deep-equal if their corresponding components are deep-equal. Two identical objects are deep-equal and shallow-equal and two shallow-equal objects are at the same time deep-equal.

When an object is assigned to another one, the two objects will share the same object. The shallow copy of an object is a new object which shares the values of the other object whereas the deep copy of an object is a new object with its own identity and its subobjects are new objects with their

own identity but having the same values as those of the other object. After a shallow-copy operation, the two objects become shallow-equal whereas the deep-copy operator generates a new object which is deep-equal to the other object.

IMPLEMENTING OBJECT IDENTITY

There are many ways of implementing object identity and they can be compared depending on the amount of value, structure and location independence they provide [28]. Data independence means that the identity of an object remains unchanged no matter what changes are made in its value and structure. On the other hand, location independence means that the identity of an object does not change even if the physical location of the object changes. Location independence is especially important in supporting load balancing in a distributed system. Some of the major implementation techniques can be listed as follows :

- Identity through physical address_ The physical address could be the real or virtual address of the object. It is fully data independent unless changes in the data cause the object to be moved in the address space due to size problems, but using the physical address as the identity does not allow an object to be moved so there is no location independence. However if the virtual address is used, pages may be moved within a virtual address space providing some location independence. Object sharing among multiple programs is limited.
- Identity through indirection_ The use of an object-oriented pointer (oop) to identify objects as in object-oriented systems is a way of supporting identity through indirection since the oop is an index into an object table which provides a mapping from oops to physical addresses. An indirect physical address or indirect virtual address can be used to identify objects. They provide full data independence, allow object sharing among multiple programs and by allowing objects to be moved within a physical or virtual address space they provide some location independence.
- Identity through structured identifier_ This approach provides full data independence and allows objects to be moved within one disk or server. Sharing of objects is also supported. A part of the structured identifier used to identify an object describes the location of the object.

- Identity through identifier keys_ It provides full location independence but is value and structure dependent since they consist of values, they are unique only within a specific relation and they are applied only to tuples.

- Identity through tuple identifiers_ This approach provides full location and value independence but is structure dependent since tuple identifiers are only unique within a relation and they are applied to tuples. Tuple identifiers are system generated identifiers which are unique for all tuples within a single relation and have no relationship to physical location.

- Identity through surrogates_ Surrogates are system-generated, globally unique identifiers, completely independent of physical location. They provide full location independence and if surrogates are generated for each object, full data independence is also obtained. However if a unique surrogate is generated for each tuple then value independence is obtained but full structure independence is not supported.

2.2 EXTENSIONS TO THE BASIC MODEL

The basic extensions that should be added to the model and which are especially necessary for artificial intelligence, knowledge representation, CAD/CAM and office information system applications are *schema evolution* [5] [6], *composite objects* [6] [50], *version management* [6] [10] and *indexing* [34] [35] [36].

2.2.1 SCHEMA EVOLUTION

Schema evolution is the ability to dynamically make changes to the class definitions and the structure of the class lattice [6]. Most systems support only a few changes to the schema and class definitions without requiring system shutdown. The operations that should be supported in an object-oriented system can be listed as follows [6] [5]:

1. changes to the contents of a node (a class)
   (a) changes to an instance variable
      i. Add a new instance variable to a class

ii. Drop an existing instance variable from a class
iii. Change the name of an instance variable of a class
iv. Change the domain of an instance variable of a class
v. Change the inheritance (parent) of an instance variable
vi. Change the default value of an instance variable
vii. Manipulate the value of a class variable
    A. Add a class variable
    B. Delete a class variable
    C. Change a class variable

(b) changes to a method
    i. Add a new method to a class
    ii. Delete an existing method from a class
    iii. Change the name of a method of a class
    iv. Change the code for a method of a class
    v. Change the inheritance (parent) of a method

2. changes to an edge
    (a) Make a class a superclass of another class
    (b) Remove a class from the superclass list of a class
    (c) Change the order of superclasses of a class

3. changes to a node
    (a) Add a new class
    (b) Delete a class
    (c) Change the name of a class

There are some properties that the class lattice must have. These are known as the *invariants* of schema evolution [6]. The class lattice is a rooted and connected directed acyclic graph. It has only one root. In the case of simple inheritance, the class hierarchy is a tree. All instance variables and methods of a class, whether locally defined or inherited, must have distinct names. All instance variables and methods of a class have distinct origin. A class must inherit all instance variables and methods from each of its superclasses. If an instance variable V_2 of a class is inherited from an instance variable V_1 of its superclass, then the domain of V_2 must either be the same as that of V_1 or a subclass of V_1. Any changes to the class definitions and

to the class lattice must preserve these invariants which ensure that changes to the schema do not leave the database in an inconsistent state. When applying schema change operations some rules are needed [6]. These are conflict resolution rules, property propagation rules and lattice manipulation rules [6].

An important problem related to schema evolution is the problem of methods becoming inoperable as a result of schema change operations. Another problem is seen when the structure of a class which has some instances is modified. One approach is to modify all instances to reflect these changes immediately after the change is made in the class definition. A second approach is just to modify the class definition and modify the instances whenever they are referenced. The first approach is cumbersome and an overhead. However, the second approach is very difficult to implement and may cause inconsistencies. It also requires a way of keeping track of which instances have been modified and which have not [43].

It is difficult to decide whether schema evolution is actually a practical problem or a theoretical problem. One approach to schema evolution is to let the user specify all the operations required to perform the necessary change in the schema and for preserving consistency and eliminating conflicts. Everything is left to the user and the system just carries out the operations specified by the user. If the operations specified by the user cause some consistency problems or some conflicts, the operations are not performed and an error occurs. In this case, schema evolution is a practical problem. However, if the system is required to resolve all conflicts and preserve consistency while making the necessary changes, schema evolution becomes a more theoretical and difficult problem.

2.2.2 COMPOSITE OBJECTS

Many applications require the ability to define and manipulate a set of objects as a single logical entity. A *composite object* is an object with a hierarchy of exclusive component objects considered as a unit of storage, retrieval and integrity. The hierarchy of classes to which the object belong forms a *composite object hierarchy* [6].

The basic object-oriented data model does not support composite objects; an object references but does not own other objects. A composite object captures the IS-PART-OF relationship between a parent class and its component

classes while a class hierarchy represents the IS-A relationship between a superclass and its subclasses.

Composite objects introduce the concept of *dependent objects* [6] [50]which add to the integrity features of an object-oriented data model. A dependent object is one whose existence depends on the existence of other objects and is owned by a single object. Since a dependent object cannot be created before its owner exists, the composite object hierarchy must be developed in a top-down fashion, that is, the root object of the hierarchy must be created first and then the children. When an object of a composite object is deleted all its dependent objects must also be deleted.

An object may contain references to both dependent objects and independent objects or to only dependent or independent objects. Such a general collection of objects is called an *aggregate object.* A composite object is, in fact, a special kind of aggregate object.

When a composite object is instantiated all its parts are also instantiated. The instantiation process is recursive so composite objects can be used as parts. The automatic instantiation of all parts brings the restriction that a composite object cannot be a part of itself. An alternative is to instantiate parts on demand [50].

The composite object concept supports performance improvements through the clustering of related objects on disk. All components of a composite object should be clustered together since whenever the root is accessed, most probably the other component objects will also be accessed.

Composite objects increase information hiding and data encapsulation through the property of *value propagation* [6] which refers to the sharing of the value of an instance variable between instance objects. In contrast, inheritance is the sharing of the name of an instance variable between instance objects. Values can be propagated only if an object has an instance variable which has the same name as some instance variable of a higher level object. Value propagation to a lower-level object takes place from the lowest-level object containing an appropriate value and is not automatic and must be specified in the definition of the composite object schema. Once value propagation is specified, it takes precedence over inheritance.

2.2.3 INDEXING

In object-oriented database managenent systems, system defined surrogates are used to identify objects. However, since these surrogates are value and location independent, in order to access the data by value a search has to be performed. To avoid this sequential search, an indexing mechanism must be added to the system [34] [35] [36].

There are some problems associated with value-based access [34]:

- Language issues_ There are two basic considerations: when to invoke auxiliary access paths for associative searching and whether to index on an object's structure or protocol. One approach is to provide a special class for handling indexes. This approach reduces physical data independence and the user has to perform index maintenance. Another approach is to consider every expression as a candidate for indexed access. A better approach is to denote certain statements as candidates for indexed access or to have a sublanguage to make use of indexes. Adding an index handling sublanguage to an existing language may cause an impedance mismatch problem and will complicate the compiler. The sublanguage may be procedural or declarative. The other major issue regarding languages is whether indexes are based on the instance variables, that is the structure of the objects or the responses to messages, that is the protocol. Indexing on structure violates the privacy of an object while indexing on protocol introduces problems when the protocol changes.

- Index structure_ Indexing could be provided only on the immediate instance variables of an object or on the instance variables and their instance variables. If an index is provided on paths with multiple links that is multiple instance variables, a single index could be provided for the whole path or several indexes could be provided, one for each link. The sequence of links is called a path expression [34] [35] [36]. With a single index for each path, there are fewer indexes to maintain and fewer indirections to be made during associative access. Indexing by links allows sharing of indexes. Some other considerations are

  - The type of the objects to be indexed_ Indexing is generally applied to collection or set objects. The objects constituting the elements of the collection or set to be indexed should be of a certain type. They could be required to be an instance of a class. An alternative

to using a class as a type is the use of kinds.`A *kind* is a class and all its subclasses.

  - Manipulation of undefined values along the index path
  - Supporting identity indexes or equality indexes_ An identity index supports searching a collection on the identity of some subobject without reference to an object's internal state. It does not support range queries. An equality index supports look-up on the basis of the value or internal state of objects and range queries [36]. In a path expression, all links except the last one must be identity indexes and the last one could be an identity or equality index.
  - The comparison operators supported during range indexes

- Indexing on classes or collections_ Indexing on classes presents some authorization problems and also applications which do not use the index are subject to the index-related overhead for indexed instances they use. However, it is easier to trace changes to an object which affect the index on that class. Each subclass may maintain its own index or the index on a class may include its subclasses. As an object may be a member of several collections, if class indexes are supported and queries against collections are made, there will be a test for collection membership in addition to the index access. Indexing on collections allows the possibility that instances of subclasses be included in a collection that is indexed. A collection of all instances of a class may be created and indexed to implement indexing on classes. A third approach which is the combination of the other two approaches , maintains a single index per class but only adds members of a certain collection to that class.

2.2.4 TEMPORAL ASPECTS AND VERSION MANAGEMENT

Most conventional databases represent the state of an enterprise at a single moment of time. Although the contents of the database change as new information is added, these changes are viewed as modifications to the state with the old data being deleted from the database. The current contents of the database are regarded as a snapshot of the problem [49]. Versions are variations of the same object that are related by the history of their derivation [6].

A lot of research has been done on representing time in databases. The

database management systems which represent the progression of states of a problem over time are temporal databases. Changes in the data are viewed as additions to the information in the database or as versions of the data. Temporal databases are generalizations of conventional, that is, static databases.

TEMPORAL ASPECTS

There are two major approaches used to incorporate time in relational database systems. One is to extend the semantics of the relational model to incorporate time directly. The other is to represent time as additional attributes [49].

There are three different notions of time. *Valid time* is the time the data becomes valid. *Transaction time* is the time the data is entered into the database. The third aspect is *user-defined time* [49].

Static databases are databases which model the dynamic real world as a snapshot at a particular point in time. As changes are made, past states of the database are discarded. An approach is to regard a relation as a sequence of static relations indexed by transaction time. The user can get a snapshot of the relation as of some time in the past, which is in fact a static relation, and make queries upon the static relation by moving along the time axis and selecting the relation. The operation of selecting a static relation is called *rollback* and such a database is known as a *static rollback database.* Changes to a static relation may only be made to the most recent static state. Each modification creates a new static relation. *Historical databases* represent valid time. They support historical queries which may utilize queries from the past. They are represented as a series of static relations indexed by valid time and the semantics represent the reality more than the update history. While a static rollback database views tuples as being valid at some time as of that time and a historical database views tuples as being valid at some moment as of now, a *temporal database management system* makes it possible to view tuples as being valid at some time relative to some other moment. In temporal databases both valid time and transaction time are represented. A temporal relation may be considered as a sequence of historical states each of which is a historical relation. The rollback operation on a temporal relation selects a particular historical state, on which a historical query may be performed.

User defined time is necessary when additional temporal information for which valid time and transaction time are insufficient, has to be stored. It is

application dependent and since it is not interpreted by the database management system it is the easiest to support. Only an internal representation and input and output functions are necessary. An example for user defined time is the effective date of some information. Transaction time is application independent, easy to implement and can be automated by the system on the other hand,valid time is application dependent and it is difficult for a system to automate it. Since transaction time is system generated and cannot be modified by users, it provides high integrity. Valid time must be modifiable by users when a discrepancy is discovered between the real world and its database model. Storing transaction time is also useful for synchronizing concurrent transactions [12].

VERSION MANAGEMENT

Version management is especially important for CAD/CAM and office information system applications and is actually an important requirement for object-oriented systems.

There are basically two ways of creating versions [6] [10]. One approach is the linear generation and storage of objects. This is the case if only one version can be generated from an older version. The most current version is the newest version which is at the same time the most correct and most complete version. Another approach is to store the versions in a hierarchy. In this case, more than one version may be generated from an older version and it is quite difficult to determine the most current version by just looking at the hierarchy.

There are two ways to bind an object with another versioned object [6]. In static binding, the reference to the object includes the full name of the object, the object identifier and the version number. In dynamic binding, the reference only needs to specify the object identifier and may leave the version number unspecified. The system selects the default version number. In some systems the default version is the most recent version but approaches are needed for versions in the form of a hierarchy.

2.3 BASIC PROPERTIES OF THE OBJECT ORIENTED APPROACH

The basic notions in the object-oriented approach are [38] [42] [45]:

- information hiding
- data abstraction
- data independence
- homogeneity
- message passing
- dynamic binding
- inheritance
- polymorphism and overloading
- reusability
- interactive interfaces
- concurrency

Two other important properties are multiple inheritance and automatic storage management. Multiple inheritance allows a class to have more than one superclass thus providing more code sharing but increasing the complexity of the system through the conflicts that may occur between the multiple superclasses.

Automatic storage management techniques such as reference counting and garbage collection allow users to ignore details related to the release of an object's storage. As a result, application code becomes cleaner and the system becomes more reliable.

2.3.1 INFORMATION HIDING

Information hiding [38] provides reliability and modifiability by reducing interdependencies among software components. The state of a software module

is contained in private variables, visible only from within the scope of the module. Only a localized set of procedures directly manipulates the data. In addition, since the internal state variables of a module are not directly accessed, a carefully designed module interface may permit the internal data structures and procedures to be changed without affecting the implementation of other system modules. Object-orientation provides information hiding since an object captures both the state and the behaviour of an entity.

2.3.2 DATA ABSTRACTION

Data abstraction [38] [42] [45] [50] [56] is a way of using information hiding. An abstract type consists of an internal representation and a set of procedures used to access and manipulate the data. Since objects capture both the state and the behaviour of an entity, an object-oriented system directly supports data abstraction. In other words, the behaviour of an object rather than its implementation is of interest and the actual implementation is hidden. The class concept provides data abstraction and the message concept provides procedural abstraction. The two concepts together result in information hiding.

Each class of objects defines an interface that is the only way that other objects can manipulate objects which are instances of that class. If this is also true for subclasses and superclasses, that is, if even the subclasses and superclasses can only communicate using the messages specified in the interface, full data abstraction is attained and the class can be modified without affecting the other classes as long as the interface does not change.

Operator overloading and generic functions provide data abstraction. Operator overloading permits a program to use multiple operators with the same name. The distinction between operators can be determined at compile time depending on the type and number of operands. Generic functions permit the definition of a module to be used with different data types. A generic function can be considered as a procedural template that can be parameterized with actual types during compilation of programs.

Through data abstraction, it is possible to decompose a large system into smaller, encapsulated subsystems that can be more easily developed, maintained and that are more portable. Data abstraction may be used to provide [42]

- multiple object instantiation
- behavioural sharing through various inheritance mechanisms
- verification of correct object usage through strong-typing
- structuring of resources in concurrent applications

Thus, data abstraction aids the system in supporting reusability, object types and concurrency.

2.3.3 DATA INDEPENDENCE

Data independence [38] [45] [56] is related to the fact that objects communicate using message passing. An object sends a message and the other selects the method to perform the operation. This property states that objects have control over their own state and existence and is important for ensuring the reliability and the modifiability of the system by reducing the interdependencies between objects. Another form of independence is the ability to add new types at run time.

2.3.4 HOMOGENEITY

Homogeneity [38] is related to the fact that everything is an object. The degree of homogeneity supported in a system depends on whether classes are objects and whether there is a differentiation between user and system defined classes and between active and passive objects. Most systems support passive objects. Active objects [43] [52] are objects that can perform automatic actions. In other words, they may be triggered.

2.3.5 MESSAGE PASSING

The basic property of object-oriented systems is the use of the object-message paradigm instead of the traditional data-procedure paradigm. In the data-procedure paradigm supported by most conventional programming languages, active procedures act on the passive data that is passed to them. In strongly-typed languages, there would be different functions to perform the same operation on different types of data while late-binding languages support

generic functions in which the data type determines the operation at run-time. Generic operations are primitives restricted to a small class of data types such as numbers or they are functions defined in terms of such primitives [38] [56].

In object-oriented systems instead of passing data to procedures, objects are asked to perform operations on themselves using messages, that is, data are active and operations are passive.

2.3.6 DYNAMIC BINDING

Generally, conventional languages perform early binding. For example code is bound to a name at compilation and a name to an address at link time. Late binding provides flexibility at the expense of efficiency in contrast to early binding. Early binding should be applied in a stable environment where the bindings will not change. Late binding is applied in unstable environments [45] [56].

Operator overloading and generic functions are only suitable if the data is homogeneous and thus the types of the operations can be determined at compile time. Dynamic binding is necessary when dealing with heterogeneous data. The basic approach used in dynamic binding is polymorphism which is similar to operator overloading where the procedure invoked is fixed at compile time. In polymorphism, the same operator performs different operations depending on its operands and the operation is determined at run-time. In object-oriented systems messages support polymorphism and dynamic binding. The same message may elicit a different response depending on the receiver.

2.3.7 INHERITANCE

Inheritance [38] [45] [50] allows the creation of classes and objects that are specializations of other objects. The subclass inherits instance variables, class variables and methods from its superclass. It may add its own instance variables, class variables and methods to its definition. It may also override the variables or methods it inherits. Methods are overridden when a new method for an inherited selector is added to the class definition. Some languages support the addition of new behaviour to existing methods.

Inheritance and the class hierarchy result in clustering by specifying shared information only once in a superclass. The overriding or redefinition of methods and instance variables during inheritance enables the user to design the schema by first specifying the ideal structure of the problem and then specifying the actual structure as being analogous to the ideal structure but with some deviations which are explicitly specified. The class hierarchy also facilitates top-down design and thus provides localization of information. Data abstraction and inheritance simplify the specification of complex structures.

2.3.8 POLYMORPHISM AND OVERLOADING

A polymorphic function is one that can be applied uniformly to a variety of objects [42] [45] [50] [56]. There are two ways polymorphism can occur:

- Same operation maintains its behaviour transparently for different argument types.
- Two operations share the same name but have completely different behaviour (adhoc polymorphism or overloading of operation names).

Class inheritance is closely related to polymorphism. Polymorphism enhances software reusability. It may or may not impose a run time overhead depending on whether dynamic binding is supported. With statically bound variables, the method can be detected at compile time while with dynamically bound variables a run time method look-up must be performed.

While object classes factor common properties of classes in parent classes, generic object classes do so by partially describing a class and parameterizing the unknowns. These parameters are the classes of objects that instances of generic classes will manipulate. There are two types of generic objects [42]:

- homogeneous container objects. They operate on any kind of object.
- tool objects. They can only operate on certain object classes.

Depending on the nature of the parameters, it may or may not be possible to compile generic classes before the parameters are bound. If parameters have to be statically bound, generic classes behave like macros.

2.3.9 REUSABILITY

Instantiation, class inheritance, overloading, polymorphism and parameterization enhance reusability.

2.3.10 INTERACTIVE INTERFACES

Most object-oriented systems are suitable for and provide an either menu or mouse driven iconic graphical user interface. In fact, sometimes the name object-oriented is used for systems providing such interfaces [56].

