OBJECT ORIENTED DATABASE MANAGEMENT SYSTEMS

by

ANTONIOS NASSIS

submitted in part fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

in the subject of

INFORMATION SYSTEMS

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: PROFESSOR C H BORNMAN

NOVEMBER 1995

SUMMARY

Modern data intensive applications, such as multimedia systems, require the ability to store and manipulate complex data. The classical Database Management Systems (DBMS), such as relational databases, cannot support these types of applications efficiently.

This dissertation presents the salient features of Object Database Management Systems (ODBMS) and Persistent Programming Languages (PPL), which have been developed to address the data management needs of these difficult applications.

An 'impedance mismatch' problem occurs in the traditional DBMS because the data and computational aspects of the application are implemented using two different systems, that of query and programming language.

PPL's provide facilities to cater for both persistent and transient data within the same language, hence avoiding the impedance mismatch problem.

This dissertation presents a method of implementing a PPL by extending the language C++ with pre-compiled classes. The classes are first developed and then used to implement object persistence in two simple applications.

KEY WORDS

- Database Management Systems
- Database Programming Languages
- Object Oriented Management Systems
- Object Oriented Technology
- Object Persistence
- Persistence
- Persistent Programming Languages
- Pointer Persistence

ABBREVIATIONS

CAD/CAE	Computer Aided Design/Engineering
CAM	Computer Aided Manufacturing
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
OID	Object Identifier
OO	Object Orientation
OOA	Object Oriented Analysis
OOD	Object Oriented Design
ODBMS	Object Oriented Database Management System
ODMG	Object Database Management Group
OMG	Object Management Group
OMT	Object Modeling Technique
OLE	Object Linking and Embedding
PPL	Persistent Programming Languages
SDLC	Software Development Lifecycle

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Database Management Systems (DBMS) are used extensively throughout the computer industry. They are utilised in many organisations that have to store, manage and manipulate large quantities of data. They have grown out of the need to be able to manage efficiently and accurately the vast amounts of data, that typical organisations have to process in their everyday life.

DBMS system developers have, with few exceptions, concentrated in the past on commercial applications. This resulted in the conception and subsequent development of some very powerful models, such as the Hierarchical, Network and Relational Models, which are particularly suitable for such applications.

Recently, however, some applications have been emerging, that require the services of DBMS's with the ability to handle more complex data. Examples of such applications are CAD/CAE, Multimedia, Simulation Systems and Software Engineering Environments.

All these 'modern' applications require the ability to merge complex data. Existing 'classical' DBMS technology cannot support these types of applications efficiently. Object Database Management Systems (ODBMS) have been developed to address the data management needs of these difficult applications.

The main aim of this dissertation is to present an overview of the present status of this ODBMS technology and particularly, the introduction of object persistence to the existing programming language C++.

The dissertation covers both the theoretical and implementation details of persistent objects. The study methodology adopted was to first perform a literature survey on the theoretical aspects of ODBMS's and then to implement some of the techniques of introducing object persistence in C++ using small but non-trivial examples.

Techniques of providing object persistence to the programming language C++ are presented and demonstrated using two examples. One example shows how an 'employee-department' relation is saved to disk, while another shows the application of these techniques to the development of engineering applications, such as that of real-time simulators.

An outline of current DBMS practice is first presented as an introduction in Chapter 2. Chapter 3 then follows with the DBMS requirements of complex modern applications.

Chapter 4 introduces the important developments of Database Programming Languages, with particular reference to Persistent Programming Languages.

Chapter 5 follows with the basic principles of Object Oriented Technology, which forms the foundation of ODBMS's. This leads to Chapter 6, which presents the characteristics and fundamental principles of ODBMS's.

Chapter 7 outlines some of the emerging ODBMS standards.

Chapter 8 is more language dependent, as it presents the implementation details of object persistence in the programming language C++. Two examples are also given which demonstrate the application of these techniques. The appendices includes the complete source code of these examples.

The literature consulted during this study is listed in the reference section.

2. DATABASE MANAGEMENT SYSTEMS

2.1 Introduction

A Database Management System (DBMS) is an important software system used extensively throughout the computer industry.

As for other major software systems, such as compilers and operating systems, fundamental principles have evolved over the years to help engineers and users to understand and use this technology in an efficient manner.

This chapter introduces the fundamental concepts of DBMS's.

A historical perspective is first given, followed by Chapters which deal with the concepts in more depth.

2.2 Historical Perspective

Database Management Technology has evolved through three generations[1] in the last twenty-five years. Successive generations have not replaced their predecessors entirely. Most of the older technologies continued to exist along with the new.

The generations of DBMS's are briefly outlined below and in figure 2.1 [LOOMIS, 90.5] . Further details can be found in other Chapters of this dissertation.

---

[1] The literature is inconsistent with the identification of DBMS generations. In certain cases, the 1st Generation is identified as navigational i.e. Network and Hierarchical, and the 2nd generation as Relational. No definitive 3rd generation database has been identified, but in most cases, it is assumed to be an advanced DBMS that addresses the weaknesses of the previous generations.[PATON, 96]

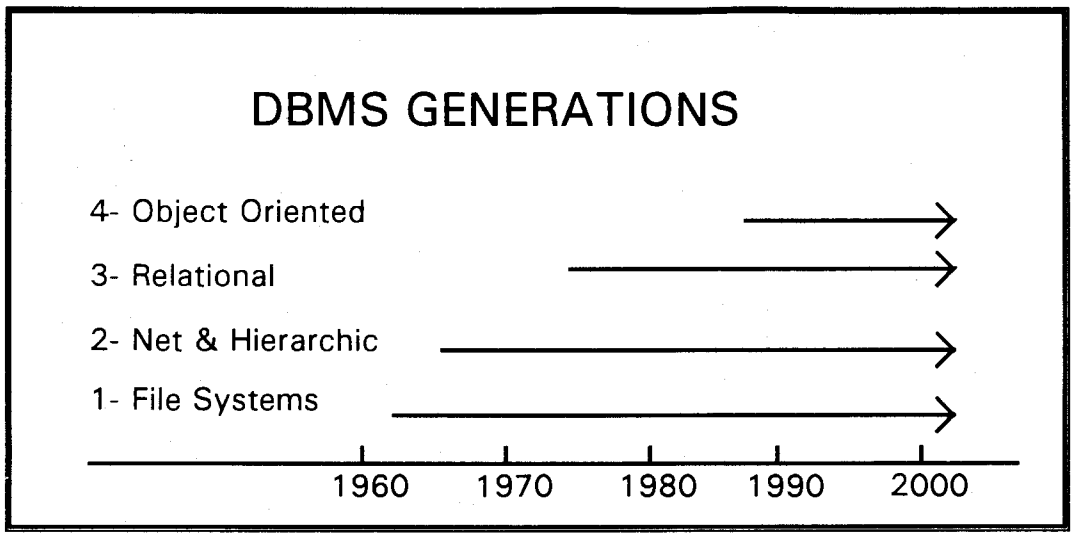


Figure 2.1- DBMS Generations

2.2.1 1st Generation: File Systems

In the absence of a DBMS, an application program uses the language features to store information in file systems provided by the computer's operating environment.