2.3.11 CONCURRENCY

There are two approaches to concurrency and communication [38] [42] [43]:

- active entities (processes) communicate indirectly through shared passive objects
- active entities communicate directly with one another by message passing

In the approach based on shared passive objects, the shared memory is structured as a collection of passive objects and a process is a special kind of active Process object. Operations are performed on the passive objects according to their interface. A mechanism is required for active objects to synchronize their accesses to shared objects. This approach is not homogeneous since there is an important difference between active and passive objects. It is not possible to directly interact with active objects. Two active objects can only communicate through a passive intermediary.Hidden message passing is required to extend this approach to a distributed environment.

In the message passing approach, any object can communicate with any other object. Objects become active in response to a communication. Threads of control are determined implicitly by message passing whereas in the first approach each thread of control was localized in an explicit process object.Explicit synchronization is not needed since message packages both communication and synchronization but a style of message passing must be used.

With message passing, strong-typing means that a message passing expression is type-correct if the message being sent is guaranteed to be valid for the recipient. The untyped view can be supported if all objects can handle any message sent to them.

2.4 APPLICATION AREAS OF THE OBJECT-ORIENTED APPROACH

The major application areas of the object-oriented approach are programming languages, database management systems, knowledge representation, CAD/CAM systems and office information systems. Each of these application areas make use of different aspects of object-orientation [56].

2.4.1 PROGRAMMING LANGUAGES

In object-oriented programming languages, every object has a set of operators which are used to operate upon and change the state of an object. This provides data encapsulation. Another concept seen in object-oriented programming languages is operator overloading. Operator overloading is using the same operator symbol to denote distinct operations on different data types. The meaning of an operator is thus overloaded and can be resolved on the basis of the operand types. When interpreting a message, an object-oriented language first binds the head to an object class, then binds the rest of the message to a method of that class. Overloading appears if distinct methods are given the same name in different classes.

There is a fairness problem when operators have two or more operands. Then one operator must be selected as the message receiver that controls the overloading, while the others ,that is, the message arguments are relegated to appendices of the method. Late binding of methods means that no recompilation is needed favoring flexibility at the expense of speed.

2.4.2 DATABASE MANAGEMENT SYSTEMS

In object-oriented database management systems an object-based approach is used as opposed to the value-based paradigm used in the conventional

database management systems. In relational systems, the information is stored in terms of tuples and relations. Tuples can only be distinguished on the basis of their values. In object-oriented systems a hidden permanent unique identifier is assigned to each entity record. An entity occurrence can refer to another using the latter's identifier. This policy provides a simple means of supporting relationships between entities and referential integrity constraints. The object-oriented approach also provides better support for managing time and changes in databases. Any changes in an entity are automatically seen by entities referring to it providing referential transparency. Another advantage is related to version management. Old versions of an object can be archived and later retrieved using their unique identifier and a timestamp or version number.

2.4.3 KNOWLEDGE REPRESENTATION

Frames are capable of storing both specific and general knowledge and of accommodating both descriptive and prescriptive computation. In a frame system, the properties of both specific objects and generic objects (classes) are described by their slots which may contain references to other frames defining their relationship, actual values or procedural attachements to compute them. Generic classes are classified using the IS-A relationship and the membership of an instance object in a class object is described using the AS-A relationship. The capability of unifying the treatment of data and metadata as seen in frame systems represents an important strength of object-oriented systems.

2.4.4 CAD/CAM SYSTEMS

CAD/CAM systems require unifying the treatment of data and metadata since schema level information has to be frequently manipulated as regular data. They require versions and multiple design transaction support.

2.4.5 OFFICE INFORMATION SYSTEMS

The object-oriented approach is very suitable for office information system applications since the approach can easily support menu and icon based interfaces and multimedia document management. The approach is also quite

suitable for distributed applications.

2.5 OBJECT-ORIENTED PROGRAMMING LANGUAGES AND SOME EXAMPLES

Object-oriented programming environments support reusability and provide tools for designing, selecting and reusing objects and managing an evolving software base. Object-oriented techniques in programming languages enhance reusability, maintenance and robustness through extendible type systems and ease the development of concurrent and distributed applications.

The object-oriented approach is very suitable for many problem areas in programming languages. It is extensively used for simulation programs, systems programming, graphics and artificial intelligence. It is also used for the theory of frames and their implementation in knowledge representation languages. Some object-oriented languages were developed from scratch while others are extensions to existing languages especially Lisp. Some of the most important object-oriented programming languages are Simula [50], Smalltalk [12] [13] [16] [20] [23], C++ [13], Objective-C [13], Loops [50], Flavors [50], Hybrid [29] [39] [40] [41] [43] and Actors [3].

Objects are a uniform programming element for computing and saving state. This makes them ideal for simulation problems where it is necessary to represent collections of things that interact. They have also been used for applications in systems programming since many things with states such as processes, directories and files must be represented. Augmented by annotation mechanisms they have also become important in the current tools for knowledge engineering.

As object-oriented languages have become more widespread, a lot of work has been done for developing standards so that objects could be used as a portable base for program and knowledge bases.

2.5.1 SMALLTALK

Smalltalk is an integrated programming language and programming environment. One of the most important components of Smalltalk is the virtual machine which consists of all system defined and user defined classes and

operations. Everything in the system is considered to be an object and this homogeneity supports consistency. All constants and contents of variables are objects. However, message selectors, comments and punctuation symbols are not considered as objects. Everything that is not an object is a message selector. The message tells an object what to do and the command carried by the message is known as the selector. A message is, in fact, a message selector with its operands. All operations are performed using messages, that is, by specifying an object, sending it a message and getting back another object as the result. Control structures and arithmetic operators are also implemented as messages. Objects with similar structure and behaviour are grouped into classes. The classes are organized in a hierarchy and simple inheritance is supported. Smalltalk also supports metaclasses. When an object receives a message, the system checks if the corresponding method appears in the object's class definition. If there is no such method in the object's class definition, the superclasses of that class are searched. Figure 2.1 shows examples of class definitions, message calls and method definitions in Smalltalk.

Smalltalk only supports temporary objects. It provides automatic garbage collection so the lifetime of an object is determined by the system and not by the user. When an object is no longer referenced, its memory is reclaimed for reuse. This approach is based on a kind of reference count mechanism. Having automatic garbage collection eliminates the dangling pointer problem in which invalid object identifiers produced by freeing some objects remain in the system and are later accessed. It also eliminates the problem of not having enough memory during the execution of a long-running program because unnecessary objects have not been freed. However, automatic garbage collection has some efficiency problems. Especially in combination with extensive use of dynamically bound messaging, it can cause costs in machine resources making Smalltalk unsuitable for performance-critical applications.

Smalltalk supports the object identity concept. In Smalltalk, objects are not directly identified by their memory address but they are identified by an offset into a table of object descriptors. One field of the entry specifies the memory address of the object. This hides memory management from the user making address handling the job of the system and allowing it to move objects in the memory. The value of a smallinteger is used as its own identifier so it does not occupy a slot in the object table. The object identifier is 16 bits, one bit is used as a flag and the remaining 15 bits are used as the value of a primitive type or the identifier of an object. The flag bit is used to signal that the identifier is not that of a smallinteger but an offset into the object table.

a) Object subclass: #Person

instance variables: 'name address'.

b) Person subclass: Student

instance variables: 'college class major'.

c) person $\leftarrow$ Person new.

d) person name: 'John'.

e) personList add: newName before: currentName.

f) name: aName name $\leftarrow$ aName.

g) name $\uparrow$name.

h) person birthDate month.

Figure 2.1: Some example class definitions, message calls and method definitions in Smalltalk

There is only one type, namely the object, so there is no need for declaring data types and arguments. Thus Smalltalk is typeless.

There are three major operations that can be performed in Smalltalk which are

- declaring object names and assigning them values
- sending messages
- defining new classes and methods

Actually, Smalltalk provides a *browser* for reading, changing and compiling methods so no commands are necessary for these operations.

In a message call, the receiver appears to the left of the message selector which is in turn followed by the arguments. A method may have some local variables. Such local variables are specified within vertical bars. If a variable appears in a method, it can be of six types [16]:

- an instance variable in the class of objects for which the method is defined
- an argument of the message
- a temporary variable local to the method
- a class variable
- a pool variable
- a global variable

Class variables are shared by all instances of the related class and its subclasses, pool variables are valid across designated classes and global variables are shared by all classes. Class, pool and global variables are not used often, so there is a restricted type of lexical scoping. This strict information hiding eliminates some scoping related problems and names need not be modified to avoid naming conflicts. Temporary variables exist for the message's execution life. Class and global variables are used for longer term storage and are not in the local memory of instances of the class. Variable names and message names can easily be overloaded.

A value is returned as the result of each method. The returned value is always an object. The returned value may be significant or may merely inform the sender that a requested action is complete. If the object to be returned from a method is not specified, the object that received the message is returned. The result of one message can be used as the object that receives another message or as an argument in another message. Thus messages can be concatenated.

There are basically four types of messages in Smalltalk [23].

1. *Unary messages_* A unary message is a message with no arguments so it is composed of two parts: the name of the object to receive the message and the name of the message. An example is the following message which retrieves the value of the age instance variable for an employee object.

   employee age

2. *Keyword message expressions_* They may have as many arguments as there are parts in the keyword. The selector of a keyword message is composed of one or more keywords, one preceding each argument. A keyword part is a simple identifier followed by a colon. An argument is needed whenever a message selector is followed by a colon. Some examples are

   aPerson name: personname
   aArray at: 1 put:5

   The first example sets the name of a given person and the second example places 5 as the first element of the array. The selectors in the examples are name: and at:put:. Some other examples of keyword message selectors are ifTrue:, ifFalse:, ifTrue:ifFalse and add:before:. Both the receiver and the arguments can be variable names, constants or other expressions.

3. *Binary messages_* They are like keyword messages with a single argument. The selector of a binary message is always one or two characters from a designated set of special characters. For example

   $3 + 4$
   $total <= max$

   are binary messages. Binary messages are generally used for arithmetic operations and comparisons.

4. *The assignment operator*. It is used to store values into variables and is handled as a message. An example of an assignment message is as follows:

   sum ← 3 + 4

Since message expressions can be nested to arbitrary levels, precedence rules are provided and can be overridden using paranthesis. Messages without arguments are executed first. They take precedence over adjacent messages that are operators (+,−,×,/,=,> etc.). Operator messages are executed before adjacent messages whose names contain colons. Two messages without arguments are evaluated left to right. This is also true for operators.

When a method needs to invoke a method defined in its class, it will specify the receiver to be *self*. Using self message calls, recursive methods can be implemented. The *super* message call, that is a message call in which the receiver is super, allows a subclass to make some incremental changes to the method it inherits from its superclass.

Smalltalk control structures are handled using the object-message paradigm. The message selectors related to control structures are ifTrue:, ifFalse:, ifTrue: ifFalse:, ifFalse: ifTrue: and do:. The do: selector can be sent to collections of various types and it iterates through the members of the collection.

Expressions are usually evaluated immediately. However, expressions can be stored for later execution by enclosing them in brackets and thus creating a block construct. A block is an object that represents a sequence of instructions whose execution is deferred until the block is sent a message to evaluate itself. There is a system defined class called Block and all created blocks are instances of this class. Most control structures are implemented as messages to objects that take blocks as arguments. An example of such an if construct is as follows:

```
(index <= limit) ifTrue: [ total ← total + (list at: index)]
```

In most languages, the execution of a block can be deferred no longer than the lifetime of the enclosing scope. In Smalltalk, blocks are allocated from the heap and are disposed of by the garbage collector. Any block and its context, that is, the environment in which it was originally created can be held indefinitely and executed any time just by sending it a message. This is independent of the message that created the block. The block is given a

reference to the caller's context when the block is created and this context persists as long as a reference to it exists in the system. Thus, a block can access the context of the calling method that created it including its local variables and arguments even when the value message is invoked by another object.

Smalltalk source code and methods are compiled into an intermediate form, called *bytecodes*, which is then interpreted. The compilation is done incrementally as new classes, messages and methods are defined. The implementation of message passing is based on bytecodes and the system defined class, Context. The compiled form of a method corresponds to an instance of the class Context. The symbolic references in the method are translated into indexes of structures similar to symbol tables. Each method context has its own structures, one corresponding to temporary variables, one corresponding to message and class references and a third on for class and global variable references. The context associated with the method being executed is known as the current context and when a message call is executed, the context associated with the method corresponding to the message becomes the current context and the other context is pushed on to a stack. When a return from a method is executed, the context on top of the stack becomes the current context. Blocks are executed in a similar fashion.

2.5.2 SMALLWORLD

Smallworld [31] is a programming environment in which the real world is represented using objects that have properties. A property represents a fact about the corresponding real world entity. Smallworld actions (programs), which operate on objects consider all of the relevant facts (all properties) concerning the objects they manipulate. It is not a programming language but a system where application programs can be developed. It is more than a database since it provides actions for manipulating objects. All of the facts related to the objects to be manipulated are stored with these objects.

Classes and superclasses are supported. Smallworld minimizes the differences between classes and non-class objects resulting in a simpler and more consistent system.

Each Smallworld object belongs to some class and each class is an object. The class of an object is specified as one of its properties. The classes are organized in a tree. Classes can have any number of properties and may

define methods defining the actions that apply to all members of that class.

A method, which implements some action, applies either to an object or to a class of objects. A method that applies to a single object is called *simple* while a method that applies to all objects in a class is said to be *inherited*. A simple method is represented as a property of the object that it acts upon. On the other hand, an inherited method is represented as a property of the class of objects that it acts upon. Objects that do not have a method for an action, inherit the method from their class. A method is executed in an environment with three string variables initialized: *verb* which is the name of the action being requested, *subject* which is the name of the object being acted upon and *parameters* which are the arguments being passed to the action.

When an action is requested on an object, the corresponding method is first searched in the object, then in the class of the object and finally in the class UNIVERSE. One of the functions of UNIVERSE is to define a set of methods and objects to serve as default definitions for actions. The objects that belong to the class UNIVERSE represent concepts that do not fit the normal object/class structure of Smallworld.

Smallworld allows the users to organize the physical storage of objects into separate databases called *libraries*.

Smallworld was influenced by Smalltalk but there are some differences. Smallworld and Smalltalk differ in their concept of an object. Smalltalk structure is homogeneous since every object is a refinement or instance of another object. On the other hand, Smallworld is heterogeneous. Each object is an independent unit. Each object is a collection of properties and can define its own methods for implementing actions. Objects are grouped into classes for organization and sharing of properties and methods. Another difference is their view of the object-oriented paradigm. In Smalltalk, every operation is treated as an object or method manipulation. Smallworld allows both object-oriented operations and operations from different environments. A final difference is the reliance on sophisticated display technology. Smalltalk, as opposed to Smallworld , depends on bit-mapped displays and pointing devices.

2.6 OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS AND SOME EXAMPLES

An object-oriented database management system is a system that provides database-like support for objects, that is, encapsulated data and operations.

Object-oriented database management systems differ from object-oriented programming languages in that they support [57]:

- persistence
- unique naming (object identifiers)
- sharing
- transactions

GemStone [12] [34] [35] [36] [46] is the only commercial object-oriented database management system. However, there is a lot of research being done and many prototypes have been proposed. Some examples are ORION [5] [6] [10], IRIS [18] [33] which is implemented on top of a relational database management system, EFDM [30] and PDM [37]. IFO [1] [2] is a semantic data model aiming at providing a formal definition of semantic and object-oriented data models.

2.6.1 GEMSTONE

The goal in GemStone [12] [34] [35] [36] [46] is to merge object-oriented programming language technology with database technology. It combines the data type definition and code inheritance of Smalltalk-80 with permanent data storage, multiple concurrent users, transactions and secondary indexes. It supports set calculus, path syntax, time, concurrency, authorization, recovery, replication and directories. It provides a flexible data model, an object-oriented, disk-based storage management system with an object-oriented language OPAL. OPAL is the language for data definition, data manipulation and computation functions of GemStone. It provides built-in identity for all temporary and persistent objects. It solves the impedance mismatch problem and provides unification by combining Smalltalk-80 and a set data type with predicate calculus. The basic idea is to combine programming and database language data types and database transactions.

The user interface of GemStone provides an interactive interface for definition of new data types and execution of queries in OPAL, a procedural interface to conventional languages and a windowing package on which to build user interfaces for applications.

GemStone supports the basic concepts of object-orientation: objects, classes, messages, methods and simple inheritence.

Objects may be atomic (integers, characters etc.) or structured. All structured objects are represented using instance variables, each of which has an object as its value. There are three types of instance variables:

- named instance variables - They are similar to attribute names in relations
- indexed instance variables - They are similar to arrays and are consecutively numbered starting with 1.
- anonymous instance variables - They are used to form collections where only membership is important and order is unimportant.

A class defining object is used to specify the common information for the instances of a class and all instances reference this object that can be the value of an instance variable in any of its instances.

All objects in the system reside in a disk-based object space which is divided into repositories. A repository represents a dismountable partition of the object space and is implemented as a direct access disk file. Repositories are divided into disjoint regions called segments for purposes of authorization and concurrency control. A segment is a chunk of object storage that is owned by a particular user, who can store objects in it and grant access to other users. Segments expand to accommodate the objects stored in them.

Repositories may be replicated on disk against media failures. Replication is used instead of transaction log files. Because repositories of objects are dismounted, a mechanism must be provided to preserve consistent object identity when information is taken off-line and later brought back online.

GemStone's transaction control uses an optimistic approach that gives read-only transactions priority over read-write transactions when they require a commit. The approach is based on the assumption that read-only transactions are more frequent than read-write transactions.

GemStone has two main parts, the executor and the object manager. The executor is responsible for session control. It handles communications between GemStone and host software: receiving blocks of code, returning results and error messages. It maintains a compiler and interpreter for each user. The interpreter is an abstract stack machine that executes compiled methods consisting of sequences of bytecodes. It dispatches bytecodes, performs stack manipulations and some primitive methods and makes calls to the Object Manager. The compiler executes calculus expressions into procedural form.

The Object Manager performs operations related to the storage and access of objects. It handles operations related to concurrency and secondary storage management: transaction control, authorization, data replication, recovery and directory management. In addition, it provides access to different versions of the data.Each user session has its own object manager with a private object space. Sessions have shared access to the permanent database through transactions.

Since GemStone objects retain history, they grow with time so it is not very suitable to use a fixed block of memory to store objects. Objects are implemented based on associations. An element is represented as an element name and a table of associations. The associations are pairs of transaction times and object pointers, each representing that the element acquired the object as its value at the time given by the transaction time. Objects are broken into elements and associations, which are organized into a linked list under header for the object. A directory may be interposed between the object header and the participating elements. Such a directory is useful when an object has a long history or it represents a set whose elements will be accessed associatively. Between objects, pointers to elements are usually physical pointers since most of the data is tree structured. Thus, physical access paths parallel logical access path where objects are not shared. When an object is an element of more than one set, one logical path is chosen as the basis for the physical access path and other references to the object use a global object-oriented pointer (GOOP). The GOOP is resolved through a global object table to get the primary logical path to the object, from which its physical access path can be deduced.

The Object Manager has several subcomponents. The transaction manager is shared by all invocations of the Object Manager and handles concurrent use of the permanent database in an optimistic manner. It records accesses to the database for each session and validates them for consistency

when a transaction commits.The directory manager creates and maintains directories which handle object histories. The Linker incorporates updates made by a transaction in the permanent database at commit time, calling for restructuring of directories as needed. The Linker is called by the Boxer whose job it is to fit objects into tracks after database changes. The track manager schedules reads and writes of tracks. The commit manager provides safe writing for groups of tracks since versions are kept, no garbage collection is needed. Garbage collection for temporary data can be done by discarding the work space at the end of a session.

2.6.2 ORION

ORION [5] [6] [10] is an object-oriented database system prototype being developed at MCC. It adds persistence and sharability of objects created and manipulated in object-oriented applications. The system supports the basic object-oriented concepts such as objects, classes, inheritance and methods. The system is being developed especially for CAD/CAM, artificial intelligence applications and office information systems with multimedia documents. There are two basic requirements for ORION which are advanced functionality and high performance. It supports version control and change notification, storage and presentation of unstructured multimedia data, and dynamic changes to the database schema. For high performance, it supports appropriate access paths and techniques for query processing, buffer management and concurrency control.

Due to the requirements of these application areas, the basic model has been enhanced to support schema evolution, composite objects and version management.

In ORION, the state of an object is represented using instance variables. The values of an instance variable can be restricted to belong to a certain class. However, typeless instance variables are also supported. The behaviour of the object is captured using messages. To reduce redundant storage and specification of objects, shared-value and default-value instance variables are introduced into the model. A value is specified for both type of variables. For a shared-value variable of a class, all instances of the class take on the specified value. These are identical to the class variables described in the previous sections. For a default-value variable, those instances of a class whose value for the instance variable is not specified take on the specified

default value. ORION also supports classes of objects. Each object must belong to a single class.

In ORION, classes and instances are viewed as objects. This is mainly for the uniform handling of messages. Messages are sent to objects and most often to instance objects. To create an instance of a class, a message is sent to the corresponding class object.

ORION was first designed to support simple inheritance but was extended to support multiple inheritance allowing a class to have any number of superclasses. It supports the default conflict resolution scheme in which the property of the first superclass in the immediate superclass list of the class is chosen. Users are allowed to explicitly change the order of the superclasses. ORION, basically, provides three ways in which a user can override the default conflict resolution scheme:

1. The user may explicitly inherit one instance variable or method from among several conflicting ones.
2. The user may explicitly inherit one or more instance variables or methods that have the same name and rename them with the new class definition. ORION ensures that all names inherited or defined within a class are distinct.
3. If conflicting instance variables have default values that are set objects, then one or more of these variables may be inherited under the same name and the default value is another set object which is the union of the inherited default values.

ORION supports the primitive types integer, float, string and boolean as the class Ptype. These can be used as primitive domains of instance variables. Collection and set objects are also supported. All user defined classes are instances of the system defined class Class and it is sufficient to send a message to the class Class to create a new class. The root class is Object.

For each user defined class and for the class Ptype and its subclasses, ORION implicitly defines a Set-of class as a subclass of the Set class [6]. These Set-of classes form a lattice parallel to the class lattice. The Set-of class of a user defined class has two special instances: the set of all instances of the class and the set of all instances of the class and its subclasses. The notion of the Set-of class is especially important for persistent objects. While

a program is executing, objects created by the program can be referenced through symbols that point to them. A program's symbol table provides handles for the objects. However, a newly started program will have no direct references to instances of classes through its symbol table. Instead, the program can refer to the special instances of the Set-of class of the required class. Predicate-based queries are messages to these set objects and return subsets of these sets. Another motivation for the automatic generation of Set-of classes for user defined classes is that instance variables often require values that are sets of objects. Set objects must belong to some class. Without these Set-of classes, the user would have to either explicitly create a class to capture the structure and semantics of these objects or treat them as instances of class Object, losing their semantics.

In ORION, all instances of a class are placed in the same storage segment. A separate segment for each class is allocated automatically. In some cases, especially when dealing with composite objects, multiple classes may be stored in the same segment. The user is required to specify which classes are to be stored in the same segment.

One of the extensions ORION has introduced is schema evolution [5] [6]. ORION supports all the schema evolution operations specified in the previous sections. The most important functions are to add a new class, add an instance variable to a class, delete a class and delete an instance variable from a class.

A new class may be defined as a specialization of an existing class or classes which may be specified as the superclasses of the class. It may redefine some of the instance variables and methods. If there is a conflict the conflict resolution rules previously described are applied.

When an instance variable is added to a class, if there is a conflict with an inherited instance variable, the new variable will override the older definition. All instances of the class will be modified to include the new variable. If the class has any subclasses, they will inherit the new instance variable and if there is a conflict the new variable will be ignored.

Whenever a class is deleted, all its instances are deleted automatically but subclasses of the class are not deleted. The deleted class will be removed from the superclass lists of its subclasses and the subclasses will be assigned the superclasses of the deleted class as superclasses. Also, the subclasses will lose the instance variables and methods they inherited from the class. If

these definitions had overridden some other definitions these definitions will be inherited. If the class to be deleted is the domain of a variable in a class, the superclass of the deleted class will be taken as the domain of the variable unless another domain is specified. When an instance of a class is dropped, all objects that reference it will be referencing a non-existent object. ORION does not automatically identify references to non-existent objects, because of the performance overhead.

When an instance variable is deleted from a class, the class may inherit the instance variable from another superclass if there had been a conflict involving the variable. All subclasses of the class will be affected if they had inherited the variable. Methods involving that variable will become invalid. These methods may be deleted or redefined.

Another schema evolution operation could be the changing of the domain of an instance variable of a class. The domain of an instance variable is always a class and the domain of a variable can only be changed to a superclass of the old domain. Thus, the instances of the class undergoing the change are not affected.

ORION supports composite objects and dependent objects [6]. A composite object consists of a *root object* connected to multiple *dependent objects*. Each dependent object can be a simple object with no dependent objects, a set of objects or the root of a hierarchical structure. In a composite object, the same instance object cannot be referenced more than once so the definition of a composite object is a strict hierarchy of composite objects. All instance objects within a composite object can be referenced by instance objects that do not belong to the composite object.

The instances that constitute a composite object belong to classes and these classes can be organized into a hierachy called a *composite object schema* and a non-root class in this hierarchy is called a *component class*. Each non-leaf class in the hierarchy has some instance variables that serve as *composite links*. These variables are *composite instance variables*. If an instance object is referenced through a composite link, it must be the only composite link to the object but the object may be referenced using other instance variables. Composite links are inherited along the class hierarchy. A composite instance variable may be changed to a non-composite instance variable but the inverse conversion is not allowed since an object can be referenced only by a single composite link and the inverse conversion would require some kind of a reference count mechanism.

A composite object schema is created through composite instance variables which have component classes as their domains. An instance object may be made a part of a composite object only at its creation time. The integrity constraint for composite objects is that any instance object within a composite object cannot be referenced through more than one composite link. Instance objects of a composite object do not contain the identifier of the composite object to which they belong. An instance object which is a dependent object cannot have independent existance. Therefore, if any instance object within a composite object is deleted, it causes a recursive deletion of instances that depend on it. A dependent object remains a dependent object throughout its lifetime unless the related composite link is converted into a non-composite link. The only way in which a composite link can be severed is either by the deletion of a dependent object or by making it a part of some other composite object.

The components of a composite object should be clustered. A composite object can be stored in a sequence of linked pages. If the object increases in size, a new page can be added and if the object decreases in size, pages may be released or compacted. The only problem occurs when two composite objects exchange parts. They should also exchange storage locations. However, ORION does not perform this reclustering.

In ORION, there are two types of versions [6] [10]. A *transient version* can be updated or deleted by the user who created it and a new transient version may be created from an existing transient version. The previous transient version then becomes a *working version.* A working version is stable and cannot be updated, it can be deleted by its owner and a transient version can be derived from a working version. A transient version can be promoted to a working version either explicitly or implicitly.

Since more than one transient version can be derived from a working version, version history is represented in a hierarchy called the *version derivation hierarchy.* Dynamic binding of an object with a versioned object is supported. The user may specify a particular version in the hierarchy as the default version. If a default value is not specified, the system selects the version with the most recent timestamp as the default.

Version handling is quite a performance overhead so versions are only kept on classes which are specified to be *versionable.* A version derivation hierarchy is kept for each instance of a versionable class. A *generic object* is used as the data structure for the version derivation hierarchy.

2.6.3 IFO

IFO [1] [2] is a semantic data model that provides mechanisms for representing structured objects and functional and IS-A relationships between them.

Although IFO is a semantic data model, a brief description of the model is included in the thesis because object-oriented data models and semantic data models are very closely related and IFO aims at providing a theoretical investigation of semantic data modelling issues. It provides a good model for developing object-oriented database schemas. There are four basic issues related to semantic data models. These can be listed as follows [1]:

- Semantic data models are object-based, that is data about objects and relationships between them are represented directly rather than using symbolic identifiers.
- Many relationships between data objects are functional.
- The system must be capable of representing IS-A relationships, that is, it must support subtypes.
- The model should include a mechanism for constructing new object types out of old ones.

One of the most important features of IFO is that IFO schemas and instances are unambiguously and rigorously defined. There is a convenient and simple method for graphically representing an IFO schema. In the graphical representation each distinct data type has a distinct representation. Another feature is that functions can be defined to depend on other functions.

In IFO objects are modelled using *object reps* (object representations). There are two atomic object reps which are *printable objects* corresponding to objects that are alphanumeric strings and *abstract objects* corresponding to objects that have no underlying structure. Similarly, there are two constructors for forming non-atomic object reps. The first is the **-vertex* constructor which forms a set of objects of a given structure type. The second constructor which is the *X-vertex* (cartesian product operator) which forms ordered n-tuples of instances of the children of that vertex. Figure 2.2 gives some examples of the graphical representation of IFO objects. Squares are used to represent printable objects and circles represent abstract objects. Therefore, the first example corresponds to a printable object 'name', the second to an

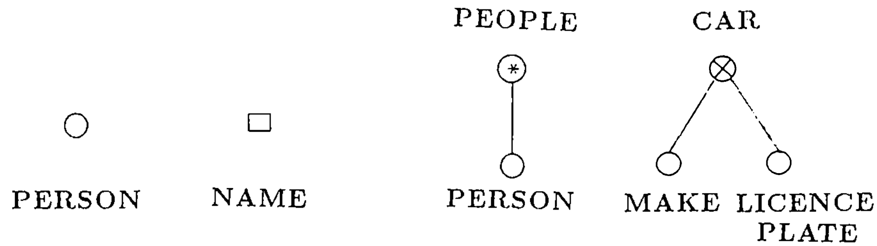


Figure 2.2: The graphical representation of IFO objects, object reps

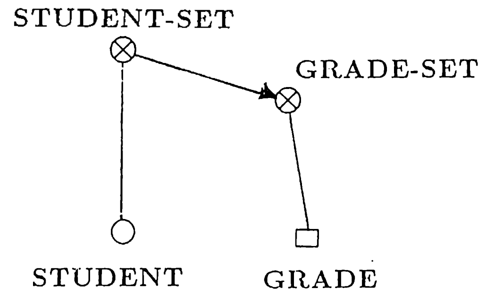


Figure 2.3: An example fragment rep

abstract object 'person', the last two examples correspond to *-vertex and X-vertex constructors.

Fragment reps (fragment representations) are used to represent functional relationships. Functions are relationships between pairs of sets of objects so only *-vertices can participate in functional relationships. The function must be onto and it may be optionally specified as being total or one-to-one. Nesting of functions to arbitrary levels is supported. Figure 2.3 shows the fragment rep corresponding to the function GR which maps a set of students to a set of grades.

The IFO model supports IS-A relationships, that is, subtypes. The basic characteristic of IS-A relationships is the top-down inheritance. In other words, the structure and properties of the superclass are inherited by the subclass. However, IFO supports two kinds of IS-A relationship, namely, *specialization* and *generalization*. For example, the fact that a student is a person is specialization whereas saying that cars and planes are vehicles is generalization. In specialization, the inheritance is from top to bottom. A student is also an instance of the person object and it has the same structure as a person object. On the other hand, in generalization the structure is inherited from bottom upwards. A vehicle is either a car or a plane and a vehicle object may have the structure of either a car or a plane. Figure 2.4 shows the representation of specialization and generalization edges in

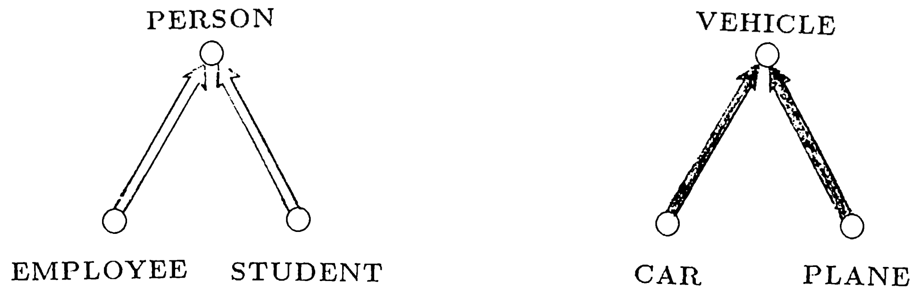


Figure 2.4: The representation of specialization and generalization (IS-A) relationships

IFO. The first example shows specialization edges representing the fact that EMPLOYEE and STUDENT objects are specializations of the PERSON object and they have the same structure. The second example represents the generalization relationships between CAR, PLANE and VEHICLE objects. The VEHICLE object is the generalization of CAR and PLANE objects. The node at the head of two or more generalization edges is *disjoint* if the vertices at the tails of the generalization edges are disjoint. A node at the head of two or more specialization edges is labelled *covers* if the object set of this vertex is equal to the union of the object sets of the vertices at the tails of these specialization edges.

An *object rep* is a directed tree in which each printable vertex and each abstract vertex has no children, each *-vertex has exactly one child and each X-vertex has one or more children ordered from left to right [2]. Therefore, the leaves of the tree are always printable or abstract vertices. A *structured object rep* is an object rep with a specification of the structure types associated with each abstract type. *Extended object reps* are obtained if generalization relationships are replaced by *+-vertices*. Each +-vertex has two or more children where the subtrees below distinct children are different and where each root of the subtrees is not a +-vertex. It acts like a union operator. Figure 2.5 shows an extended object rep. A *structuring assignment* for an object rep is a mapping which assigns an extended object rep to each abstract vertex.

A *fragment rep* is a directed tree R = (V,E) where E is the disjoint union of object edges, E_O, and fragment edges, E_F, (V,E_O) is a forest of object reps, the head and tail of each fragment edge is a *-vertex and the tail of each fragment edge is either the root of R or the child by an object edge

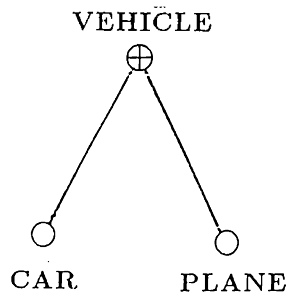


Figure 2.5: An extended object rep

of the head of a different fragment edge. A *structured fragment rep* can be defined similarly to structured object reps.

An *IFO schema* is a forest of fragment and object reps which has any number of specialization and generalization edges and can be represented using an *IFO graph* which obeys some restrictions. From the IFO graph, the *object definition graph* is obtained by reversing the object and specialization edges and taking the union of these with the generalization edges [2]. If the edge (p,q) is in the object definition graph, then vertex p must be defined before vertex q. There should be no directed cycles in the object definition graph. The restrictions which must be satisfied by the IFO graph corresponding to an IFO schema can be listed as follows:

- There should be no directed cycles of IS-A edges in an IFO schema.
- A given type cannot be a subtype via specialization of two fundamentally different types.
- Generalizations result in a set of objects of the generalized type. Generalization edge heads cannot occur in object reps which are the range of some fragment rep. This is because the members of a generalized type must be determined by the constituent subtypes and should not be affected by the behaviour of some function.

If the object structure type of an abstract vertex is not determined by IS-A edges then only unstructured abstract objects can be used to populate it. If two abstract vertices are not explicitly related by IS-A relationships then the set of objects associated with them must be disjoint.

When building a system, the user may start by specifying the printable

and abstract types and then the fragment and IS-A relationships may be added. While specialization edges may be used to restrict the possible set of objects associated with a vertex, generalization edges should be used to create new object sets out of existing ones. For a good design, the head and tail vertices of each generalization edge and the head vertex of each specialization edge should be a primary vertex. A *primary vertex* of an IFO graph is the object-child of a *-vertex which is the root of a maximal fragment rep.

A calculus-based language has been designed for users to perform various operations on an IFO schema.

2.7 CONVENTIONAL VERSUS OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS

Current database systems implement an abstract data type like a relation rather than support natural and easy modelling of real world entities. They hide the complexities of file systems and indexing techniques and provide a degree of physical data independence. The future of database systems will be knowledge management systems with more support for data semantics, inferencing and general purpose programming.

Current systems are limited in both data modelling and programming interfaces [12] [36].

a) Type definition facility. Most database systems supply a fixed set of operations. They do not allow the definition of new types or the addition of operations. The constructors are also limited. The operations for higher-level types are induced by the type constructors and can not be extended. There is no distinction between type definition and data definition causing some redundancy. The aim in O-O DBMS is to support arbitrary levels of data structuring, to allow definitions of operations on types and to uniformly separate type definition from type instantiation.

b) Artificial Restrictions. In conventional systems there are some restrictions imposed by the implementation which must be satisfied by legal database schemes .Some examples are limits on field length, number of fields in a record etc. O-O DBMS try to avoid implementation dependent limits on the sizes of database schemes and data items. The size of data items should only be limited by the secondary storage capacity.

c) Structural Limitations_ The data modelling capabilities of current database systems do not support the complexities and variations seen in real-world problems. There are also restrictions on how the data structuring operations may be applied. Dynamic modification of schemas is not always supported and if supported usually requires database restructuring. In O-O systems variations in structured objects, having arbitrary data items as values, schema evolution without restructuring are supported .

d) Modelling power_ In conventional DBMS real-world problems are encoded into available data structures and they are usually over-simplified, thus the utility and reliability of the data are compromised. When information is encoded, application programs must deal with the encoding and extra integrity constraints are needed to ensure legal encodings. A data model provides entity identity if the data representing any entity can be referenced directly as a unit and the entity may explicitly appear in multiple places in a database without any pointer or other indirection mechanism visible to users. Lack of entity identity leads to inconvenience in modelling and application of constraints. Entity identity also allows easier sharing of data between data items. Object-based models provide entity identity through object identity. Commercial databases do not support a hierarchy of types where as O-O systems support classes, inheritence and a class lattice structure. Another problem with current systems is that update commands are machine-oriented. Changes in the state of the real world involves updates to several database objects. Being able to model real-world changes is a powerful capability for a database system. It can help in choosing implementations for data structures and reduce the overhead in integrity checking, since updates can be made to preserve constraints. Applications become easier to write. O-O DBMS provide powerful data modelling capabilities through flexible data structuring.

e) Access to past states of a database_ A temporal extension to a data model provides historical access for users and an error recovery mechanism. A temporal data model replaces deletions by maintaining object history. Most systems keep a history in the form of checkpoints and recovery logs for error recovery. However, they do not support user access to history. A temporal data model provides both historical access and error recovery. A goal of O-O system is to support version management.

f) Separation of languages_ In most programming languages, persistent and temporary data is treated differently. The persistent data is stored in

files. However files support fewer data structures, so the data to be stored has to be encoded. In databases, data manipulation languages do not support arbitrary computations on database entities, requiring an interface to a general-purpose programming language. One language must be embedded in other. The problem associated with having two languages is impedance mismatch [12] [50] [56]. The mismatch can be conceptual or structural. The conceptual mismatch occurs if the two languages support different programming paradigms, one being declarative and the other procedural whereas the structural mismatch occurs if the two languages support different data types. O-O systems aim at providing a single language for data manipulation, general computation and system commands.

g) The problems related to the data dictionary [56]_ In a database management system, the data dictionary/directory is used to control access to the database, ensure data integrity and supervise the distribution of data. In the past, the data dictionary was a collection of static record structures designed and built after a study of the problem to be modelled. It was fixed throughout the life of database applications. Dictionaries were viewed as static tools for the control of data and information resources. Especially for CAD/CAM and knowledge representation applications, dictionaries are required to be dynamic and active in the design and management of databases. Database design, dictionary definition and data acquisition must be integrated. This brings two features for the dictionary [56]:

    - the need for more dynamic structures capable of evolving over time and with changing requirements
    - a closer integration between data and metadata

    An object-oriented dictionary facility uses an object-oriented organization to represent and describe a data dictionary schema. Objects are used to represent classes and instances of schema structures. All schema related operations are implemented as methods and schema descriptions are maintained as object properties. The methods maintain the consistency of the schema and database objects when the schema is modified.

2.8 ADVANTAGES AND DISADVANTAGES OF THE OBJECT-ORIENTED APPROACH

Object-oriented systems provide major advantages in the production and maintenance of software: shorter development times and a high degree of code sharing. They support versatility, flexibility and a high degree of portability [45]. The problem can be decomposed into subproblems. Systems can be modelled easily since all conceptual entities are modelled using a single concept, that is, objects. These advantages make object-oriented systems an important tool for building complex systems.

One of the most important ideas behind object-orientation is the fact that it crosses a threshold of perception [45]. Working with objects and messages is like working at the human cognition level and provides a high level of abstraction. The body of information and action can be realized as a single unit. This is similar to human perception. Humanbeings perceive the world as being made up of objects and the brain arranges the information into chunks. By using object-orientation, the same idea can be used to solve various problems. Designing a system in terms of objects makes the system easier to understand. The ease of understanding does not actually come from the details of how a procedure is constructed but from not having to consider the other parts of the system. This, in turn, is a result of the data encapsulation and abstraction supported by object-oriented systems. Good design and clean code are also results of using object-oriented systems.