The early file systems provided only sequential data access. i.e. the records could only be read in their physical sequence. This method is sufficient for batch processing of data.

The introduction of more powerful processing hardware, introduced the need for more complex applications. These applications required direct and random access of stored data. Sequential processing was no longer efficient.

The need for random access of stored data was addressed by the introduction of indexing , hashing and other access techniques.

The continual quest for more powerful data processing applications and the use of file systems, precipitated to a plethora of files within organisations. Very often, the same data was held on more than one file, since more than one department had to have access to this data.

Soon, data inconsistency across the file systems of an organisation became a major problem. This problem pre-empted the development of database management systems and the concept of a central database to be shared by the entire organisation.

2.2.2 2nd Generation: Network & Hierarchical DBMS

These systems were developed in the late 1960' and early 1970's to address the problems with file systems mentioned above. [DATE, 86]

The systems offered central data storage management, concurrency control and data recovery. They were mainly developed for the main-frame environment.

2.2.3 3rd Generation: Relational DBMS's

In the 1960's and 1970's, the network and hierarchical databases were dominating the industry. From the late 1970's however, the relational model started to gain some acceptance.

The main disadvantage and criticism of 2nd Generation DBMS's was the complexity involved in using them to create database application. In particular, the models could not accommodate changes easily.

It was for the above reason that alternatives were sought.

The relation model was conceived in the early 1970's. The success and acceptance of the relational model was its simplicity. It views data as records (or tuples) in tables.

Furthermore, the relational model is based on sound formal concepts, that of predicate logic. [DATE, 86] [ELMASRI, 89]

2.2.4 4th Generation: Object Oriented DBMS

This technology has started to appear in commercially available products during the early 1990's.

The ODBMS has its roots in both database technology and object oriented techniques.

It was developed essentially to address the problems of using DBMS technology in applications that handle complex data. Examples of such applications are: CAD, CAE, CAM, Software Engineering Environments, Simulation Tools etc.

2.3 The Characteristics of a DBMS

There are two fundamental characteristics that distinguish database management systems from other software systems [ULLMAN, 88]:

- The ability to manage persistent data, and
- The ability to access large amounts of data efficiently.

The first characteristic above states that there exists a database in which data is stored permanently. The DBMS will access and manipulate this data.

The second characteristic distinguishes a DBMS from a pure file storage system. A pure file system does not, in general, provide facilities to access arbitrary data fast , although it will support permanent storage.

A DBMS is most useful, when large amounts of permanent data must be managed. If the data is not large, a simple linear access technique will suffice.

While the above two characteristics can be regarded as fundamental, there are a number of other facilities which typical commercial DBMS's provide. Some of these facilities are given below:

- Support for at least one **data model** to be used by the implementor and user as an abstraction mechanism.
- Support for a **Data Definition Language** (DDL) whereby the user can define the data structures used.
- Support for **Data Manipulation Language** (DML) or Query Languages. The most common query language for relational DBMS's is SQL.
- **Transaction Management** and **Concurrent Access** to the database by many users.
- **Security** and **Access Control** to inhibit illegal access of data.
- **Data integrity** and **Validity Checks**
- **Resiliency** and the ability to recover from system failures without loss of data.

2.4 Levels of Abstraction in a DBMS

The average user of a DBMS does not get concerned with how the data is physically stored on the permanent media (i.e. hard disk). Hence, levels of abstraction have been introduced to hide storage implementation details from the users.

The Codasyl DBTG[2] report of 1971 recognised the need of a two-level approach. The levels proposed were as follows [DEEN, 85]:

- Schema - which was the system view
- Subschema - which was the user view

The 2-level approach introduced by Codasyl was further expanded to 3-levels by the ANSI/SPARC[3] proposal of February 1975. [ELMASRI, 89] [ULLMAN,88]:

---

[2] Codasyl (Conference on Data Systems Languagess) is an international organisation of computer users, manufacturers, software houses and other interested groups. Its principal objective is the design, development and specification of common user languages. It has produced the Cobol language. Codasyl's involvement in databases is a direct consequence of its interest in the extension of Cobol. [DEEN, 85]

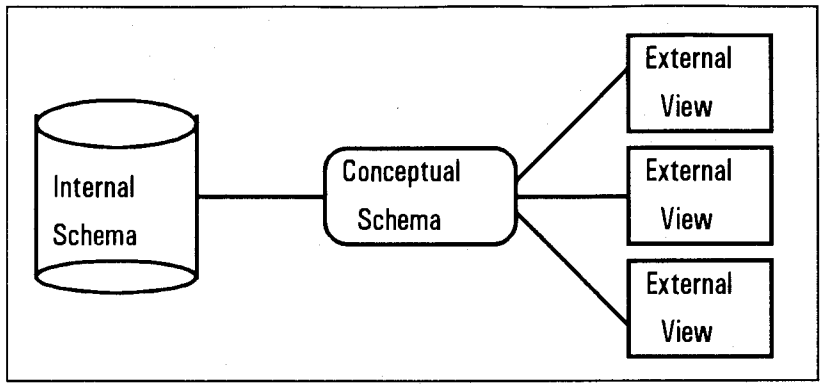


Figure 2.2 - The 3-Schema Architecture of a DBMS

The goal of the three schema architecture, shown in figure 2.2, is to hide the physical implementation details from the user. The three levels are [ELMASRI, 89]:

- **Internal Schema**: which describes the physical storage structure of the database.

- **Conceptual Schema**: which describes the structure of the whole database. The description hides the details of physical storage structures and presents the data in the format of the high-level data model used by the DBMS.

- **External View Schema**: which presents the data to a particular group of users in a way that is most useful to them. External views can be created per group of users with particular needs.

2.5 Data Independence

The DBMS architecture given above provides, what is known as, data independence. Two types of data independence can be defined [ELMASRI, 89]:

---

[3] ANSI (American National Standards Institute) is responsible for the standardisation of products in the USA. Its subcomitte X3 deals with data processing systems and has a committee called SPARC (Standards Planning And Requirements Committee) which conceived the ANSI/SPARC 3-level architecture. [DEEN, 85]

- **Logical Data Independence**: which is the capacity to change the conceptual schema without having to change the external view or application programs.