The object-message paradigm and encapsulation tend to promote a more modular system since each message represents a module. Modules are easier to create and understand. In addition, there is a tendency for each message to be a coherent unit. Problems of interfacing modules, which are generally associated with bottom-up development, are absent when working with objects and messages. Method creation can be considered as being bottom-up whereas class creation is top-down. One reason interfacing problems are minimal is that objects are generally passed as arguments in messages and their instance variables do not explicitly appear. Consequently, changes in the structure of an object have no effect on most messages in which the object appears. The support of typeless object variables and dynamic binding add to this property.

Due to the existance of predefined classes and messages, new classes and messages can be defined using them thus eliminating unnecessary details. In

addition, objects can be arguments to messages and since objects are a set of variables, they reduce the number of arguments and hence result in more readable code.

Structural changes can be made to parts of an object-oriented system without the need of extensive compilation and linking. Most changes occur at a high conceptual level and can be translated directly into objects and messages.

Information hiding and data abstraction increase reliability and separate procedural and representational specification from implementation. Dynamic binding increases flexibility by permitting the addition of new classes of objects, that is, new data types without having to modify existing code. Inheritance together with dynamic binding permits code to be reused. This introduces the advantage of reducing overall code and increasing programmer productivity. Inheritance enhances code factoring which means that code to perform a particular task is found in only one place and this eases the task of software maintenance.

One of the most important disadvantages of object-oriented systems is the run-time cost of supporting dynamic binding [45]. A message call costs more than a function call. Actually, messages perform more operations than a function call. Therefore, in some applications the functionality provided by message passing can make the application run faster due to the fact that a message can perform the operations that require multiple function calls. Implementation of object-oriented systems is more complex than comparable conventional systems, since the semantic gap between these languages and the hardware is greater. Therefore, more software simulation is needed. Another possible problem with object-oriented systems is that a user must learn a completely new and different approach and an extensive class library. As a result, object-oriented systems are more dependent on good documentation and development tools.

3. THE OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM PROTOTYPE

3.1 AN OVERVIEW

The object-oriented database management system prototype developed at Bilkent University supports most of the basic concepts of object-orientation. The supported concepts are the concept of an object, object identity, classes, methods, messages and inheritance. As to the extensions, only indexing is supported. The system was implemented using the C programming language [25] [26] and Sun workstations [51] [19] [17] [15] [11] [55] running Berkeley Unix 4.2 BSD [9].

A computationally complete language has been designed and implemented. It serves the data definition, data manipulation and computation aspects of the prototype. The aim is to provide a unified language performing all the operations and solving the impedance mismatch problem. Methods are generally written using the command language. A compiler recognizing the language has been implemented. Methods or commands entered in this language are first compiled into a set of integer codes and then executed. The language is strongly-typed and supports the primitive data types integer and character in addition to supporting collections, sets, arrays and strings. Also, the name of a class in the system is a valid type allowing data types and their related operations to be added to the system at run time. Such newly added classes are treated as any other system defined class. Complex type definitions are not directly allowed but all kinds of objects and types can be defined using class definitions and inheritance.

The system is based on the class model. The objects in the system are grouped into classes and each object belongs to a single class while a class may have many instances.

Everything in the system is considered as an object. Therefore classes are also objects. Thus, the system is homogeneous in its treatment of objects. A class defines the instance variables representing the state of its instances and the messages that represent their protocol. Methods are used to specify how the required operations are to be performed. A class definition can define some class variables or shared values as they are called in the prototype, to represent values shared by all instances of the class. A default value can be specified for each instance variable. A class can be associated with a list of keys. If specified, these keys could be used for indexing operations. Since the language is strongly-typed, a type must be specified for each instance variable and method or message argument. The notion of composite objects or dependent objects is not supported. All objects are passive and independent.

The system uses value and location independent, system generated, unique surrogates to represent object identity. There is a unique surrogate corresponding to every object and this surrogate remains unchanged all through the lifetime of the object. The surrogate is assigned to the object as soon as it is created through instantiation. The surrogate to be assigned is determined by a permanent counter. Only for integer and character objects, the value is encoded into its surrogate. The applied object identity scheme provides both location and data independence, so it supports strong identity in both the representation and temporal dimension.

The system always preserves its consistency with respect to object identifiers. No two distinct objects ever have the same identifier.The dangling identifier problem is also solved.

Each class in the system is associated with a set of methods and messages. The methods can either be written in the designed command language or can be written in C. If a method is written in C, when a message involving the method is invoked, the method call is translated into a C function call and executed as a system call. If the method invoked is written in the command language, it is first compiled and then the resulting set of integer codes is used to execute the necessary operation. Each instance object may have its own set of methods in addition to the methods defined in its class object. This removes the necessity of having a metaclass to support instance objects with their own methods. The system allows the receiver of a message to be a pseudo variable. Calls to self and super are supported allowing recursive methods.

The system allows classes to be specified as being subclasses of previously

defined classes. However, a class may have only one superclass. Therefore simple inheritance is supported and all user and system defined classes form a tree. A subclass inherits all instance variables, shared values, keys, methods and messages from its superclass and inductively, from all classes in its superclass chain. It may redefine and override these inherited properties. It may make use of an inherited method and do some incremental additions to it using the message call to super. Type theory inheritance, external interface inheritance, code sharing and reusability are supported but polymorphism is not supported since generic operations are not allowed. Polymorphism in method and message names is supported.

Each object has its own set of instance variables representing its state. An object is stored as a contiguous block of memory. If an instance variable is of a primitive type, since its value is encoded in its surrogate, its value is stored in the corresponding location. On the other hand, if it is not of a primitive type its surrogate is stored in the corresponding location and the physical address of the object is found from the object table which maps the object identifiers to their corresponding addresses. All objects are treated uniformly, that is, no distinction is made between very large objects and other objects or variable sized objects and fixed size objects.

Reference counting and garbage collection are not performed in the system. Whenever an object is deleted from the system, a flag is set in the corresponding entry of the object table which is used to access objects and which provides a mapping from object identifiers or object-oriented pointers (oops) as they are called in the prototype, and the memory location of the object. This flag indicates that the object is deleted and whenever a reference to that object is made, the system will detect that the object has been deleted by looking at the object table. All objects and methods are persistent.

A different storage mechanism is used in secondary storage and clustering is performed. All references in an object are resolved and are clustered.

The developed prototype being a single-user system does not support concurrent access to objects and authorization control. The notion of a transaction is not supported either.

Some of the basic schema evolution functions such as adding a class and deleting a class are supported. However, versions are not supported since the notion of time is not supported by the system. Indexing is provided on specified key instance variables. Both equality and identity indexes are

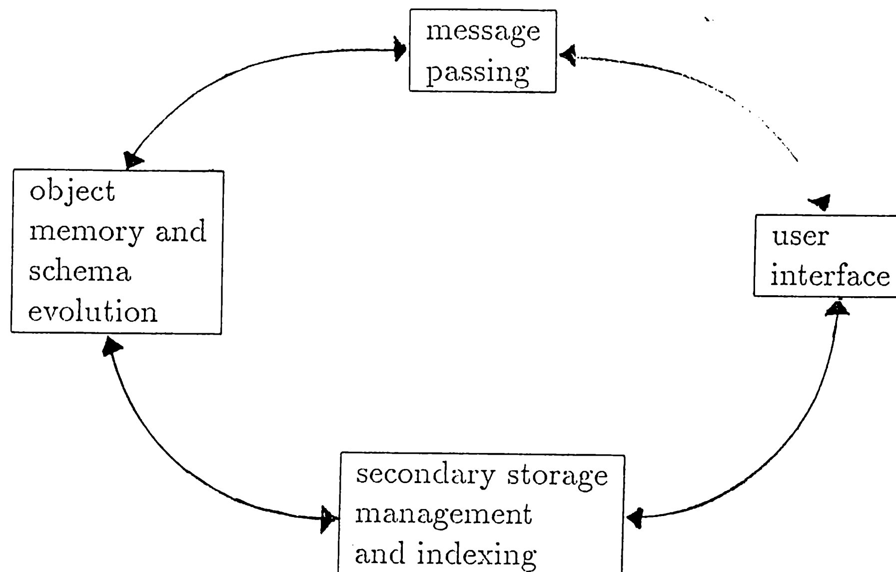


Figure 3.1: The four major modules of the prototype

supported and indexes are created on classes upon user request. Indexes are implemented using B-trees.

3.2 THE MODULES OF THE SYSTEM

The object-oriented database management system prototype designed and implemented at Bilkent University consists of four major modules which are object memory and schema evolution; message passing ; secondary storage management, indexing and the user interface [44]. The user interface is the highest level module. It is built on top of the message passing module which is in turn built on the object memory and schema evolution module. At the lowest level is the secondary storage management module. The four modules are shown in Figure 3.1.

3.2.1 OBJECT MEMORY AND SCHEMA EVOLUTION

Object memory [44] [27] handles the representation, access and manipulation of the objects in the system. Each object is associated with a unique surrogate called *object-oriented pointer* (*oop*). Object-oriented pointers are used to identify objects independently of their values. The message passing module

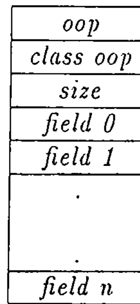


Figure 3.2: The format of an allocated object

and the object memory communicate about objects using object-oriented pointers. An oop is a 32 bit positive even number allowing approximately 2^{30} objects to be referenced. Object memory supports primitive type objects, string objects, class objects, collection objects and instance objects. The primitive type objects are integers and characters. To provide efficiency, the values of the primitive type objects are encoded in their oops.

Object memory uses an object table which maps the oops of the objects to their physical locations in the memory. All references to an object are indirected through the object table. Thus, the oops of the objects are in fact indices into the object table. This indirection provides the benefit of moving the objects easily in the memory. Object memory is implemented as a hash table in which oops are used to provide direct access.

Objects are represented as contiguous series of words. Each word is used to store the value of an instance variable. The actual data of the object are preceded by a header information which includes the oop of the object, the oop of the class to which the object belongs and the size of the allocated space for the object. The format of an allocated object is shown in Figure 3.2. The fields of an object are accessed by zero-relative integer indices.

Classes are themselves objects. The representation of a class object is different from the representation of an instance object. It contains information necessary to construct and use its instances. This information includes the name and oop of the class, oop of its superclass, the number of its instances, the names and definitions of its instance variables, the names of its messages and methods, the domain of the instance variables, and a pointer to the list of its instances. The format of a class object is shown in Figure 3.3.

Classes form a hierarchy, that is each class has only one superclass. The hierarchy is implemented as a tree. There are five basic system defined classes

class oop
class name
super oop
instance count
instance variable count
ptr to variable definitions
ptr to method definitions
ptr to instance variable domains
ptr to the first instance
ptr to the place in the hierarchy tree

Figure 3.3: The format of a class object

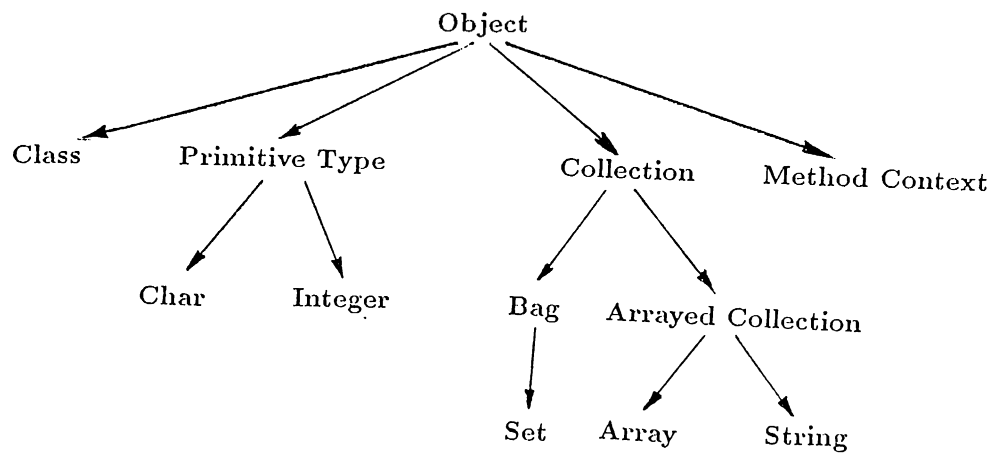


Figure 3.4: The initial class hierarchy and the system defined classes

as shown in Figure 3.4. These are Object class, Class class, Collection class, Primitive Type class and Method Context class. Object class is the root of the hierarchy. The user defined classes are instances of the Class class and they are inserted into the hierarchy when they are created. The information stored in the nodes of the tree includes the oop and name of the class, a pointer to its superclass, a pointer to its subclass list and a pointer to the next sibling in the subclass list of its superclass.

When a new instance of a class is created, a chunk of memory is allocated. This new instance will also be the instance of the superclasses in the hierarchy. Since every class has its own private representation, a separate chunk is allocated for each class in the superclass chain.

The object memory provides the following fundamental functions:

- Determine an object's size, class and implementation
- Access and change the value of an object's instance variable
- Access a class object
- Create a new object

One of the important requirements of database applications is the schema evolution, that is the ability to change the database schema dynamically. In object-oriented databases, there can be changes to the class definitions or to the structure of the class hierarchy. The types of changes include creation and deletion of a class, alteration of inheritance between classes, addition and deletion of instance variables and methods. In the proposed system, only a few of these changes are supported such as adding or deleting a class which is a leaf node in the class hierarchy, adding or deleting instances of a class and adding or deleting an instance variable [44] [27].

The object memory and schema evolution module is approximately 2700 lines long.

3.2.2 MESSAGE PASSING

The message passing module [44] is built on top of the object memory and schema evolution module and forms the basis for the user interface module. It includes the definition and support of the designed command language and error handling in addition to message passing. It consists of five submodules which are the lexical analyzer, parser, code generator, executor module and the query processor.

AN OVERVIEW OF THE COMMAND LANGUAGE

In conventional database management systems, the query language consists of two independent parts: the data definition language and the data manipulation language. Having two separate languages for the two functions introduces the impedance mismatch problem. One of the aims of object-oriented database management systems is to provide a single language handling both

data definition and data manipulation, thus providing unification and solving the impedance mismatch problem.

The command language of the object-oriented database management system prototype is designed to provide unification so it captures both the data definition and data manipulation language aspects. The language can be used both interactively, that is, command by command or in the batch mode, that is, in the form of methods [44].

Message calls which are executed by the executor module have the following format:

< destination >< message name > [<argument list >]

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

METHOD HANDLING AND MESSAGE PASSING

A method is used to access and manipulate objects and is invoked using the corresponding message. A method is created using a method definition statement and is formed of a header and a body. The header contains the method name, the corresponding message name, the name of the class to which the method belongs and a list of optional or mandatory arguments of any system defined type or of any class. The method body is formed of a group of batch mode or interactive mode statements. The method and message name may be the same. All methods are persistent and the code for a method and its compiled form are kept in separate data files [44].

Methods are accessed through a method definition table. Each class object has its own method definition table.

The lexical analyzer, parser and code generator form the compiler for the command language. Every time a new method is created or a method is modified and a compile method statement is executed or each time a message is invoked and the compiled form of the corresponding method is not available, these subroutines are invoked. At the end of the code generation phase, the interactive statement or the method is converted into a set of integer codes

and stored in a file. The executor module takes the generated integer codes as input and performs the corresponding operations using a structure called an *activation record.* During the execution phase, the interactive statements are considered as methods with the necessary arguments for the class Object.

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

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

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

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

3.2.3 SECONDARY STORAGE MANAGEMENT AND INDEXING

Efficient storage and retrieval of objects in the secondary storage constitutes an integral and important part of the prototype implementation [44] [24]. The rest of this section describes the necessary requirements and the actual design preferences.

The secondary storage management and indexing module is approximately 1000 lines.

REQUIREMENTS

The memory system is composed of system defined and user defined persistent objects and temporary objects that are present only during the session and

not accessible to the users. The kernel is composed of the minimum set of system defined objects that are necessary to initialize the system defined tables and hierarchy of system classes. User defined objects and classes will be built upon this kernel and they represent the dynamically changing part of the memory. The secondary storage module must be responsible for managing the transfer of objects between main memory and disk while making sure that the object identity remains unchanged throughout its internal (secondary storage) and external (main memory) representation. The issue of being able to propose a uniform method for handling objects of very different sizes is also very important. The data model presents no problem in this aspect but secondary storage implications are more critical [44].

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

The requirements can be summarized as:

1. Access- Fast random access to objects (and to their chunks) via their oops should be provided; clustering and preloading of objects during disk accesses to attain better performance and providing associative access to an object via value (indexing on value) should be available;

2. Updates and reorganization- Updates that may change the size of objects must be tolerated and stability against relocation should be guaranteed without having to reorganize the whole database in order to avoid unsatisfactory performance.

3. Extensible typing- Schema updates such as class definition updates, addition and deletion of instance variables and class evolutions must be supported in the secondary storage.

STORAGE CONCEPTS AND STRUCTURE

The objects of the main memory data model are mapped to disk objects (called containers) each of which can be viewed as a segment with proper definitions of its class instance variables and super objects [44] [24]. The main objective is to hold together individual chunks of an object contiguously on

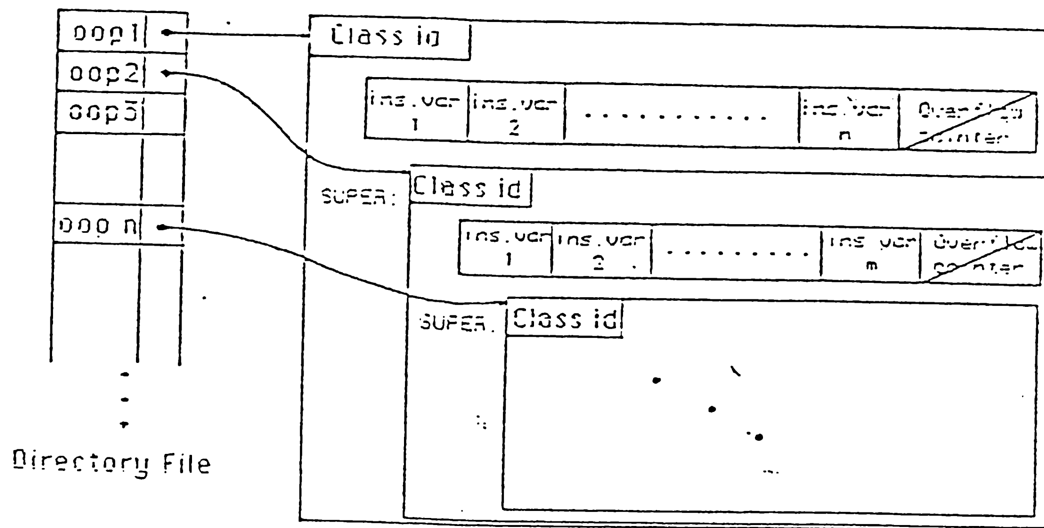


Figure 3.5: The abstract view of a variable sized container

disk which also happens to be the clustering preference of the prototype. It is assumed that retrieving a chunk into main memory would most likely reference to other chunks of the same object due to inheritance and thus retrieving a complex object in its entirety is important for eliminating single chunk disk retrievals. Another partitioning approach such as storing all instances of a class together for clustering would satisfy queries requiring the search of all objects of a class. It is up to the application to determine which access pattern would be more suited. However clustering could be achieved by only one preference and the system's default is storing the objects with its super objects. Another clustering scheme is implemented so as to cluster together collection objects that are values of nested-type instance variables of an object. Objects are stored in disk as a byte stream using Unix low-level file services [9] [51] [19] [17] [15] [11] [55].

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

An abstract view of a variable sized object container is given in Figure 3.5.