- **Physical Data Independence**: which is the capacity to change the internal or physical schema without having to change the conceptual schema. This implies that optimisation can be carried out at the physical level without the need of modifying any of the above levels.

Data independence provides an isolation between the database system implementors and the programmers that are coding the DBMS itself.

2.6 DBMS Languages

In ordinary programming languages, the declaration and executable statements are all part of the same language. In DBMS's, it is common to separate the declarative part into a different language. [ULLMAN, 88]

Hence a DBMS usually has provision for the following languages:

- Data Definition Language (DDL)
- Data Manipulation Language (DML) (e.g. SQL)
- Host language (i.e. Cobol, C, Pascal)

The concept behind this separation is that in a database, the data persists and hence should be declared only once, while in an ordinary program, the data exist only while the program is running[4].

DBMS language concepts are further expanded below:

---

[4] This separation, however, leads to the 'impedance mismatch' problem which is discussed in section 2.8.7.

2.6.1 Data Definition Languages

DDL is used to define the conceptual schema. This is a notational language which is used when the database is designed or when it is modified.

In relational databases, a subset of SQL is used to implement the DML.

2.6.2 Data Manipulation Languages

DML is used to perform operations on the data of the database. SQL is a DML language used most often on relational databases.

SQL is a declarative query language. This means that the user need only specify what the data query is and not how to obtain it.

As an example, consider a database containing a list of employee names and their manager:

EMPLOYEES[Name, Manager]

This is represented as a table of Employee names with the name of their manager.

An SQL query to extract the manager of employee 'Kent' is:

```
SELECT   MANAGER
FROM     EMPLOYEES,
WHERE    EMPLOYEES.NAME = 'Kent'
```

The types of operations that can be performed include:

- Update
- Retrieval
- Query

2.6.3 Host Languages

DML is limited to the few fundamental operations given above. It is usually necessary, however, to perform more complex operations that simply manipulate the data in the database.

For this reason, DBMS's provide links to procedural host languages.

The host language is used to implement the required functionality of the system, except the actual querying and modification of the database.

2.7 Classical DBMS Data Models

Four generations of DBMS's have been given in the section above dealing with Historical Perspectives. Three of the generations are referred to as 'Classical' in the current literature. The Data Models used in the classical DBMS's are outlined below:

2.7.1 Hierarchical

A hierarchy (or tree) is a network in which nodes are connected by links such that all links point in the direction from child to parent. Hence every node has a parent node except for the single root node of the hierarchy.

In a hierarchical database, the nodes of the hierarchy consist of records which are connected to each other via links.

Hierarchical databases often exhibit poor flexibility, but because of the 'hard-wired' access paths, they often provide very good performance.[DATE, 86]

2.7.2 Network

The network model uses additional pointers to add flexibility to the hierarchical model.

In its most general form, a network is a collection of nodes with links possible between any of the nodes.

Network databases have the same performance advantage of hierarchical databases. [DATE, 86]

2.7.3 Relational

A relational database consists of a set of tables. Each row in a relation represents a relationship between a set of values.

One reason for the widespread acceptance of relational databases is that they are based on an easily understood model. This model is given below. [ELMASRI, 89]

The relational model represents the data in a database as a collection of relations. Each relation resembles a table or, to some extend a file.

In relational database terminology, a row is called a tuple, a column name is called an attribute and the table is called a relation. A domain is a set of atomic values and is usually allocated a name such that it can be referenced. An atomic value is a value which is indivisible. A data type and/or format is also specified per domain. Examples of acceptable domains and their data types are given below:

Domain Name	**Data Type & Format**
SA_Phone_no	(ddd)ddd-dddd, d is a decimal digit
Salary	float
Name	string[20]

The following rules are strictly enforced by the Relational Model:

- The domain of the column (or attribute) values is a set of Atomic values. This means that each attribute value is indivisible. i.e. composite and multivalued attributes are not allowed. This has the disadvantage that structures cannot be stored directly, they have to be decomposed: This decomposition leads to a loss of semantic information.
- A relation is defined as a set of tuples. Mathematically, elements of a set have no order among them. Hence tuples do not form an ordered set.
- A tuple however, is defined as an ordered list of 'n' values where 'n' is the degree of a relation.
- A relation is defined as a set of tuples. By definition, all elements of a set are distinct. Hence, all tuples in a relation must also be distinct. This means that no two tuples in a relation can have the same values for all their attributes.

The relational model is based on mathematically precise concepts. This means that relational algebra can be used to define operations on the data in a declarative way.

The ability to use mathematics on this model enables the implementation of declarative query languages and the automatic optimisation of such queries. This ability, together with the models inherent simplicity, has been one of the main reasons for its wide support and acceptance across the industry.

2.8 Weaknesses of the Relational Model

The great strength of the relational model is its basis in formal theory, that of predicate logic. This is what makes it possible to have a relationally complete[5], declarative query language.

This formal basis of the relational model also allows automatic optimisation of queries and rigorous normalisation theory to be applied.

Although currently the relational data model predominates this market, there are instances were this model cannot be applied. Some of its shortcomings are given below. [GRAHAM, 91]

2.8.1 Normalisation

Normalisation is a process during which relation schemas are decomposed by breaking up their attributes into smaller relations schemas which possess desirable properties.

This decomposition ensures that certain update anomalies and data redundancy is avoided.

---

[5] A language that can express all of the safe tuple calculus or equivalently, relational algebra operations, is said to be relationally complete. [ULLMAN, 88]

The problem, however, is that normalisation is driven by computational or logic considerations rather than the structure of the application, The end-result often does not resemble the real world and the semantics and structure of the data is lost.

2.8.2 Integrity and Business Rules

The relational model usually provides two integrity rules:

- **Entity Integrity:** No primary key may include an attribute which may take null as a value.
- **Referential Integrity:** A tuple in one relation that refers to another relation must refer to an existing tuple in the relation.

Business rules are statements such as

> **If** (employee has over five years service) **then**
> (award extra day leave)

Most relational DBMS's today have provision for stating entity and referential integrity rules.

Business rules, however, have to be coded using SQL or the host language. This makes understanding and changing of these rules difficult.

2.8.3 Null Values

Null values in tuples present difficulty in interpretation. A null can be taken to mean either,

- the value is not applicable to a particular tuple, or
- the value has not been supplied.

The above is as a result of the relational model formal basis. Formal relational theory cannot deal easily with null values if this value form part of the Primary Key.

2.8.4 Complex Objects

The first normal form forbids the storage of complex objects. Hence abstract data types (i.e. lists, trees etc.) cannot be represented directly by the relational model and have to be decomposed into individual atomic values. Semantic information, such as inheritance hierarchy, is therefore lost

An example of this decomposition is given in figure 2.3

The figure represents an aggregation (which is a 'part-of' association) of Motor Vehicle parts.

The usage of 'tables' to represent the data, results in a loss of semantic information. i.e. there is no way that the aggregation can be derived directly from the tables. Hence, applications usually depend on the use of a richer model, such as the Entity-Relationship conceptual model, to capture the original semantic information. [ELMASRI, 89]

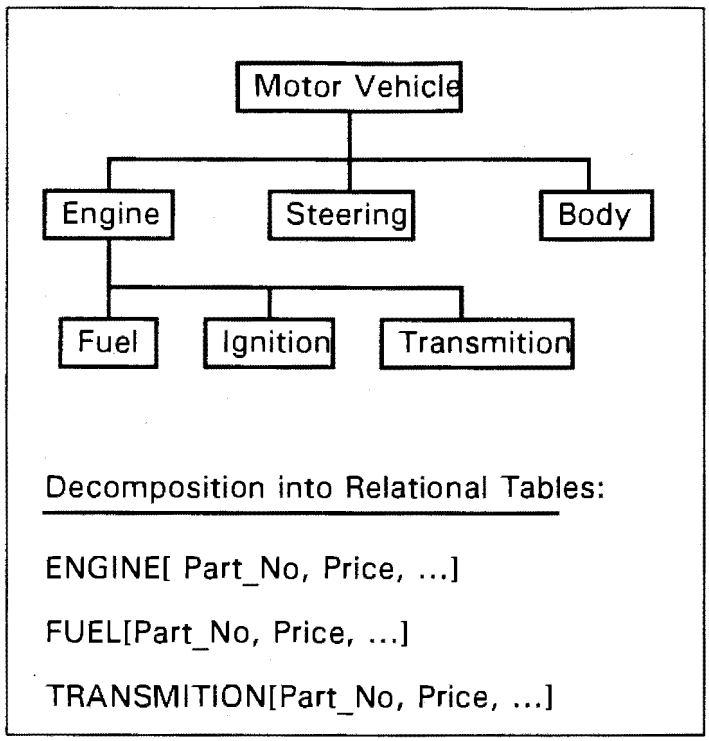


Figure 2.3 - Decomposition of Aggregate Association into Relational Tables

2.8.5 Recursive Queries

Another limitation of the relational model, is that recursive queries cannot be handled.

A recursive query is a query about relationships that an entity has with itself. For example in the Employee/Manager relationship shown below:

EMPLOYEES[Name, Manager]

it is impossible to ask, in one query, for all employees who are managers of a particular employee, at all levels in the hierarchy.

2.8.6 Identity

In a relational table, all tuples must be distinct. In cases where the possibility exists for duplicate tuples, another unique attribute must be defined to provide this tuple uniqueness.

For example, in the EMPLOYEE(Name, Manager) relation, if two employees have the same name, then employee Identification Numbers must be given in order to satisfy the relational model's needs. (even if such numbers are not required by the organisation)

Systems based on the relational model are called 'Value Oriented' since identification is based purely on the value of the attributes. [ULLMAN, 88]

In contrast, Object Oriented Systems are not 'Value-Oriented', since object identity is inherent to the object and is not based on the 'state' of the object i.e. not based on the value of its attributes.

Hence, systems based on the relational model might appear less natural, since artificial attributes have to be attached to their tables to make them conform to the relational model's requirements.

2.8.7 Impedance Mismatch

As mentioned in the sections above, a host language is often used together with the DBMS in order to provide more computational power to the application than is normally available by the declarative query language.

This secondary language leads to a few problems collectively referred to as the 'impedance mismatch[6]'.

---

[6] The term 'impedance mismatch' is borrowed from electrical engineering and refers to the problem of terminating transmission lines with the same impedance of the line.

There are two type of impedance mismatch as shown below [PATON, 96]:

- **Data Type Mismatch**

  The data types stored in the database are not directly supported by the programming language. Hence transformations are required by all data that has to be both stored by the database and manipulated by the host language.

  The host language cannot support strong type checking, since it does not support the data type of the DBMS.

- **Evaluation Strategy Mismatch**

  Database query languages act on sets of tuples, whereas host languages act on one record at a time.

  Hence, when a retrieval is requested by the host language, a set of tuples will be returned. This set is then stored by the system such that the host language can process it one record at a time. This process leads to a degradation of efficiency.

2.9 Conclusions

Most data intensive applications are developed using a DBMS together with a host language.

The relational DBMS continues to predominate this market since its inception during the late 1970's. The main strengths of this data model is its simplicity and its mathematical basis, which enables the implementation and use of declarative query languages.

The relational model, however, has two major weakness in that it cannot represent complex objects directly and it introduces an impedance mismatch between the database and the host language.

The chapters that follow expand on the above issues and present alternative DBMS's that have been developed to eliminate the weaknesses of the classical data models.

3. MODERN DATABASE APPLICATIONS

3.1 Introduction

The classical DBMS's described in chapter 2 were designed to fulfil the need of the business community with important but limited capability applications. Applications such as pay-roll systems, employee databases and store systems have been successfully developed using commercially available relational databases.

The common characteristics of these commercial applications are that they have large amounts of data to be stored, but the required operations to be performed are relatively simple. In such database systems, insertion, deletion and retrieval, together with the simple data queries provided by a declarative DML, is all that is required.

There are however applications that are distinct from the above mentioned ones in the following ways:

- Applications that require large amounts of complex (but structured) data.
- Applications that require complex database queries and manipulations.

Examples of the above are:

- Computer Aided Design, Engineering and Manufacture (CAD, CAE & CAM)
- Very Large Scale Integration (VLSI) Circuits Design e.g. design of microprocessors.
- Multi-Media Applications
- Central Repository for Software Engineering Environments.

In the systems mentioned above, current classical database technology has been found to be limited and in most cases, inappropriate to use.

This chapter presents two examples of complex data intensive applications, that of multimedia and CAD/CAE. Alternative data models developed to address such applications are described in the chapters that follow.

3.2 Compound 'Multimedia' Document Storage

Figure 3.1 is an example of a compound document which, in this case, is a memorandum sent interactively to someone using multimedia technology. [KIM, 93]

The document consists of many objects including sound-clips, visual images and text.

Figure 3.2 is an aggregation hierarchy that models the document.