A container is recursively defined as a variable sized segment in disk which contains an object's instance variables' values and either the container of its super object's instance or a reference to that container. Resizing a container is possible in two ways; by reorganization or by using an overflow file to keep overflown instance variables. Accessing subcontainers in a container

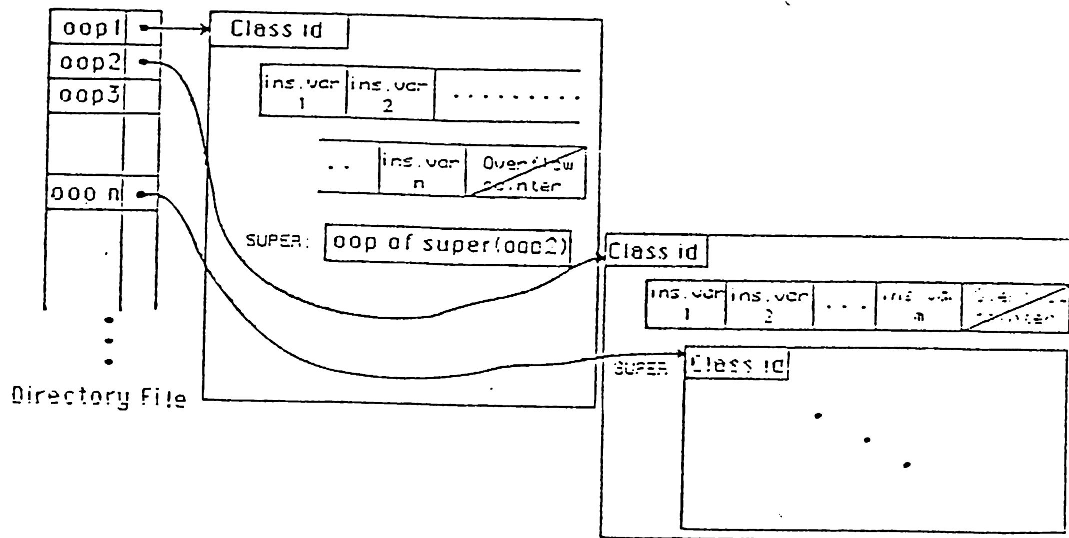


Figure 3.6: The abstract view of a variable sized container with external super-part

is possible via an oop-to-container-address conversion. However once one is in the root container one can make use of physical contiguity and skip the address mapping. If a super object can not be contained in a container due to multiple referencing or identity assignments, then a slot containing the oop of that object is used in resolving the reference (Figure 3.6). The storage manager guarantees that instance values of an object can be found in exactly one container and other references to that object will be redirected to point to this container.

INDEXING

In order to provide alternate access paths to objects, based on the values of their instance variables (i.e. to provide associative access to objects) an indexing module is implemented [44] [24].

Indexing is performed on classes and automatic index maintanence is provided by the system. An index is specified by a path name which is a string of the form $A_1 ... A_n$ where $A_i \in$ user defined classes and A_i is a subclass of A_{i+1} for $i = 1..n-1$ and there does not exist any i such that $A_i =$ CLASS class and the indexed instance variable is among the instance variables of A_n. Indexing a path $A_1 ... A_n$ on the instance variable V will associate the oops of the objects found at class A_1 with the value of V in the corresponding super object.

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

3.2.4 THE USER INTERFACE

The User Interface of the designed prototype [44] [24] is also object-oriented and the user is navigated by a pop-up menu driven system to the operations he/she desires to perform. The User Interface provides three different environments corresponding to three groups of users: (i) developer/maintainer , (ii) domain specialist, (iii) end-user .

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

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

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

3.3 THE NECESSARY STRUCTURES

This section describes the proposed data model and its associated structures. Some parts of the proposed system were not implemented. The previous section described the actual system that was implemented.

The aim of the project was to get an insight on object-oriented systems and object-oriented databases and design a system which supported the basic concepts seen in the object-oriented approach. Among those that are supported in the system one can list the concept of an object which captures both the state and the behaviour of an entity, a class, messages, methods, inheritance and class hierarchy. Although the system is basically memory-based, it provides for the storage, access and clustering of objects in the secondary storage. It also supports an indexing mechanism to avoid the search during

value-based access.

The following structures are needed for the storage,access and manipulation of objects.

1. an instance access table for each class
2. Class table
3. Object table
4. a method definition table for each class and for the necessary instances
5. a hierarchy table

3.3.1 CLASS DEFINITION OBJECT

Each entity is stored as an object. Objects with similar internal representations constitute a class. Every object is an instance of a single class whereas a class may have any number of instances. The class object defines the properties and methods shared by its instances.

A class may define instance variables, shared values, default values, keys and derived variables for its instances. Each instance has its own copy of instance variables. On the other hand, shared values are the properties which all instances of the class share. Default values are similar to instance variables but a default value for the variable is provided so that the instances of the class for which the value of the variable have not been defined take on the default value for the variable. Keys are the properties on which indexes can be built. The order of indexing , that is, whether in ascending or descending order will be specified by the user. Indexes are created only when the user invokes a message for index creation. Derived variables cannot be assigned a direct value. Their value is a function of the other properties of the object. The value corresponding to a derived variable is calculated using the derivation function associated with that value. In the designed system, the derivation functions are treated as methods and evaluated as message passing.

The definition of a class is given in a class definition object. Such an object contains the following information as shown in Figure 3.7:

- class name_ the name of the class being defined

class name
object identifier
organization type
ptr to instance access table
ptr to the corresponding hierarchy object
number of search keys
list of search keys
number of instance variables
a list of istance variable
notify ptr
method definition table ptr
number of instances
number of shared variables
list of shared variables

Figure 3.7: A class definition object

- object identifier_ the oop of the class being defined, since each class is also an object it has an oop.
- organization type_ the organization of the object, that is, whether it is stored as a B-tree or linearly.
- pointer to instance access table entry_ there is an instance access table for each class and it provides a linked list of all instances of the class and domains for the instance variables
- pointer to the corresponding hierarchy object_ a pointer to the hierarchy object which represents the position of the class in the class hierarchy
- number of search keys
- a pointer to the list of search keys_ For each entry in the list of search keys
  - the name of the key
  - the order of indexing (ascending or descending)
  - a flag indicating whether an index has been created on that key or not

  is kept.
- number of instance variables
- a pointer to the list of instance variables_ for each instance variable the following information is stored:

- the instance variable name
- derived or not flag
- unique valued or not flag
- if not derived, type
- if derived pointer to formula definition method
- if it is an indexed type, the maximum size
- if it is an indexed type, the element type
- default value if specified

- notify pointer_ It could be used to support active objects as a trigger.
- method definition table pointer_ Each class has its own method definition table. This table is used to access a method whenever a message is sent to the class.
- number of instances
- number of shared variables
- a pointer to a list of shared variables_ For each shared variable its type and value is stored. A shared value can either be an integer or a character, that is of a primitive type.

This representation solves the problem of derived instance variables. A derived instance variable requires:

- a derived flag
- a formula definition method, that is, the derivation function
- a pointer to the corresponding method

Different approaches have been used for storing the instance variable values for an instance object

- All definitions could be stored as a linked list. Deleting and adding instance variables can be handled as additions to and deletions from a linked list.

number of variable defn
defn 1
defn 2
.   .   .
defn n
ptr to next block

Figure 3.8: Storing an object as contiguous blocks of memory in a linked list

- Instance variables can be stored as a contiguous block of memory. If new instance variables are added a new block is linked to the old block. Each block consists of a count representing the number of instance variable entries in the block, the instance variable entries and a pointer to the next block. Deletions will cause wasted space. Such an organization is shown in Figure 3.8.

- Association lists are used for representing instance variable definitions in class definitions and for representing instance variables in object instances. With this approach it is easier to support temporal aspects. This approach is used in GemStone. Each entry of the association list contains a time and a value denoting that the object had gained that value as its property at the specified time [12].

- The properties of an object is stored as instance variable name and value pairs [22].

In the prototype objects are stored as contiguous words of memory. No garbage collection is applied.Figure 3.9 illustrates the way an instance object is stored in memory.

If an object does not define all instance variables of its class, the corresponding entries will be NIL. The storage for all instance variables will be allocated. If an instance of a class does not define all instance variables of its class and a query is made on the undefined variables, either an error message could be generated or the default values corresponding to the variables, if they exist, could be used.

In the representation used in the prototype, an error can be detected if there is a null value in the corresponding instance variable entry of the object representation.

object ID
length
class defn table entry ptr
method table ptr
notify ptr
instance variable 1
instance variable 2
. .
instance variable n

Figure 3.9: The storage representation of an instance object

Instead of allocating storage for all variables, an approach would be to allocate storage for only the defined instance variables and store pairs in the object definition. However this representation requires restructuring.

3.3.2 METHOD DEFINITION TABLE

There is a method definition table for each class and if necessary for an object instance. If an object has some specific methods of its own these methods will be specified using the method definition pointer in the object definition. This facility eliminates the need for metaclasses.

All methods are persistent and stored in files so the method definition table provides a mapping from message or method names to the files. A method may have any number of arguments and any of the arguments may be defined to be optional. Arguments can be of any type, primitive, indexed, collection, set or an instance of a user defined class.

Each entry of the table corresponds to a method defined for the class or instance and contains the following information:

- method name
- the message name corresponding to the method
- the number of arguments
- a pointer to the list of arguments. For each argument in the list the name of the argument, its type and if it is an indexed type the maximum

oop	free flag	address
.	.	.

Figure 3.10: The object table

length and element type and a flag indicating whether the argument is optional or not is stored.

- the name of the file that contains the method

The default is to have the same name for both the method and its message but this can easily be overridden.

3.3.3 INSTANCE ACCESS TABLE

With the given description, this table provides a collection, in fact a set, of all instances in a class and in a way a domain for each (simple) instance variable. An instance access table is created for each class.

3.3.4 OBJECT TABLE

It provides a mapping between object identifiers and the physical address of the object. Direct access using object identifiers is necessary for this table. In the prototype, hashing is used to perform the direct access on object identifiers. Figure 3.10 shows the structure of the object table.

3.3.5 CLASS HIERARCHY OBJECT

The prototype supports simple inheritance, that is, a class can have only one superclass. Thus, the classes are organized into a class hierarchy in the form

class name	*class definition table entry ptr*	*superclass ptr*	*next subclass ptr*	*subclass link ptr*	*instance access table entry pointer*

Figure 3.11: The format of a class hierarchy object

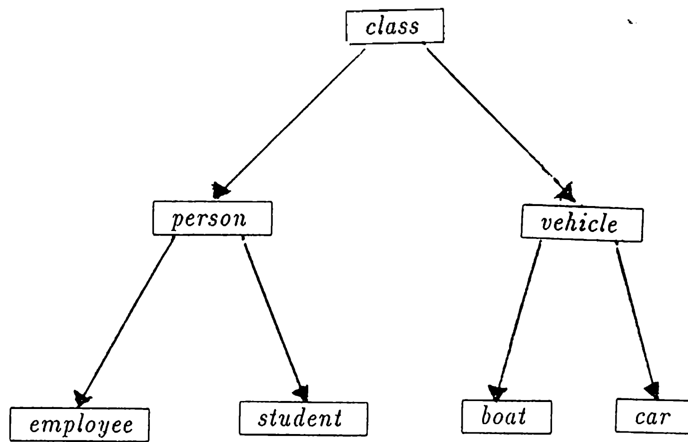
of a tree.

The only kind of conflict that can occur during inheritance, is the conflict between a class and its superclass. This conflict is resolved by giving priority to the class over its superclass. Therefore, whenever a conflict occurs the definition in the class overrides and redefines the definition in its superclass.

The class hierarchy is represented using class hierarchy objects. The format of a class hierarchy object is given in Figure 3.11. As can be seen from the figure, a class hierarchy object contains the following information:

- the class name of the object whose position in the class hierarchy is represented using the class hierarchy object.
- a pointer to the class definition object associated with the class being considered
- the superclass pointer which contains the oop of the superclass of the class
- a pointer to the next subclass in the subclass list of the class's superclass
- a pointer to the first subclass of the class
- a pointer to the instance access table associated with the class

Figure 3.12 provides an example of the representation of the class hierarchy using class hierarchy objects. The first figure is the class hierarchy and the second is the internal representation corresponding to that class hierarchy.



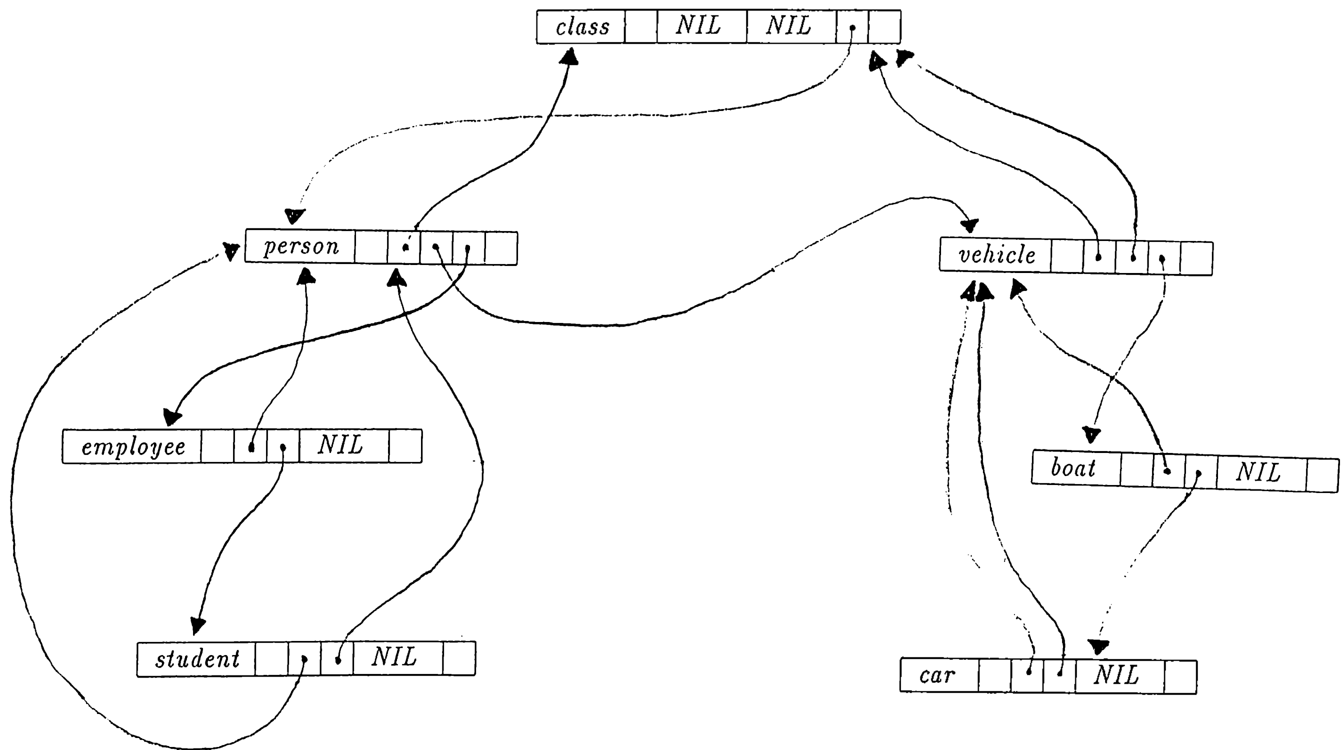


Figure 3.12: An example class hierarchy and its internal representation

4. THE COMMAND LANGUAGE

The basic goal is to solve the impedance mismatch problem between database languages and programming languages by providing a single unified language capturing the power of both languages. The designed language is computationally complete and strongly-typed and supports the following from the computational point of view:

- assignment operator
- relational operators
- aritmetical operators
- logical operators
- conditional constructs (if-then-else)
- looping constructs (while-do)
- declarations (declaration of temporary variables)
- blocking (begin .. end)
- comments
- data types, variables and expressions
- message calls and return statements

The system also supports quite a few database related query statements, data definition and data manipulation statements. Some examples are statements for defining a new class, defining a new method, defining a new instance, modifying the definition of a class or method, accessing instances satisfying certain criteria, equality checks and copying objects.

The language can be used both interactively, that is, command by command or in the batch mode, that is, in the form of methods.

The commands can be classified into two major groups: interactive mode statements and batch mode statements. The interactive mode statements can be further classified as follows [44]:

1. Definition statements

   a. New definitions- One can define a new class, a new method for a class or a new instance of a class.

   b. Redefinition statements- One can redefine an existing class or an existing method of a class.

2. Schema evolution statements

   a. Additions to a class definition- One may add a new instance variable, key, superclass, subclass, shared variable or method to a class. Also, one may add an argument to a method of a class.

   b. Deletions from a class definition- One may delete an instance variable, key, superclass, subclass, shared variable, method or method argument from a class.

   c. Modifications to a class definition- The user may change the default value of an instance variable, the search order of a key, the value or type of a shared variable, the type of an instance variable, the code or argument of a method. Furthermore, one may specify an instance variable as derived and change the function definition, that is, the method used to calculate the value of a derived variable.

   d. Renaming operations- The user may rename an instance variable, method, message or method argument.

   e. Additional changes to a class definition- In addition to the above operations, the user is allowed to replace a key of a class, change the superclass of a class and to make an argument of a method optional or mandatory.

   f. Changes in the class hierarchy- One may rename a class, delete a class or delete an instance from a class.

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

All interactive statements are treated as message calls to the class Object.

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

< destination >< message name > [<argument list >]

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

In the following sections, the language commands, some system defined default values and possible errors are explained giving some examples.

4.1 DATA DEFINITION LANGUAGE

The data definition language supports the following operations:

- define a class
- define a method
- define an instance

* Defining a class

$$define_class < classname > \left[\left\{ \begin{array}{c} persistent \\ temporary \end{array} \right\} \right] [unique][with$$

$$[superclass < classname >;]$$

$$[< number - of - subclasses > subclasses$$

$$< class - name_1 >, ..., < class - name_k >;]$$

$$[< number - of - shared - values > shared_values$$

$$< type >< variable - name_1 >=< value >,$$

$$< type >< variable - name_i >=< value >;]$$

$$[< number - of - instance - variables > properties :$$

$$[unique]\left\{\begin{array}{c} derived \\ < type > \end{array}\right\} < variable - name_1 > [default < value >],$$

$$\vdots$$

$$[unique]\left\{\begin{array}{c} derived \\ < type > \end{array}\right\} < variable - name_n > [default < value >];]$$

$$[< number - of - search - keys > keys :$$

$$\left\{\begin{array}{c} ascending \\ descending \end{array}\right\} < key_1 >,$$

$$\vdots$$

$$\left\{\begin{array}{c} ascending \\ descending \end{array}\right\} < key_m >]].$$

The interface and the methods related to the class must be separately defined. No indexes are created during class definition.

Possible error messages are:

- no such superclass
- no such subclasses
- invalid classnames
- same class already exists
- number of instance values must be an integer >= 0
- number of search keys must be an integer >= 0
- invalid types
- key does not match an instance variable name
- invalid default value

* Defining a method:

$$define_method < methodname > for < classname >$$

$$corresponding - to < message - name >$$

$$with < numberofarguments > arguments$$

$$[optional] < type >< argument_1 >,$$

$$[optional] < type >< argument_n >;$$

$$begin$$

$$\vdots$$

$$< method - body >$$

$$end.$$

Default values:

class name	OBJECT
message-name	method-name
number of arguments	0

Possible error messages :

- invalid method name
- method already exists
- invalid class name
- no such class name
- invalid message name
- message already exists
- number of arguments must be an integer >= 0
- invalid argument name

* Defining a method:

$$define_method < methodname > for < classname >$$

$$corresponding - to < message - name >$$

$$with < numberofarguments > arguments$$

$$[optional] < type >< argument_1 >,$$

$$[optional] < type >< argument_n >;$$

$$begin$$

$$\vdots$$

$$< method - body >$$

$$\vdots$$

$$end.$$

Default values:

```
class name              OBJECT
message-name            method-name
number of arguments     0
```

Possible error messages :

- invalid method name
- method already exists
- invalid class name
- no such class name
- invalid message name
- message already exists
- number of arguments must be an integer >= 0
- invalid argument name

- invalid type specifications
- too few arguments are specified

* Defining a method for a derived instance variable

$$derived_method < method{-}name > for[< class{-}name >] < variable{-}name >;$$

$$begin$$

$$\vdots$$

$$< method - body >$$

$$\vdots$$

$$end.$$

* Defining an instance of a class: (receiver : classname)

$$new < classname > [< variablename >] \left[\left\{ \begin{array}{c} temporary \\ persistent \end{array} \right\} \right].$$

(allocates enough storage but assigns only default values)

$$define < classname > with$$

$$< instance - variable_i >=< value_i >,$$

$$\vdots$$

$$< instance - variable_j >=< value_j >,$$

$$as \left[\left\{ \begin{array}{c} temporary \\ persistent \end{array} \right\} \right] < variablename > .$$

NULL VALUE PROBLEM- If an instance does not define all instance variables, if a method accesses the undefined variables and if the variables have default values the default values will be used, otherwise an error occurs.