MEMORANDUM

TO : John Smith

FROM : Errol White

SUBJECT : Meeting on Database Strategy

Please note that there will be a meeting on 14/8/94 at 14h00 to discuss the 'future database strategy' presentation to management.

Click here to view the **video** clip that I have made : O

Click her to view the **graphic** slides made by Jane: O

Regards,

Errol.

Figure 3.1 - A Compound Document

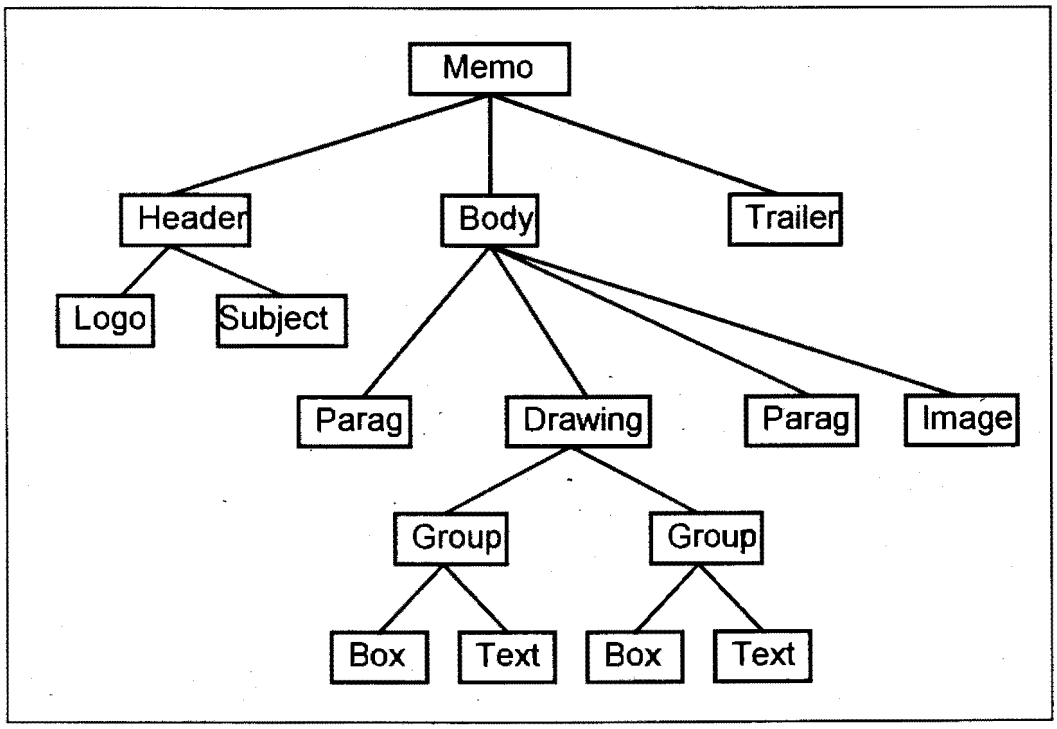


Figure 3.2 - Class Hierarchy of a Compound Document

Storing this data in a relational database, will necessitate the decomposition and hence the loss of semantic information. i.e. The semantic meaning of the hierarchical aggregation will be lost.

Although the images and sound clips can be stored in a relational database as 'Blob' (Binary Large Objects) fields, it will be impossible for someone to edit and modify these images. This is because a 'Blob' field can only store an image in binary 'pixel' format. The image can never be decomposed into lines segment , as it was originally, and hence cannot be edited.

3.3 DBMS Requirements for Engineering Applications

A large multi-disciplinary engineering project will involve a group of engineers working co-operatively on complex designs. [AHMED, 92]

Sophisticated CAD/CAE tools, running on distributed workstations, are typically used in such projects, in order to handle the complexity and to improve the quality and productivity of the team.

The designers and engineers in such a project will require to closely interact amongst themselves and dynamically share design data amongst themselves as well as with outside contractors.

Such design tools necessitate complex data modelling and handling capabilities which cannot be found in the 'classical' DBMS.

The DBMS features required for such tools or environments are listed below:

- **Complex Information Modelling Capabilities**

  Engineering data representation is complex because of the complexity of the physical systems that have to be modelled and designed.

- **Semantic Schema Design**

  Large database schemas must reflect and preserve the semantics and hierarchy of the original design data.

- **Dynamic Schema Evolution**

  Engineering design applications require constant and frequent modifications to the structure of the data. This necessitates the on-line schema modification capability. [AHMED, 92]

- **Rigorous Constraint Management**

  Due to the size and complexity of engineering databases, consistency of data state must be maintained by enforcing rigorous design constraints as the data evolves.

  For example, in positioning of equipment in a 3-dimensional space, the system must check if clashes with other equipment exist.

- **Efficient Management of Large Volumes of Data**

  Engineering design applications are highly data intensive and hence require efficient management of data.

  In addition, management efficiency is very critical in interactive, high-resolution graphic based CAD transactions.

  Relational databases have proven to be very slow for such applications.

- **Data Versioning**

  Data must also be versionable so that different versions coexist in the database and data updates do not result in the overwriting of old data.

- **Interclient Communications**

  Designers need to be aware of each others design state. This prevents repetition and inconsistent designs.

  Facilities and communication protocols must hence be provided and supported by the system.

  For example, someone that is designing the civil work for the plant's building needs to know whether the mechanical engineers have finalised their selection and placing of equipment.

- **Computationally Complete Database Programming Language**

  Engineering applications require complex mathematical transformations. These cannot be done using SQL.

  Ideally, the database language must be computationally complete in order to avoid the 'impedance mismatch'. i.e. combination of two languages, host and database.

- **Compatibility, Extensibility and Integration**

  Such an engineering environment will necessitate the integration of many tools and facilities. The DBMS will hence have to be flexible, extensible and have the capability of integrating many foreign systems.

  This could be accomplished by having a standard public interface to its data and facilities.

From the above, it can be concluded that the 'classical' DBMS systems cannot meet the requirements of engineering applications.

Recently, 4th Generation DBMS's have evolved to address such needs. These systems are described in the sections that follow.

3.4 Conclusions

The classical relational DBMS cannot meet the demands of complex data intensive applications, such as multimedia and engineering applications presented in this chapter.

These applications require the direct representation of complex data whereas the relational model will normalise the data into flat tables. The structure of this data will not, therefore, be directly represented.

Fourth generation DBMS's have evolved to address such needs.

4. DATABASE PROGRAMMING LANGUAGES

4.1 Introduction

Traditionally, applications that required the handling and storing of large amounts of data tended to be developed using a database system, such as a relational database management system (DBMS) and a programming language.

The use of a programming language was necessary because the computational power available in the traditional DBMS's was not sufficient to handle complex data computation and processing.