Error messages

- invalid class name
- no such class name

- invalid instance variable
- class has no such instance variable
- invalid value for instance value
- invalid variable name

4.2 DATA MANIPULATION STATEMENTS

- change class definition completely
- add instance variable
- delete instance variable
- add search key
- delete search key
- change search key
- change name of instance variable
- change superclass
- change default value of a variable
- redefine method
- change message name
- add argument
- delete argument
- change body
- add method to a class
- delete a method from a class
- rename a method
- change the message name corresponding to a method
- change the search key indexing order

- rename a class
- add superclass
- delete superclass
- delete a class
- change status of variable (derived or not)
- change formula definition method
- change type of variable
- changes to shared variables

* Change class definition completely:

$$redefine_class < classname > \left[\left\{ \begin{array}{c} persistent \\ temporary \end{array} \right\} \right] [unique] with$$

$$superclass < classname >;$$

$$[< number - of - subclasses > subclasses$$

$$< class - name_1 >, ..., < class - name_k >;]$$

$$[< number - of - shared - values > shared_values$$

$$< type >< variable - name_1 >=< value >,$$

$$\vdots$$

$$< type >< variable - name_i >=< value >;]$$

$$[< number - of - instance - variables > properties :$$

$$[unique] \left\{ \begin{array}{c} derived \\ < type > \end{array} \right\} < variable - name_1 > [default < value >],$$

$$[unique] \left\{ \begin{array}{c} derived \\ < type > \end{array} \right\} < variable - name_n > [default < value >];]$$

$$[< number - of - search - keys > keys :$$

$$\left\{ \begin{array}{c} ascending \\ descending \end{array} \right\} < key_1 >,$$

$$\vdots$$

$$\left\{ \begin{array}{c} ascending \\ descending \end{array} \right\} < key_m >]].$$

The formula definition methods must be specified for each derived instance variable. The interface and methods should also be specified. When a class is redefined there are two possibilities: Delete all old instances of the class or give an error message if the class has instances. Default values can come from the previous class definition. No indexes will be created and indexes created for the previous class definition will be lost. If the specified class name does not exist, then the affect will be a new class definition. The error messages are the same as in define class.

* Adding an instance variable to a class:

$$add_to < classname >$$

$$< numberofinstancevariables > properties$$

$$[unique] \left\{ \begin{array}{c} derived \\ < type > \end{array} \right\} < variable - name_1 >$$

$$[default < default - value >]$$

$$[as \left\{ \begin{array}{c} ascending \\ descending \end{array} \right\} key],$$

$$\vdots$$

$$[unique] \left\{ \begin{array}{c} derived \\ < type > \end{array} \right\} < variable - name_i >$$

$$[default < default - value >]$$

$$[as \left\{ \begin{array}{c} ascending \\ descending \end{array} \right\} key].$$

The formula definition methods must be specified for each derived instance variable. New methods and interface may have to be specified.

Error messages are

- invalid class name
- no such class

- invalid class names
- too few variable names specified
- number of instance variables must be >= 1 (default is 1)
- invalid type
- invalid default value for type

* Delete instance variable:

$$delete_from < classname >$$

$$[< numberofinstancevariables > properties]$$

$$< variablename_1 >, .., < variablename_i > .$$

If indexing is available on one of the variables, it will also delete the index tree and delete the variable from the search key list

Error messages

- invalid classname
- invalid variable name
- no such class
- no such variables must be >= 1

* Add search key:

$$add_to < classname >$$

$$< number_of_search_keys > keys$$

$$\left\{ \begin{array}{c} ascending \\ descending \end{array} \right\} < variablename_1 >,$$

$$\left\{ \begin{array}{c} ascending \\ descending \end{array} \right\} < variablename_i > .$$

No index is created. Variables should already be defined in the class and not as keys.

Error messages

- invalid classname
- invalid variable name
- no such class
- no such variables in class
- variable already a key
- number of search keys must be >= 1 (default 1).

* Delete a search key:

$$delete_from < classname >$$

$$< numberofsearchkeys > keys$$

$$< variablename_1 >, .., < variablename_i > .$$

If an index was created, it will be deleted.

Error messages

- invalid classname
- invalid variblename
- no such class
- no such variable name in class
- variable name not a search key for class
- number of keys must be >= 1 (default 1).

* Change search key:

$$replace_in < classname > key < keyname_1 >$$

$$with \left\{ \begin{array}{c} ascending \\ descending \end{array} \right\} key < keyname_2 > .$$

$Keyname_1$ must be a key in class name and $keyname_2$ must not. It will be implemented as a

$$delete_from < classname > keykeyname_1.$$

$$add_to < classname > key \left\{ \begin{array}{c} ascending \\ descending \end{array} \right\} keyname_2.$$

Error messages are similar to those add_to and delete_from

- invalid classname
- invalid variable name
- no such class
- no such instance variables in class
- $keyname_1$ not a key in class
- $keyname_2$ already a key in class

* Rename an instance variable

$$rename_in < classname > property < variable1 >$$

$$as < variable2 > .$$

If variable1 is a search key, then variable2 will be a search key.

Error messages

- invalid classname
- invalid variable name
- no such class

- variable1 not an instance variable in class
- variable2 already exists in class

* Change default value of a variable

$$change_in < classname > default_of < variable >$$
$$to < default_value > .$$

Error messages

- invalid class name
- invalid variable name
- no such class
- no such variable in class
- invalid default value for type of variable

* Change superclass:

$$change_superclass_of < classname > to < superclass > .$$

The necessary inheritance problems should be solved.

Error messages

- invalid classnames
- no such classes
- class name must not be the same as superclass

* Change the search key indexing order

$$change_in < classname > search_order_of < variable > to$$
$$\left\{ \begin{array}{c} ascending \\ descending \end{array} \right\}.$$

If indexing is available for the variable, it will no longer be valid. The new indexing will not be created.

Error messages

- invalid class name
- invalid variable name
- no such class
- no such variable in class
- variable is not a search key for class

* Add superclass

$$add_to < classname > superclass < superclass > .$$

Necessary inheritence problems should be handled. Error messages are the same as for change superclass.

* Delete superclass

$$delete_from < classname > superclass < superclass > .$$

Error messages are the same as for change superclass. The necessary inheritence problems should be handled.

* Add shared variable

$$add_to < classname > shared < type >< variablename >$$

$$withvalue < value > .$$

Shared variables can only be of a simple type.

Error messages

- invalid class name

- invalid type
- invalid variablename
- no such class
- variable already exists in class
- invalid value for specified type

* Delete shared value:

$$delete_from < classname > shared < variablename > .$$

Error messages

- invalid classname
- invalid variable name
- no such class
- no such variable in class
- variable name not a shared variable in class

* Change value of shared variable:

$$change_in < classname > shared < variablename >$$

$$valueas < new_value > .$$

Error messages

- invalid classname
- invalid variablename
- no such class
- no such variable in class
- not a shared variable in class

- invalid value for shared variable

* Change type of shared value:

$$change_in < classname > shared < variablename >$$

$$typeas < newtype > .$$

Error messages are the same as for change value of shared variable.

* Change type of variable:

$$change_in < classname > property < variablename >$$

$$typeas < newtype > .$$

Error messages

- invalid classname
- invalid variablename
- invalid type
- no such class
- no such variable in class
- variable not a simple instance variable

* Change derived instance variable to simple variable:

$$change_in < classname > derived < variable - name >$$

$$to < type > [withdefault_value < value >].$$

Error messages

- invalid classname
- invalid variable name

- invalid type
- invalid value for type
- no such class
- no such variable in class
- variable not derived

* Change simple instance variable to derived:

$$change_in < classname > property < variable_name >$$

$$toderived.$$

The function definition method must be specified.

Error messages

- invalid class name
- invalid variable name
- invalid type
- invalid value for type
- no such class
- no such variable in class
- variable not derived

* Change formula definition method:

$$change_in < classname > functionforderived$$

$$< variable_name >$$

$$begin$$

$$\vdots$$

$$< method - body >$$

$$end.$$

Error messages

- invalid classname
- invalid variable_name
- no such class
- no such variable in class
- not derived variable for class

* Add method to class:

$$add_to < class_name > method < method_name >$$

$$[correspondingto < message_name >][with$$

$$< numberofarguments > arguments$$

$$[optional] < type >< variable_1 >,$$

$$[optional] < type >< variable_n >;]$$

$$begin$$

$$\vdots$$

$$< method - body >$$

$$end.$$

Error messages

- invalid class_name
- invalid method_name
- invalid message_name

- invalid variable_name
- no such class
- method already exists in class
- message already exists in class [default for message_name is the method_name itself]
- number of arguments must be >= 0
- too few arguments are specified
- invalid type

* Delete a method from a class:

$$delete_from < class_name > method < method_name > .$$

Error messages

- invalid class_name
- invalid method_name
- no such class
- no such method as in class

* Rename a method;

$$rename_in < class_name > method < method_1 > as$$
$$< method_2 > .$$

Error messages

- invalid class_name
- invalid method_name
- no such class

- no such method as method_1 in class
- method_2 already exists in class

* Change the message name corresponding to a method

$$rename_in < class_name > message < message_1 >$$

$$as < message_2 > .$$

Error messages

- invalid class_name
- invalid message_name
- no such class
- no such message as message_1 in class
- message_2 already exists in the specified class

* Add an argument to method:

$$add_to < class_name > method < method_name >$$

$$[optional] argument < argument_name >$$

$$oftype < type > .$$

Error messages

- invalid class_name
- invalid method_name
- invalid argument_name
- invalid type
- no such class
- no such method in class

- argument already exists for method

* Rename an argument to a method:

$$rename_in < class_name > method < method_name >$$
$$argument < argument_name_1 > as < argument_name_2 > .$$

Error messages

- invalid class_name / method_name / argument_names
- no such class
- no such method in class
- no such argument as argument_1 in method
- argument_2 already exists for method

* Change type of argument of a method:

$$change_in < class_name > method < method_name >$$
$$argument < argument_name > typeto < type > .$$

Error messages

- invalid class_name / method_name / argument_name / type
- no such class
- no such method in class
- no such argument for method

* Make argument optional / mandatory

$$make < class_name > method < method_name >$$
$$argument < argument_name > \left\{ \begin{array}{c} optional \\ mandatory \end{array} \right\} .$$

Error messages

- invalid class_name / method_name / argument_name
- no such class
- no such method for class
- no such argument for method

* Delete an argument:

$$delete_from < class_name > method < method_name >$$
$$argument < argument_name > .$$

Error messages are the same as those for making an argument optional or mandatory.

* Redefine a method:

$$redefine < method_name > [for < class_name >]$$
$$[correspondingto < message_name >]$$
$$[with < numberofarguments > arguments$$
$$[optional] < type >< argument_1 >,$$
$$[optional] < type >< argument_n >;]$$
$$begin$$
$$\vdots$$
$$< method - body >$$
$$end.$$

Error messages

- invalid method_name / class_name / message_name / argument_name / type
- no such class

- no such method for class (the effect will be the same as a define statement)
- message already exists
- number of arguments must be >= 0
- too few arguments are specified

* Recode a method

$$change_in < class_name > method < method_name >$$

$$codeas$$

$$begin$$

$$\vdots$$

$$< method - body >$$

$$end.$$

Error messages

- invalid class_name / method_name
- no such class
- no such method in class

* Rename a class:

$$rename < class_name_1 > as < class_name_2 > .$$

Error messages

- invalid class_name_1 / class_name_2
- class_name_1 does not exist
- class_name_2 already exists

* Delete a class:

$$delete < class_name > .$$

Error messages

- invalid class_name
- no such class

* Delete an instance of a class:

$$remove_from < class_name >< instanceoop > .$$

Error messages

- invalid class_name
- no such class
- no such object in class

* Checking the existance of an object:

$$exists \left\{ \begin{array}{c} < instanceoop > \\ < classoop > \end{array} \right\} .$$

It returns true if object exists, false otherwise.

* Get the value of a property of an object:

$$retrieve < object >< property_name > .$$

Error messages

- no such object
- invalid property_name

- no such property in object's class

* Set the value of the property of an object:

$$set < object >< property_name > to < value > .$$

Error messages

- no such object
- invalid property_name
- no such property in object's class
- invalid value for property type
- property not primitive (ie. derived or is another object)

* Retrieve the property names of an object:

$$retrieve < object > properties.$$

Error message

- no such object

* List all objects belonging to a class:

$$retrieve < class_name > members.$$

$$retrieve < class_name > membervalues.$$

The first statement retrieves the oops of the instances of the class while the second also retrieves the values. Error message

- invalid / no such class_name

* Get the class of an object:

$$class < object > .$$

Error message

- no such object

* List all property_value pairs of an object:

$$retrieve < object > information.$$

Error message

- no such object

* Copy an object to a file:

$$save < object > [in < filename >].$$

Error message

- no such object

* Copying a property

$$copy_property < object_1 >< property_1 > to$$

$$\left\{ \begin{array}{c} < object_2 > \\ self \end{array} \right\} < property_2 > .$$

Error messages

- no such objects
- invalid property_names

- no such properties in objects
- property_1 cannot be the same as property_2
- property_1 type does not match property_2 type

* Make an instance object the member of another class

$$make_instance < oop >< class_name > .$$

Error messages

- no such object
- no such class

* Display the code for a method:

$$display < object > method < method_name > .$$

Error messages

- no such object
- invalid method_name
- no such method for object

* Copying a method:

$$copy_method < object_1 >< method_1 > to \left\{ \begin{array}{c} < object_2 > \\ self \end{array} \right\} < method_2 > .$$

Error messages are similar to those in copying a property.

* Find the oop of a class

$$find_id < class_name > .$$

Error message

- no such class

* Retrieve the superclass of a class:

$$superclass < class_name > .$$

Error message

- invalid / no such class

* Retrieve the subclasses of a class:

$$subclass < class_name > .$$

Error message

- invalid / no such class

* Display all information about an object

$$status < object > .$$

Error message

- invalid / no such class

* Determining if two objects are identical:

$$identical < object_1 >< object_2 > .$$

It returns true if the two objects are identical, false otherwise.

Error message

- no such objects

* Equality of objects:

$$equal < object_1 >< object_2 > .$$

It returns true if the two objects are equal, false otherwise

Error message

- no such objects

* Copying an object (shallow and deep copy):

$$copy_object < object_1 > to < object_2 > .$$

$$duplicate_object < object_1 > as < object_2 > .$$

The *copy_object* statement forms a shallow copy of an object whereas the *duplicate_object* statement generates a deep copy of the object.

Error message

- no such object as object_1

* Create index

$$index < class_name > on < property >$$
$$in \left\{ \begin{array}{c} ascending \\ descending \end{array} \right\} order.$$

Error messages

- invalid class_name / property
- no such class
- no such property in class

- property not search key in class
- order not correct with respect to class definition

* Delete index:

$$remove_indexon < class_name >< property > .$$

Error messages

- invalid class_name / property
- no such class
- no such property in class
- property not search key in class
- index not available on property in class

* Does index exist:

$$exists_index < class_name >< property > .$$

This command returns true if an index on property in class exists and false otherwise.

Error messages

- invalid class_name / property
- no such class
- no such property in class
- property not search key for class

* Indexing order:

$$order_of_index < class_name >< property > .$$

Error messages

- invalid class_name / property
- no such class
- no such property in class
- property not search key in class

* Change index order:

$$change_index < class_name >< property >$$

$$orderto \left\{ \begin{array}{c} ascending \\ descending \end{array} \right\}.$$

Errors are the same as those for the indexing order statement.

* Check if an instance variable is a key

$$index_allowed < class_name >< instance_variable > .$$

Error messages

- no such class
- no such instance variable defined for the class

* Accessing instances:

$$\left\{ \begin{array}{c} find \\ retrieve \end{array} \right\} \left[\begin{array}{c} first \\ next \\ last \\ all \end{array} \right] < classname > with < condition >$$

$$giving < variable > .$$

Find locates the instance while retrieve will get the values. The $< condition >$ field can either be a relational, arithmetic or logical expression or a query.

* Return statement

$$\mathit{return_value} < \mathit{variable_name} >$$

$$\mathit{return_object} < \mathit{object_oriented_pointer} >$$

* Statements related to method manipulation

$$\mathit{create} < \mathit{class_name} >< \mathit{method_name} > .$$

$$\mathit{modify} < \mathit{class_name} >< \mathit{method_name} > .$$

$$\mathit{compile} < \mathit{class_name} >< \mathit{method_name} > .$$

$$\mathit{execute} < \mathit{class_name} >< \mathit{method_name} >$$

$$[\mathit{with} < \mathit{argument_list} >].$$

* Temporary Variable Declarations:

$$< \mathit{simple} - \mathit{type} >< \mathit{variable_list} > .$$

$$< \mathit{indexed} - \mathit{type} >< \mathit{variable} - \mathit{name} >< \mathit{size} >$$

$$\mathit{of} < \mathit{element} - \mathit{type} > .$$

Other constructs supported are assignment statements, if statements and while statements. Begin..End blocks are used to group related statements.

4.3 SOME EXAMPLES

Considering the following relation scheme in a relational database, the key for all three relations is NAME and SURNAME.

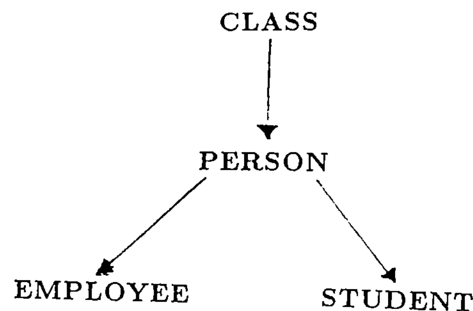


Figure 4.1: The organization of the three classes

PERSON (NAME, SURNAME, AGE)
EMPLOYEE (NAME, SURNAME, COMPANY, SALARY)
STUDENT (NAME, SURNAME, COLLEGE, DEPARTMENT, YEAR)

With this representation there are a lot of inconsistency and redundancy problems. It is very easy to represent such a relation scheme using the classes and instances of object-oriented database management systems. There will be three classes, namely, PERSON, EMPLOYEE and STUDENT with PERSON being the superclass of the other two classes as shown in Figure 4.1. The PERSON class will have three instance variables which are NAME, SURNAME and AGE. The EMPLOYEE class will inherit NAME, SURNAME and AGE from the PERSON class and will have the additional instance variables COMPANY and SALARY. The STUDENT class will inherit the three instance variables defined for PERSON and will add the instance variables COLLEGE, DEPARTMENT and YEAR. The following statements must be executed to create these three classes.

```
define_class PERSON with

    3 properties
        string NAME [10],
        string SURNAME [10],
        integer AGE.

define_class EMPLOYEE with
```

```
    superclass PERSON
    2 properties
        string COMPANY [15],
        integer SALARY.

define_class STUDENT with
    superclass PERSON
    3 properties
        string COLLEGE [20],
        string DEPARTMENT [5],
        integer YEAR;
    key
        ascending DEPARTMENT.
```

For the PERSON class, the superclass is not specified so it is created as a subclass of the system-defined class Class. The DEPARTMENT instance variable is specified as a key for the STUDENT class but the B-tree is not automatically created.

A student may transfer to another department and at the end of each year he will pass if he is successful. The two methods to carry out these operations could be defined as follows:

```
define_method PASS for STUDENT
begin
    integer temp.
    self retrieve_year year.
    year := year + 1.
    self set_year year.
end.

define_method TRANSFER for STUDENT
    with argument
        string NEW_DEPT [5];
begin
```

```
        self set_major NEW_DEPT.
    end.
```

The first method has no arguments but a temporary variable. The temporary variable is needed since some operations on an instance variable are performed. It makes use of two messages of which one returns the value of the YEAR instance variable of the object and the other setting the value of the same instance variable. These will be implemented using the system-defined *retrieve* and *set* statements.

```
    define_method retrieve_year for STUDENT
        with 1 arguments
            integer temp_year;
    begin
        temp_year := retrieve self YEAR.
    end.
```

The second method has a single mandatory argument and it is implemented as another message call with the selector set_major which sets the DEPARTMENT instance variable of the receiver object to the value specified as the argument. The methods sey_major and set_year can be implemented similar to the retrieve_year method. Since a message name has not been specified both will be invoked using the method name. The instance variables of the object receiving a message will be retrieved or modified.