As mentioned in chapter 2, this has led to what is known as the 'impedance mismatch[7]' between programming language and database management systems.

Attempts to alleviate the above mismatch problem has led to the following approaches [PATON, 96]:

- Extend the relational model such that more complex data and computation can be handled without the need of an external programming language. Examples of this approach is the proposed extended SQL model and query language.
- Develop a programming language that can handle both the database needs of the application as well as the computational needs. These languages aim to fully and transparently integrate the computational power of a programming language with the database requirements of a DBMS. These languages are called 'Database Programming Languages'.

There is no single universally accepted approach to the implementation of such languages, but the sections that follow expand on some of the issues of the implementation and usage of such languages.

---

[7] See section 2.8.7 for the definition of this term.

4.2 Database Programming Languages

The requirements of a database language are as follows [PATON, 96]:

- It must increase the computational power of the typical database manipulation language, such as SQL. This language should avoid the need of using separate programming environments for the database and computational needs of the application, thus avoiding the impedance mismatch problem.
- It must provide a richer data model than the relational model such that complex data structures can be represented without the need of disassembling them into atomic entities.

Current research in this direction has led to the following database language developments:

- **Deductive Database Systems**

  In these systems, the logic programming paradigm is combined with the relational data model.

  The database can store facts and rules using 1st order logic, which can then be used to derive new information.

  These systems, which were developed mainly from the programming language Prolog, have the following advantages [PATON, 96]:

  - They are a natural extension to the relational model.
  - They are founded on a sound theoretical model which is an extension of the relational data model.
  - They can represent and manipulate knowledge and data in a natural way.

- **Functional Database Systems**

  In these systems, the functional programming paradigm is used together with either the relational or the object oriented data models.

These systems have the following benefits as a result of their functional programming paradigm:

- Ease of reasoning.
- Freedom from a detailed execution order.
- Freedom from side-effects.

- **Persistent Programming Language**

  These are programming languages that support mechanisms that allow all data types defined by the language to be stored directly on disk.

  Hence, programs do not have to use an external database system to access data i.e. there is no impedance mismatch.

  The data structures available are more complex than the ones provided by the relational model.

  The imperative programming paradigm is usually implemented.

- **Object Oriented Database Systems**

  These are usually persistent programming languages that support the object oriented programming paradigm.

  Richer data structures and hierarchies can be directly implemented and stored on disk with no need for data model translation.

The sections that follow expand on the principles of both persistent programming languages and object oriented databases.

4.3 Persistent Programming Languages

Traditional programming languages, such as C, C++ and Pascal , have been developed with an aim to provide elegant structures and computational features. Programming paradigms such as Structured Programming and Object Orientation have been developed specifically to enable efficient representation of the problem domains to be solved.

The elegance of the above mentioned paradigms diminishes when the problem domain requires the long-term storage of data to a device such as a hard-disk. Such persistent data requirement must be implemented either by using a file system or an external DBMS.

A language that supports both DBMS features and computational features of an accepted programming paradigm seamlessly is called a Persistent Programming Language. [PATON,96]

The desirable features of such languages are presented below. [PATON,96]

4.3.1 General Principles

The following are some general guidelines used in the design of programming languages[PATON, 96] [SEBESTA,93]:

- **Principle of Correspondence**
  This states that the ways in which named objects are introduced should be the same everywhere.

- **Principle of Abstraction**
  Abstraction means the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored.

- **Principle of Data-Type Completeness**

This states that every data type should have the same rules for manipulation, with no exceptions.

4.3.2 Representing the Persistent Data

In a data intensive application, the need often arises to treat two types of data as follows:

- **transient data**

  This data ceases to exist once the program is out of execution scope.

- **persistent data**

  This data will have to be kept on a non-volatile media, such as a disk, for further use beyond the current execution.

Traditionally the representation and implementation of persistent data was achieved by using data models and DBMS's whereas type systems and host programming languages were used to handle the temporary data. This implied that the programmer had to continually convert between the two data representation systems.

Database Programming Languages aim to avoid this difference by having only one representation within the language. Hence the real-world data is represented using the same data model whether it is persistent or transient.

Some of the desirable data representation features of such languages are as follows:

- Rich support for complex data structures-
- Strong Typing
- The possibility of user-requested run-time type checking i.e. support for variant data types.

4.3.3 Computational Features

Some of the desirable computational features that a persistent programming language should have are:

- The language should be computational complete. i.e. be able to compute expressions of any complexity.
- Support for functions and/or subprograms.

4.3.4 Data Storage Features

The following are the preferred features of data persistence:

- The same data structure should be used for both the transient and persistent data.
- The type constraints of a value should be retained and not violated whether the data is persistent or transient.
- The same identifier (or name) should be retained throughout the lifecycle of a persistent or transient data element.
- It should be possible to make any data type persistent. This is the principle of orthogonal persistence.

4.3.5 Methods of Providing Persistence

The following methods of providing data persistence in a language can be used:

- **File Persistence**

  Persistent values are simply written in a disk file.

- **Session Persistence**

  All the values in a program are saved to disk at the end of the program execution and read back at the beginning of execution.

  This presents the following disadvantages:

- Usually it is not required for all data values to be persistent. Hence the technique is not space efficient.
- It is also impossible to save or load values to and from the disk incrementally. Again this results in inefficient use of both memory and persistent store.
- Concurrent access to data is also not possible.

- **Orthogonal Persistence**

  This allows the incremental saving and loading of data elements to the persistence store, thus greatly enhancing the space efficiency of both the persistent store and the system memory.

  This feature can be provided in a language in the following ways:

  - By variable Declaration e.g.:

    persist int x;
  - By an explicit language operator e.g.:

    save x;
  - By identifying a structure that can hold many elements as persistent. Data values entered in this structure become persistent. The structure will be traversed by the system in order to extract and save all elements. This is called persistence by reachability.

4.4 Object Oriented Databases

Object Oriented Databases are usually persistent programming languages that support the object oriented paradigm.

They can be implemented by using the following techniques:

- **Extend an existing Programming Language**
  Both C++ and Smalltalk have been used to implement persistent objects. Usually classes or frameworks are created which are then inherited by the objects that need to be persistent.

- **Create a new language**
  A completely new language can be created to provide both object oriented programming and persistent storage.

Chapter 6 describes the ODBMS in more detail and chapter 7 presents techniques for implementing persistent objects in the existing programming language C++.

4.5 Conclusions

Database Programming Languages have been developed as an attempt to eliminate the impedance mismatch and the complex data representation problems experienced by classical relational DBMSs.

These languages integrate fully the computational power of a programming language with the data processing features of a DBMS.