There are two ways of creating an instance of a class. The *new* statement creates a new instance of a class but no values are assigned to the instance variables whereas the *define* statement creates a new instance and also sets the values of the specified instance variables. Figure 4.2 shows the result of executing the following *new* statement and Figure 4.3 shows the result of executing the given *define* statement.

```
    new STUDENT STUDENT1.

    define STUDENT with
```

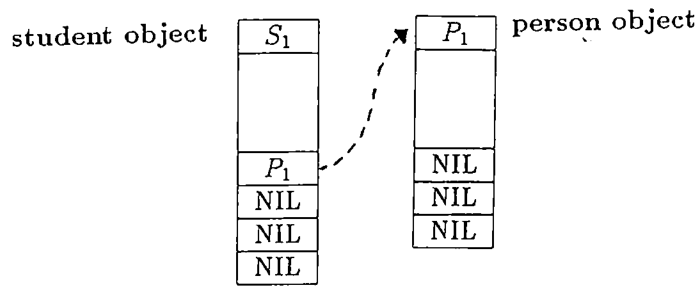


Figure 4.2: The result of executing the *new* statement

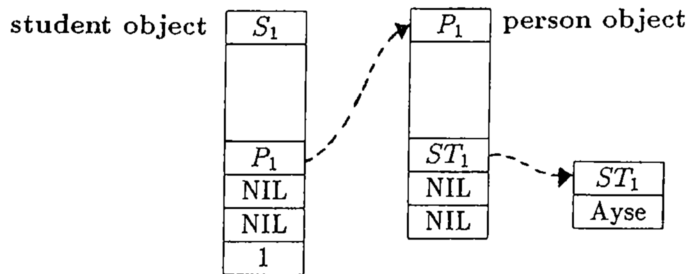


Figure 4.3: The result of executing the *define* statement

NAME = ~Ayse~
YEAR = 1

as STUDENT1.

As explained in the previous section, there are many modifications that can be done to a class definition. One can add new instance variables, shared values, keys and methods to a class and may also delete existing ones. In addition to these some changes can be made to the definition of a method. To add an instance variable STATUS to the class PERSON it is sufficient to execute the following statement:

add_to PERSON 1 properties

integer STATUS default 0.

If an instance variable GPA is added to the class STUDENT, a derived instance variable STANDING whose value depends on the GPA value could be defined.

add_to STUDENT 1 properties

integer GPA default 0.

```
add_to STUDENT 1 properties

      derived STANDING.


derived_method CALCULATE_STANDING for STUDENT STAND-
ING;
begin

      integer temp.
      integer result.
      self retrieve_gpa temp.
      if (temp < 3) true result := 1
      else result := 0.
      self set_standing result.


end.
```

The system supports value-based queries as in other database management systems. Such queries are expressed using the *find* or *retrieve* statements. An example is

```
find all STUDENT with (STANDING = 0).
```

which locates all instances of the class STUDENT with the STANDING instance variable equal to 1. If an index on the STANDING field is available, the requested values can easily be found. Otherwise all instances of the class STUDENT will be searched sequentially. Another example is

```
retrieve all STUDENT with (NAME = ~Ayse~).
```

In this case, since the instance variable name is inherited from the class PERSON, the search will start in the PERSON class. After all the instances of the class PERSON with the specified NAME value are located, the instances of the STUDENT class will be searched to find the ones having that person as their superclass instance. An index could be created from the NAME instance variable to the STUDENT instances. If such an index is created, there is no need for the sequential search. An index can be created using the following statement

index STUDENT on NAME in ascending order.

The specified instance variable must already be defined as a key for the relatedclass. The *index_allowed* statement can be used to determine if an index on a specified instance variable can be created or not.

5. THE MESSAGE PASSING SCHEME

Methods are accessed through a method definition table. Each class object has its own method definition table. Each entry of the table corresponds to a method defined for the class and contains the following information:

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

The message passing module consists of five basic modules: the lexical analyser, parser, code generator, query processor and the executor module. The lexical analyzer, parser and code generator form the compiler for the command language. Every time a new method is created or a method is modified and a compile method statement is executed or each time a message is invoked and the compiled form of the corresponding method is not available, these subroutines are invoked. At the end of the code generation phase, the interactive statement or the method is converted into a set of integer codes and stored in a file. The executor module takes the generated integer codes as input and performs the corresponding operations using a structure called an *activation record.* During the execution phase, the interactive statements are considered as methods with the necessary arguments for the class Object. The query processor handles various associative retrieval queries using the routines provided by the object memory and the indexing modules. Currently, the lexical analyzer and parser have been implemented completely.

Each message returns a fixed size and fixed structure block. This block contains an error flag, a flag indicating whether a value is returned or not,

returned value type, the address of the memory location containing the returned value and for indexed return values the maximum length and the element type.

Methods are stored in data files with an extension '.cl' denoting command language. The output of the lexical analysis phase, that is the tokens of the input method are stored in a file with the same name but extension '.tok' representing token. The parser takes files of tokens as input and if the program represented by the tokens is syntactically correct the intermediate code corresponding to the token is generated and stored in a file with extension '.int'. The code generator generates the actual code corresponding to the intermediate code. The actual code is stored in a file with extension '.com' which is the input for the executor module or the query processor.

5.1 THE LEXICAL ANALYZER

The lexical analyzer submodule can be used to recognize any set of tokens. It is implemented as a deterministic finite state automaton [4] [21] [32]. It takes as input the transition diagram and a set of final states in the transition diagram. The token identifiers corresponding to tokens are also given as input and they are embedded in the transition diagram. The initial state the system will be in is also dependent on the transition diagram and should be determined at run time.

The transition diagram should be input in the following form:

< initial state > < input symbol > < next state >

This represents that once in the initial state, if the read symbol matches the input symbol, the system will move to the state denoted by the next state field. If the initial state field contains a final state, the next state on an empty input gives the token identifier of the recognized token.

At the beginning of the program, the transition diagram and final states are read into memory. The transition diagram is stored in an array structure. The array structure contains an entry for each possible state. Each entry is a pointer to a list of input symbol- next state pairs for each valid combination.

The lexical analyzer performs no error checking. If an input token is not a reserved symbol or keyword then it is checked to see if it is an identifier name

or a constant. if it starts with a letter, it is considered to be an identifier name. On the other hand if it starts with a digit or a minus it is considered to be a numerical constant. In the designed language, string or character constants are delimited by '~'. Therefore, a token starting with a '~' is given a token identifier indicating that it is a string or character constant. In the language, '@' is used to indicate comments. All comments are ignored.

The program handling the lexical analysis function is approximately 500 lines and is written in C.

5.2 THE PARSER

The parser is used to analyze both methods and interactive statements and to generate the associated intermediate code.

The parser submodule implements a deterministic pushdown automaton. The implemented pushdown automaton [4] [21] [32] can be defined as follows: The pushdown automaton is formed of a finite state control and a pushdown store, in fact, a stack. There is a single state and the pushdown automata accepts by empty state. The terminal tokens of the language form the input symbols whereas the nonterminal tokens form the stack symbols. Initially, the start symbol of the grammar corresponding to the language is placed in the stack. Depending on the input character and the character on top of the stack, the production to be applied is selected from among the productions which have the stack symbol as their left-hand side element and the right-hand side tokens of the selected production are pushed on to the stack. If the stack is empty when all input symbols have been processed, the input is a valid expression in the language.

In order to be able to implement a deterministic pushdown automaton the grammar should be unambiguous. In this way, the production to be applied can be detected just by looking at the next input token, thus eliminating the need for backtracking. In order to obtain an unambiguous grammar, left factoring must be applied and left recursion must be eliminated [4] [21] [32].

Left factoring is applied by replacing two productions of the form A $\rightarrow$ BC and A $\rightarrow$ BD by A $\rightarrow$ B$A\prime$ and $A\prime \rightarrow$ C and $A\prime \rightarrow$ D. When eliminating left recursion, the productions A $\rightarrow$ AB and A $\rightarrow$ C are replaced by A $\rightarrow$ C$A\prime$ and $A\prime \rightarrow$ B$A\prime$ and $A\prime \rightarrow \epsilon$ [4] [21] [32].

After applying left factoring and eliminating left recursion unambiguous leftmost derivation can be applied.

Most syntax checking must be done during the parsing phase. Other than spelling checks, a string constant may exceed the maximum string length, a variable name may exceed the maximum variable name length or a numeric constant may exceed the system limits. Boundary and index checking should be performed.

Error handling is a difficult task. Typical errors are:

- the insertion of an extraneous character or token
- the deletion of a required character or token
- the replacement of a correct character or token by an incorrect character or token
- the transposition of two adjacent characters or token

These are syntactic errors. There are also semantic errors that can be detected at compile time. These are errors of declaration and scope. They include undeclared or multiply declared identifiers and type incompatability in various operations. Some other errors can be detected at run time or at compile time. This is the case for range checking, exceeding system limits and exceeding the declared array or string bounds.

When an error occurs, it is up to the syntax analyzer to decide what action to take. A token might be missing or misspelled or the user might have put some extra characters or tokens and the sought token might be further along in the string.

Some approaches to error handling in the syntax analysis phase are [4]:

- The syntax analyzer will stop parsing the input when it detects an error. This is the simplest approach to implement but is not user-friendly.
- Panic mode_ When an error occurs, all input symbols are discarded until a synchronizing character, usually a statement delimeter, such as a semicolon is encountered. The parser then deletes stack entries until it finds an entry such that it can continue parsing given the synchronizing token as the input. This approach is simple and can never result in an infinite loop.

- Minimum Hamming Distance Method_ A program is said to have k errors if the minimum number of error transformations that will map any valid program into the program with k errors is k. The minimum Hamming distance is the least number of insertions, deletions and symbol modifications necessary to transform one string into another. Although the approach is quite complex a simple heuristic based on the assumption that most spelling errors result from one application of an error transformation such as inserting an extra character, deleting a character, modifying a character or transposing two adjacent characters has been developed. The strategy is to check whether any keyword can be transformed into the erroneous string by a single error transformation. For example, one can eliminate as candidates words whose length differ from that of the erroneous string by more than one.

The implemented parser is approximately 3500 lines long and written in C. It takes as input a list of tokens in a file with extension '.tok' and produces the corresponding intermediate code if the tokens correspond to a valid program. The intermediate code is stored in a file with extension '.int'. The parser also takes as input the unambiguous productions for the language and a list of nonterminals. Therefore, similar to the lexical analyzer it can be used for any language.

The productions of the language are read from an input file and then stored in the structure shown in Figure 5.1. The map array provides a mapping from nonterminal tokens to an identifier which is used as an index into the production array. The production array has an entry corresponding to each nonterminal token. This entry is a pointer to a list of productions. Each member of the production list corresponds to a production with the associated nonterminal token as its left-hand side and contains a pointer to a list of tokens. This list of tokens represents the right-hand side of the production and contains an entry for each token. An example can be seen in Figure 5.2 which shows a list of productions and their internal implementation.

Error messages and the intermediate codes to be generated are embedded in the production rules and are also input at the beginning of the syntax analysis phase. Each input production must have the following format:

```
<left-hand side token> | <storage-flag> <intermediate-code>
                          <list of right-hand side tokens> $
```

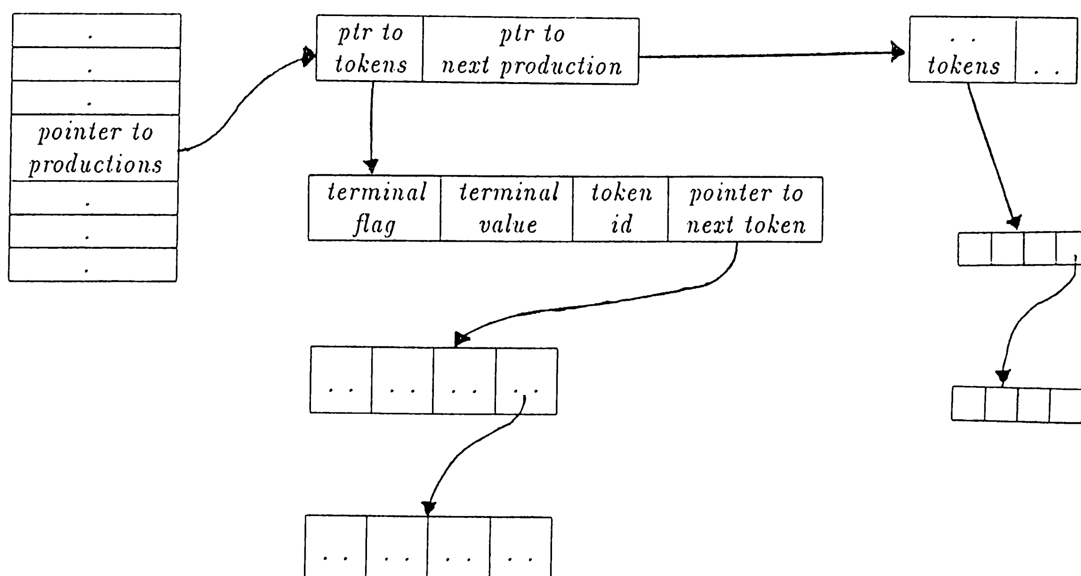


Figure 5.1: The internal representation of production rules

When a production is being applied, the corresponding intermediate code is stored. However if the intermediate code field of the production is zero no intermediate code is generated for that production. The storage flag is related to multivalued tokens such as variables, numerical constants or character or string constants. If the flag is 0 only the intermediate code corresponding to the production is stored if specified. If the flag is 1 then the actual values corresponding to the multivalued tokens are stored together with the specified intermediate code.

The list of right-hand side tokens is formed of members which represent each token and the corresponding error messages. Each member has the following format:

```
<token value> <error-condition-1> <error-code-1>
                           .
                           .
              <error-condition-n> <error-code-n>
```

The error condition may be token missing or misspelled and the corresponding

$S \rightarrow aA$
$S \rightarrow bB$
$S \rightarrow a$
$A \rightarrow 1$
$B \rightarrow A1$
$B \rightarrow S$

MAP_ARRAY

	.	.
i	S	1
	.	.
j	A	2
	.	.
k	B	3
	.	.
	.	.

PRODUCTION_ARRAY

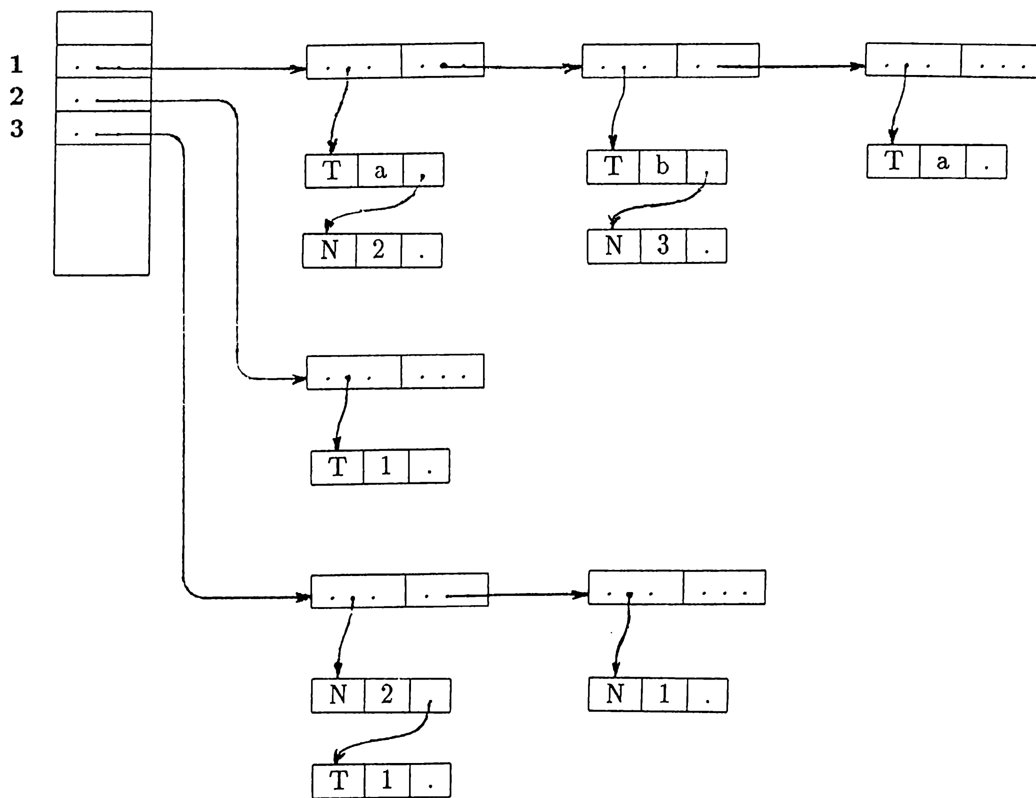


Figure 5.2: An example for the internal representation of productions

line count	*token value*	*error code*

Figure 5.3: An error entry

error code	*error description*

Figure 5.4: The record structure of the error file

error code determines the error message to be generated.

In the productions a differentiation should be made between single valued tokens and multivalued tokens. Multivalued tokens starting with '$\wedge$' denote variable names, a '@' as the first character denotes arithmetic constants and a '$\sim$' denotes a token which may be a character or string constant.

The error-handling approach used, applies the minimum Hamming distance heuristic and tries to detect all syntactic errors and declaration related errors. The minimum Hamming distance heuristic is applied when a keyword is expected but not found or when no production to be applied can be found for the input string. As an incorrect method or interactive statement is being parsed a list of the errors that are detected is maintained and after the input has been processed an error report is generated using an error file to determine the actual error messages. Figure 5.3 shows the internal representation of an erroneous token and Figure 5.4 shows the format of the error file.

All temporary variables are stored in a symbol table. The symbol table maintains the name of the variable, its type, if it is an indexed type its maximum size and element type and a usage flag representing whether the variable is an identifier or constant etc. for each variable. The symbol table is stored in a file with extension '.sym' if the input is syntactically correct. Symbol tables are only generated for methods and no symbol table is created

for an interactive statement.

When a token is input and the production to be applied is sought, first the productions starting with a terminal value are checked. If a match is not found the nonterminal productions and then the multivalued productions are checked. If there is still no match, the productions are tried again checking for the minimum Hamming distance criteria.

5.3 THE CODE GENERATOR

It takes as input the intermediate code generated by the parser module and generates the actual code. Unlike the first two mudules, this module is language specific.

The actual code is a set of integer codes representing some primitive operations. The primitive operations include system calls for C function calls, message calls, branching to an address, conditional checks, relational operators, arithmetic operators, logical operators and assignments. During the translation phase, constructs like while statements in the language are converted into a sequence of primitive operations

5.4 THE EXECUTOR MODULE

The executor module is the most important submodule of the message passing module. It handles the actual message passing operation. It takes as input the code obtained after the compilation of a method or interactive statement and performs the operations required.

Initially, message passing was going to be implemented as C function calls and the language of the system was going to be C and all methods were going to be written in C. Since this would result in an unreliable system, a command language was developed and a message passing scheme was proposed.

For uniformity, at the execution level each method returns a fixed size and fixed structure block which contains the following information:

- an error flag
- a flag indicating whether a value is returned or not

- returned value type
- the maximum size of an indexed type
- the element type of an indexed type
- a pointer to the returned value

The basic structure for the executor module is the activation record. Each method is represented using an activation record and message passing is also implemented using activation records. Since interactive commands are also treated like methods, they are also associated with an activation record during the execution phase.

An activation record contains the following information:

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

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

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

The activation record and other related structures corresponding to the following method segment are shown in Figure 5.5.

```
begin
    integer temp.
    self retrieve_year temp.
    temp := temp + 1.
    self set_year temp.
end.
```

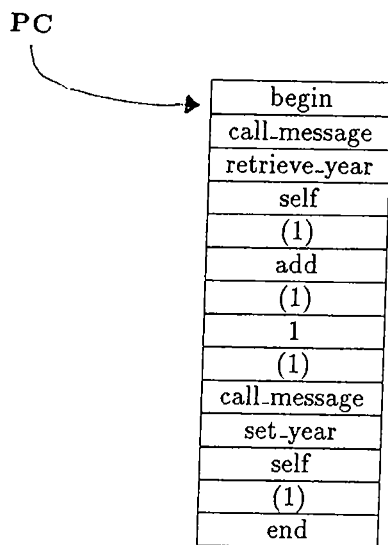
The internal representation of *if* and *while* statements are given in Figure 5.6 and Figure 5.7 respectively. They correspond to the following program segments, for the *if* statement:

```
if (temp < 4) true temp := temp + 1
else temp := 0.
```

and for the *while* statement

```
while ((temp > 0) and (temp < 4))
begin
    temp := temp + 1.
    i := i + 1.
end.
```

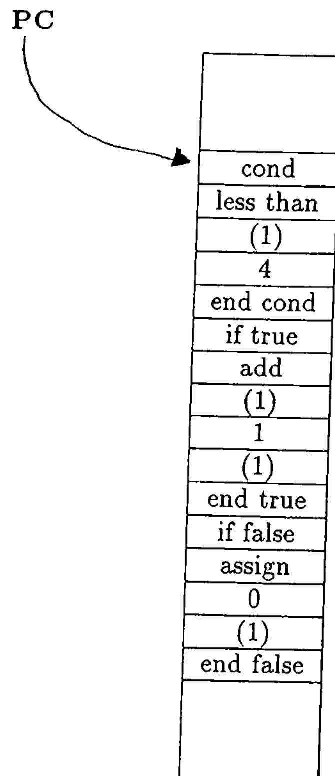
In the system, all operations are performed by defining classes and their interface and invoking messages. All interactive statements are also treated



Symbol Table

temp	integer	. . .

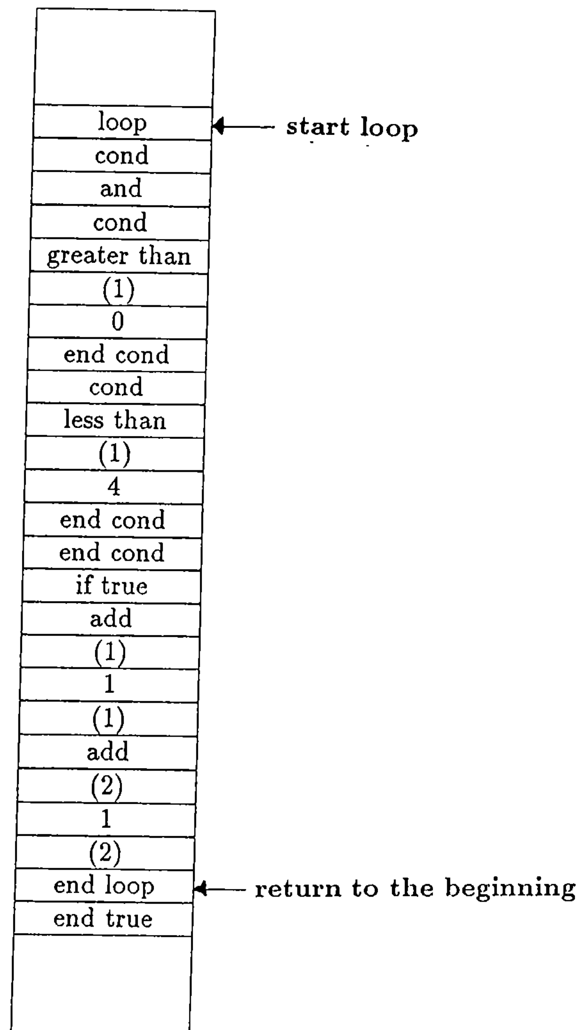
Figure 5.5: The internal representation of an example method



Symbol Table

temp	integer	. . .

Figure 5.6: The internal representation of an *if* statement



Symbol Table

temp	integer	. . .
i	integer	. . .

Figure 5.7: The internal representation of a *while* statement

as messages. When a message is invoked, first of all the method definition table of the receiver class is checked. If the table does not contain an entry corresponding to the message, the search is continued in the superclass of the class. If no corresponding entry can be found in the hierarchy, an error message is generated indicating that either the message name is misspelled or such a message does not exist in the system. Once an entry corresponding to the message is found in a method definition table the specified arguments and the required arguments are checked. If there is no error, the method associated with the message and the file of the name containing the message are obtained from the table. If the compiled form of the method exists, that is, there is a file with the specified name and with extension '.com', the executor module will generate the activation record corresponding to the method and perform the specified operations. Otherwise, the method will first be compiled and then executed.

Within the activation record, the code to be executed will be input from the associated file with extension '.com'. The program counter determines which statement is to be executed. The execution is performed sequentially unless a branch or a message or system call is executed. System calls, that is, C function calls are usually used to reference the functions provided by the object memory and schema evolution module.

5.5 THE QUERY PROCESSOR

The query processor handles various associative retrieval queries using the routines provided by the object memory and the indexing modules. It is in a way embedded in the executor module.

Associative queries are usually performed on collections. When an associative query is requested, the query processor will check if there is a related index. If there is the routines provided by the index manager will be used to perform the necessary query. If a related index does not exist, a sequential search will be performed using the functions provided by the object memory and schema evolution module.

6. OPEN PROBLEMS AND FUTURE EXTENSIONS

The problems related to the object-oriented approach in general can be listed as follows [56]:

- Performance_ The problem with most object-oriented systems is related to the fact that they are rather slow. The systems could be made faster by using compilation instead of interpretation. Also, dynamic binding and full overloading may have to be resolved at run time. This causes some additional overhead.
- Overloading_ In most systems, the operation type is determined by the first operand even when there are multiple arguments. This causes an unfairness problem. Overloading should be over all arguments rather than only the first one.
- Higher order_ Everything is first-order. Defining a flexible system allowing functional types as arguments to a function is the problem. The problem becomes even more complex if polymorphic types (parameterized types) are allowed.
- Parameterized types_ A type inference mechanism is used to infer most general types of arguments to functions at compile time. This gives run time efficiency by eliminating the need for run time type checking. The problem appears for partially ordered types.

Little has been done to define formal semantics of object-oriented models of computation.

Some problems related to object-oriented database management systems are [43]:

- No standard data model for object-oriented database management systems
- No standard guidelines for designing them
- No standard query language
- Should queries on attributes which is against the concept of encapsulation be allowed?
- Should they be considered as repositories for persistent objects or as providing a complete picture of executing applications?
- Should active objects be viewed as executing within the database or should running applications be viewed as being explicitly outside the database?

The problems related to the evolution of the software base [43]:

- Providing tools to maintain global consistency. When changes are made to the software base, the changes must be properly distributed. The management of evolution is especially important for inheritance and subtyping. As long as the interface to an object class is not modified, its realization may be modified. When the interface changes, there is the problem of invalidated references between object classes.
- Assuring that the right object is in the software base. Whenever a referenced object cannot be found in the software base, either a new object has to be added to the software base or an existing object has to be modified.

Other problem areas of the object-oriented approach and object-oriented database management systems are version control, manipulation of composite or dependent objects, schema evolution and handling conflicts in the case of multiple inheritance. The use of object identity requires a sequential search during associative access unless some kind of mapping or indexing is provided, thus degrading system performance. Index handling in object-oriented database management systems is a very important research area. Another problem associated with the object identity concept is the preservation of object identity consistency. Some other open problems related to object-oriented database management systems are garbage collection, storage management and especially the storage of variable-size or very large objects and

clustering. Also, there is a great demand for a theoretical model and some standards for the object-oriented approach.

The object-oriented database management system prototype developed and implemented at Bilkent University supports the basic object-oriented concepts such as object identity, classes, inheritance and message passing but there are some open problems.

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

The system does not support versions. In order to be able represent the temporal aspects of the data, the basic storage scheme used for object instances has to be modified. Instead of a value, a value and time pair must be stored for each instance variable. Versions introduce an overhead from the storage point of view but they eliminate the need for garbage collection since all data is kept in the form of versions.

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

Some other open problem areas for object-oriented database management systems are indexing, version management and composite object handling. A composite object is a complex object formed of a set of subobjects that are treated as units of storage, retrieval and integrity checking. The existance of the subobjects depends on the existance of the principle object. Composite objects represent the IS-PART-OF relationship between objects. The indexing problem is introduced by the use of a location and value independent surrogate to reference an object. The problem arises during the value-based access of objects.

7. CONCLUSION

A combination of the object-oriented language capabilities with the storage management functions of a conventional database management system results in reduced application development efforts. The flexible data modelling capabilities allow the representation of information not suited for normalized relations. Also, an object-oriented language is complete enough to handle database design, access and applications.

The major advantages of the object-oriented approach are versatility, flexibility, reusability, implementation independence and increased programmer productivity. Also, since duplication and redundancy are reduced data integrity is automatically satisfied. The main disadvantages are the relatively poor performance and the complexity of implementing such a system. This is due to the lack of a theoretical model and other basic standards for object-oriented systems. In addition, object-oriented systems require a new and different approach to problem-solving.

The main problem areas of the object-oriented approach and object-oriented database management systems are version control, manipulation of composite or dependent objects, schema evolution and handling conflicts in the case of multiple inheritance. Unless some kind of indexing or mapping is provided, the use of object identity requires a sequential search during value-based access and this degrades system performance. A very important research area related to object-oriented database management systems is index handling. The preservation of object identity consistency is another problem associated with the object identity concept. Garbage collection, storage management and especially the storage of variable-size or very large objects and clustering are some other open problems related to object-oriented database management systems. Also, there is a great demand for a theoretical model and some standards for the object-oriented approach.

The basic object-oriented concepts such as object identity, classes, inheritance and message passing are supported by the object-oriented database management system prototype developed and implemented at Bilkent University. There are some open problems such as schema evolution and multiple inheritance but the main aim is to gain an insight on the subject and provide a basis for future research.

The implemented prototype is a single-user system so it may be extended to support multiple users. This requires the addition of the transaction concept, authorization control, concurrency control and data integrity checks. The system does not support versions. The model could be extended to capture the temporal aspects of the data. Another extension could be the addition of composite objects and dependent objects to the model. Currently, the system only supports passive objects. Active objects are also an interesting research area. The support of all schema evolution functions could be added to the system. Work could also be done to support multiple inheritance instead of simple inheritance and efficient techniques.

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

A. LIST OF BASIC ROUTINES

A.1 THE LEXICAL ANALYZER

printfile(filename)
punctuation(input-chr)
delimeter(input-chr)
arithmetic_operator(input-chr)
relational_operator(input-chr)
assignment(char1,char2)
character(input-chr)
digit(input-chr)
parse_method(input-file,output-file)
create_parse_array()
create_final_states()
reached_final_state(current-state)
find_next_state(current-state,input-char)
parse_string(input-str)

A.2 THE PARSER

initialize_parse_stack(top,full,empty)
push_token(top,value,full,empty)
pop_token(top,full,empty)
stack_top(top)
terminal_stack_top(top)
keyword_stack_top(top)
variable_stack_top(top)
constant_stack_top(top)

string_stack_top(top)
display_stack(top)
initialize_temp_stack(top,full,empty)
push_temp_stack(top,value,full,empty)
pop_temp_stack(top,full,empty)
display_temp_stack(top)
create_map_array()
compare(string1,string2)
copy(string1,string2)
length(str)
token_id(str)
initialize_production_array()
alloc_token()
alloc_production()
attach_next_production(map-array-index)
attach_intermediate_code_id(map-index,value)
attach_token_storage_flag(map-index,value)
attach_token(map-index,str)
attach_error_condition(map-index,value)
attach_error_code(map-index,value)
read_error_values(index)
create_production_array()
display_production_array()
character(input-char)
digit(input-char)
check_variable_name(str)
check_numeric_constant(str)
check_string_constant(str)
check_terminal_token(input-token)
find_error_code(input-token)
store_error(mode,error-count,input-token,error-no)
find_token_value(index)
data_type(token-ptr)
indexed_data_type(token-ptr)
allocate_symbol_table_entry()
add_production_to_symbol_table(production-ptr)
store_production(pr-index,token-ptr)
add_variable_to_symbol_table(input-token)
store_correspondence(input-token,stack-ptr)
convert_to_integer(str)

read_token(input-file,input-token)
match_string(string1,string2)
ordered_string_matching(string1,string2)
minimum_hamming_distance(string1,string2)
try_minimum_hamming_distance()
process_terminal_stack_top(top,input-token,error-count)
multivalued_token(input-token)
search_terminal_productions(mode,in-token,pr-index)
search_nonterminal_productions(mode,in-token,pr-index)
search_terminal_multivalued_productions(mode,in-token,pr-index)

search_nonterminal_multivalued_productions(mode,in-token,pr-index)

search_approximate_terminal_productions(mode,in-token,pr-index)

search_appproximate_nonterminal_productions(mode,in-token,pr-index)

find_next_production(in-token,flag)
push_production(new-production)
search_other_stack_entries(top,in-token,error-count,replace-flag)
process_nonterminal_stack_top(top,in-token,error-count)
process_token(in-token,error-count)
display_error_message(mode,error-no,error-message)
show_errors(mode,error-count,filename)
display_correspondence()
display_symbol_table(symbol-count)
display_parse_tree()
copy_production(token-ptr)
initialize_token_list()
display_token_list(token-list-ptr)
append_token_list(head,tail,token-ptr)
delete_token_list(head,token-ptr)
find_nonterminal_token(head)
find_associated_production(token-ptr)
find_corresponding_entry(str)
replace_corresponding_tokens(token-ptr)
generate_intermediate_code()
store_intermediate_code(filename)
store_symbol_table(symbol-count,filename)
parse_method(argument-count,arguments)

read_token(input-file,input-token)
match_string(string1,string2)
ordered_string_matching(string1,string2)
minimum_hamming_distance(string1,string2)
try_minimum_hamming_distance()
process_terminal_stack_top(top,input-token,error-count)
multivalued_token(input-token)
search_terminal_productions(mode,in-token,pr-index)
search_nonterminal_productions(mode,in-token,pr-index)
search_terminal_multivalued_productions(mode,in-token,pr-index)

search_nonterminal_multivalued_productions(mode,in-token,pr-index)

search_approximate_terminal_productions(mode,in-token,pr-index)

search_appproximate_nonterminal_productions(mode,in-token,pr-index)

find_next_production(in-token,flag)
push_production(new-production)
search_other_stack_entries(top,in-token,error-count,replace-flag)
process_nonterminal_stack_top(top,in-token,error-count)
process_token(in-token,error-count)
display_error_message(mode,error-no,error-message)
show_errors(mode,error-count,filename)
display_correspondence()
display_symbol_table(symbol-count)
display_parse_tree()
copy_production(token-ptr)
initialize_token_list()
display_token_list(token-list-ptr)
append_token_list(head,tail,token-ptr)
delete_token_list(head,token-ptr)
find_nonterminal_token(head)
find_associated_production(token-ptr)
find_corresponding_entry(str)
replace_corresponding_tokens(token-ptr)
generate_intermediate_code()
store_intermediate_code(filename)
store_symbol_table(symbol-count,filename)
parse_method(argument-count,arguments)

```
read_token(input-file,input-token)
match_string(string1,string2)
ordered_string_matching(string1,string2)
minimum_hamming_distance(string1,string2)
try_minimum_hamming_distance()
process_terminal_stack_top(top,input-token,error-count)
multivalued_token(input-token)
search_terminal_productions(mode,in-token,pr-index)
search_nonterminal_productions(mode,in-token,pr-index)
search_terminal_multivalued_productions(mode,in-token,pr-index)

search_nonterminal_multivalued_productions(mode,in-token,pr-index)

search_approximate_terminal_productions(mode,in-token,pr-index)

search_appproximate_nonterminal_productions(mode,in-token,pr-index)

find_next_production(in-token,flag)
push_production(new-production)
search_other_stack_entries(top,in-token,error-count,replace-flag)
process_nonterminal_stack_top(top,in-token,error-count)
process_token(in-token,error-count)
display_error_message(mode,error-no,error-message)
show_errors(mode,error-count,filename)
display_correspondence()
display_symbol_table(symbol-count)
display_parse_tree()
copy_production(token-ptr)
initialize_token_list()
display_token_list(token-list-ptr)
append_token_list(head,tail,token-ptr)
delete_token_list(head,token-ptr)
find_nonterminal_token(head)
find_associated_production(token-ptr)
find_corresponding_entry(str)
replace_corresponding_tokens(token-ptr)
generate_intermediate_code()
store_intermediate_code(filename)
store_symbol_table(symbol-count,filename)
parse_method(argument-count,arguments)
```

REFERENCES

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

[2] Abiteboul, S., and R. Hull, *IFO: A Formal Semantic Database Model*, Technical Report T-84-304, University of Southern California, April 1984.

[3] Agha, G.,*A Message Passing Paradigm for Object Management*, Database Engineering, vol.8, no.4, December 1985, pp. 311-318.

[4] Aho, A.V., J.D. Ullman,**Principles of Compiler Design**,Addison Wesley, 1977.

[5] 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.

[6] 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.

[7] Borning, A.H., *Classes Versus Prototypes in Object-Oriented Languages*,

[8] Buneman, P., and M. Atkinson, *Inheritance and Persistence in Database Programming Languages*, Proceeding of the ACM SIGMOD International Conference on Management of Data, 1986.

[9] Christian, K., **The Unix Operating System**,John Wiley and Sons, !983.

[10] Chou, H.T. and W. Kim,*A unifying Framework for Version Control in a CAD Environment*, Proc. International Conference on Very Large Databases, Kyoto, Japan, 1986.

[11] **Commands Reference Manual**, Sun Microsystems Inc., 1986.

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

[13] Cox, Brad J., **Object-oriented Programming An Evolutionary Approach**, Adddison-Wesley, 1986.

[14] Date, C.J., **An Introduction to Database Systems**,Fourth Edition, vol.1 and vol. 2, Addison Wesley, 1986.

[15] **Debugging Tools for the Sun Workstation**, Sun Microsystems Inc., 1986.

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

[17] **Editing Text Files on the Sun Workstation**, Sun Microsystems Inc., 1986.

[18] Fishman, D.H., et al., *IRIS: An Object-Oriented Database Management System*, ACM Transactions on Office Information Systems, vol.5, no.1, January 1987, pp.48-69.

[19] **Getting Started with Unix: Beginner's Guide**, Sun Microsystems Inc., 1986.

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

[21] Hopcroft, J.E., J.D. Ullman, **Formal Languages and Their Relation to Automata**, Addison Wesley, 1977.

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

[23] Kaehler, T., and D. Patterson, *A Small Taste of Smalltalk*, Byte, August 1986, pp.145-159.

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

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

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

[27] Kesim, N., *An Object Memory for an Object-Oriented Database Management System*, M.S. Thesis, Bilkent University, Ankara, July 1988.

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

[29] Konstantas, D., O.M. Nierstrasz and M. Papathomas,*An Implementation of Hybrid, a Concurrent Object-Oriented Language*,**Active Object Environments**, ed. D. Tsichritzis, Centre Universitaire D'Informatique, Université de Genève, June 1988.

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

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

[32] Lewis, H. R., C.H. Papadimitriou,**Elements of the Theory of Computation**, Prentice Hall, 1981.

[33] Lyngbaeg, P, and V. Vianu, *Mapping a Semantic Database Model to the Relational Model*, ACM SIGMOD International Conference on Management of Data , 1987, pp.132-142.

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

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

[36] 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.

[37] Manola, F., and U. Dayal, *PDM: An Object-Oriented Data Model*, Database Engineering, vol.8, no.4, December 1985.

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

[39] Nierstrasz, O.M.,*Hybrid - A Language for Programming with Active Objects*, **Objects and Things**,ed. D. Tsichritzis,Centre Universitaire D'Informatique, Université de Genève,March 1987,pp.15-42.

[40] Nierstrasz, O.M.,*Triggering Active Objects*,**Objects and Things**, ed. D. Tsichritzis, Centre Universitaire D'Informatique, Université de Genève, March 1987,pp.43-78.

[41] Nierstrasz,O.M.,*A Tour of Hybrid*, Technical Report, Centre Universitaire D'Informatique, Université de Genève.

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

[43] Nierstrasz, O.M., *Active Objects in Hybrid*, OOPSLA '87 proceedings.

[44] Özelçi, S.M., N. Kesim, M. Karaorman, E. Arkun, *An Experimental Object-oriented Database Management System Prototype*, to appear in the Proc. of the Third International Symposium on Computer and Information Sciences, October 1988, Çeşme, Turkey.

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

[46] Penney, D.J., and J. Stein, *Class Modification in the GemStone Object-Oriented DBMS*, Technical Report, Servio Logic Corporation.

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

[48] Smith, J.M., and D.C.P. Smith, *Database Architectures: Aggregation and Generalizations*, ACM Trans. Database Systems, vol.6, no.1, pp.160-173, 1977.

[49] Snodgrass, R., *A Temporal Query Language*,Technical Report TR 85-013, University of North Carolina, Chapel Hill, May 1985.

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

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

[52] Tsichritzis, D., et.al., *KNOs: Knowledge Acquisition, Dissemination and Manipulation Objects*, ACM Transactions on Office Information Systems, vol.5, no.1, January 1987, pp.96-112.

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

[54] Ullman, J.D., *Database Theory: Past and Future*, Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems , 1987, pp.1-10.

[55] **UNIX Interface Reference Manual**, Sun Microsystems Inc., 1986.

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

[57] Zdonik, S.B., *Why Properties are Objects or Some Refinements of 'is-a'*, ACM/IEEE Joint Computer Conference, 1986, pp.41-47.