Although there is no single universally accepted database programming language, the following systems have been developed:

- Deductive Database Systems
- Functional Database Systems
- Persistent Programming Languages
- Object Oriented Database Systems

5. THE OBJECT ORIENTED TECHNOLOGY

5.1 Introduction

The Object Oriented Technology has steadily been gaining acceptance in the commercial and industrial software industries. Some of the object oriented technologies that are emerging are as follows:

- Object Oriented Programming Languages
- Compound Documents
- Component Software
- Object Oriented Database Management Systems (ODBMS)

The methodology used for the analysis and design of object oriented systems is applicable to both the design of applications that are data intensive (using ODBMS and object persistence) and those that are not.

The most common programming languages are C++ and Smalltalk.

Compound Documents have been introduced recently with the aim of integrating different types of documents into a single document. For example, a report file could contain a number of different types of documents, such as graphs, images and other multi-media data. Prior to this technology, a number of different files had to be kept, one for each type of document.

Component Software has the main aim of code re-use. For example, a number of applications, such as spread-sheet, word-processor etc., could share one spelling correction application. The alternative would be for each of the applications to embed their own spelling checker.

Both compound documents and component software have to use persistent object storage, which is a central part of an ODBMS.

The above technologies are presented in more detail in the sections that follow.

5.2 Historical Perspective

5.2.1 The Generation of Programming Languages

As we look back into the history of software engineering, two significant trends emerge:

- The shift in focus from small program development to large and complex program requirements.
- The evolution of high-level Programming Languages.

The evolution of languages and their abstraction mechanisms can be categorised as follows [BOOCH, 94]:

1st Generation Languages (1954-58)

FORTRAN I	math expression
ALGOL 58	math expression

2nd Generation Languages (1959-61)

FORTRAN II	subroutines
ALGOL 60	data types
COBOL	data description, file handling
LISP	list processing, pointers

3rd Generation Languages (1962-70)

PL/1	FORTRAN + ALGOL + COBOL
PASCAL	simple successor to ALGOL
SIMULA	classes, data abstraction

The level and kind of abstraction was changed in each generation.

First Generation languages were designed to solve mathematical problems. Hence the kind of abstraction was mathematical. The languages therefore represented a step closer to the problem space and a step further away from the computing machine.

In the second generation languages, algorithmic abstraction was introduced. This presented a step closer to the problem domain, as it allowed programmers to 'tell' the machine what to do. (e.g. read records first, sort them and print them)

As larger and more complex programs begun to emerge, data became more complex to manipulate. This prompted the conception of data abstraction languages, such as PASCAL. Hence the third generation of languages again moved the software a step closer to the problem domain.

The drive for new generations of languages has therefore been mainly as an aid to the programmer for managing complexity.

5.2.2 Emergence of Object Oriented Languages

The term 'object' emerged almost independently and simultaneously in the 1970s, to refer to notions that were different in their appearance but mutually related. [BOOCH, 94]

The language SIMULA 67 was the first language to be based on the fundamental ideas of classes and objects.

SMALLTALK-80, took Simula's object-oriented paradigm to its natural conclusion by making everything in the language an instance of a class.

What followed then was an emergence of OO Languages e.g. C++ (derived from C) and ADA.

5.3 The Object Model

Object Oriented Techniques are built upon a sound foundation whose elements are called the Object Model. The elements of this model are presented in this section. [BOOCH, 94]

5.3.1 Elements of the Object Model

This Object Oriented Conceptual Model has the following major elements[8] [BOOCH,94]:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy & Inheritance
- Classification

The above major elements must be present in the model for it to be called 'object oriented'.

The following minor elements are useful but not essential to the model. This means that it is not necessary for a model to support the minor elements for it to be called 'Object Oriented'.

- Typing
- Polymorphism
- Concurrency
- Persistence

Each of these elements are now described in detail:

---

[8] There is some dispute in the literature about exactly what the minimum characteristics are of an Object Model. For example, some proponents (eg J.Rumbaugh) argue that 'polymorphism' is an essential property. [RUMBAUGH, 91] [SOMMERVILLE, 89]

- **Abstraction**

  Abstraction is a means that humans use to cope with complexity. For example, to teach a person how to drive a car, it is not necessary to refer to the car as a collection of machinery (e.g. carburettor, cylinder etc.). Rather the car is introduced as containing the essential objects for driving i.e. brake pedal, gas pedal, gears etc.

  Hence a simplified representation is abstracted from a complex object.

  An abstraction focuses on the relevant view of an object and so serves to separate an object's essential behaviour from its implementation.

- **Encapsulation**

  Encapsulation is used to hide the abstraction's implementation from its users.

  The benefits of encapsulation is that users of the abstraction do not depend on its internal details (i.e. abstraction implementation details are hidden). Changes to these details will then not affect other users. This also prevents a program from becoming so interdependent such that a small change has massive ripple effects.

- **Modularity**

  This is a process of dividing the program into separate compilable sections. These sections (or modules) should be cohesive and loosely coupled.

- **Hierarchy and Inheritance**

  Hierarchy is used to manage large systems containing a large amount of abstractions. A set of abstraction often form a hierarchy and by a process of classification, the abstractions can be grouped in hierarchical way.

  Hence hierarchy is a ranking or ordering of abstractions.

The property of Inheritance enables subclasses to inherit behaviour and attributes from their parent classes.

- **Classification**

  A Class is a collection of objects which share common behaviour and attributes.

  Classification refers to the ability of first declaring object classes and then instantiating these classes into objects.

- **Typing**

  This is the enforcement of the class of an object, such that objects of different types may not be interchanged.

- **Polymorphism**

  This refers to the ability to use the same message for similar operations. For example, the same message, 'draw', can be sent to a 'square' and to a 'circle' object. Each object will know how to draw itself.

- **Concurrency**

  Concurrency enables the concurrent execution of different objects.

- **Persistence**

  An object with this property continues to exist after its creator ceases to exist (e.g. exists after the program is terminated)

  Persistent objects are used in the implementation of object oriented database systems, and will be discussed in later sections of this dissertation.

5.3.2 Objects

An object can be any of the following:

- A tangible and/or visible thing

- Something that may be apprehended intellectually
- Something toward which a thought or actions can be directed.

Hence an object is something that has crisply defined boundaries.

A more formal definition of an object is [BOOCH 90]:

- An object has **State, Behaviour and Identity**
- The structure and behaviour of similar objects are defined in their **common class**
- The terms **instance** and **object** are interchangeable.

The terms used in the above definition are now described in more detail:

- **State**

  The state of the object describes the object properties and their current value. For example, a class 'person' could be defined to have the following properties:

  Name

  Address

  Age

  An object of the above class, if it exists at some time 't', must have values associated with the above properties. These properties and values together constitute the State of the Object.

- **Behaviour**

  The behaviour of an object is described by how an object acts and reacts, in terms of its state changes and message passing.

  The methods of an object are the operations that clients may perform on that object.

- **Identity**

  Identity is that property of an object that distinguishes it from other objects.

5.3.3 Classes

Objects get created as belonging to a particular class. By means of this mechanism, the methods of similar objects do not have to be defined every time an object gets created.

For example, in a personnel database system a 'PersonRecord' class can be defined to hold all the properties and methods required to store and process personnel data. An object of class 'PersonRecord' then gets created for every employee.

Hence, a Class is a set of objects that share a common structure and a common behaviour.

A Class has two portions as follows:

- The Interface Portion which provides the outside view of the class. This portion consists of the method declarations and interface properties applicable to instances of this class.

- The Implementation Portion which provides the inside view of the class. This portion contains the hidden implementation of its behaviour.

5.3.4 Class Relationships

Through a process of classification, different classes can be related to each other.

There are two types of relationships as follows:

- Inheritance Relationship
- Using Relationship

'Inheritance relationships' can be used to express generalisations and associations. 'Using Relationships' can be used to express aggregation. These concepts are described below [ELMASRI, 89]:

- **Generalisation**

Generalisation is the process of generalising several classes into a higher level abstract class.

For example, the classes 'CAR' and 'TRACK' are generalisations of the class 'VEHICLE'.

- **Association**

  This is used to associate a class with several independent classes.

- **Aggregation**

  Used for building composite objects from their component objects. e.g. the classes 'CARBURETTOR', 'CYLINDER', 'RADIATOR' can be aggregated to the class 'CAR_ENGINE'.

Through the above processes, classes can inherit structure and methods from higher order classes. This means that code does not have to be re-generated for every new class required.

Hence code re-usability and extensibility is possible in OO Programs through the use of inheritance.

5.4 Compound Documents

Compound Document technology transforms the way software applications interact and appear on the screen. Previously, if users needed to create documents which contained heterogeneous data such as text, graphic pictures, graphs etc., they first had to use different applications to generate these different sets of data and then attempt to integrate them into a single document. [ADLER, 95]

For example, it was common to first use a graphics package to generate the different drawings required in the document and then, using an appropriate word-processor, to integrate these drawings into one document.

Compound Document Technology alleviates these laborious steps by focusing on the data to be created, rather than the applications required to generate this data. In essence, a compound document is a container for sharing heterogeneous data. The mechanisms that manage this integration of data, maintain associations between the data and the application that created it. This specific application is executed transparently when the user needs to update the specific data.

The net result, is that the primary document application appears functionally and seamlessly integrated with all other relevant applications. e.g. if the main application is a Word-processor, a user can draw and edit figures and charts using transparently a drawing application.

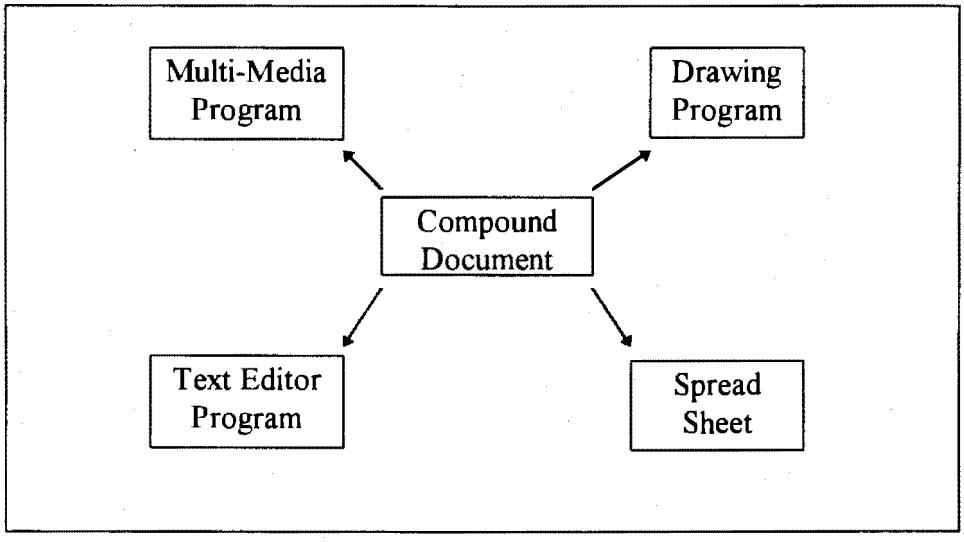


Figure 5.1 - Compound Document

5.5 Component Software

Component Software technology addresses the general problem of developing systems from application elements (components) that were constructed independently by different developers using different languages, tools and development platforms. [ADLER, 95] [BETZ, 94]

The aim of this technology is to increase the re-usability of applications. This will result in the reduction of development time and resources.

As an example of the potential of this technology, a Word-processor development company might decide to use a Spelling Checker which exists, instead of developing their own. Component technology will enable this seamless integration between the different products.

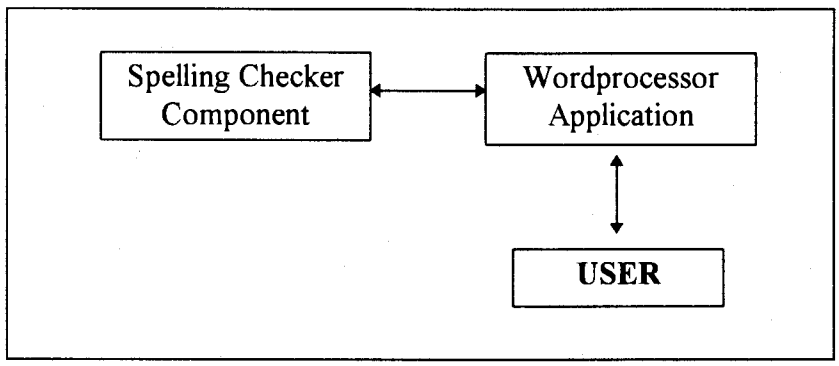


Figure 5.2 - The use of Component Software

5.6 Distributed Object Computing

Component Software Technology will enable the seamless use and integration of objects, thereby drastically reducing development costs.

Distributed Object Technology enables the use and integration of component objects across a network of different application environments and operating systems. This will enable the sharing of applications across different architectures, operating systems and geographically remote platforms.

Using the above example of the Word-processor application, the Speller application can reside anywhere on the network, as shown below:

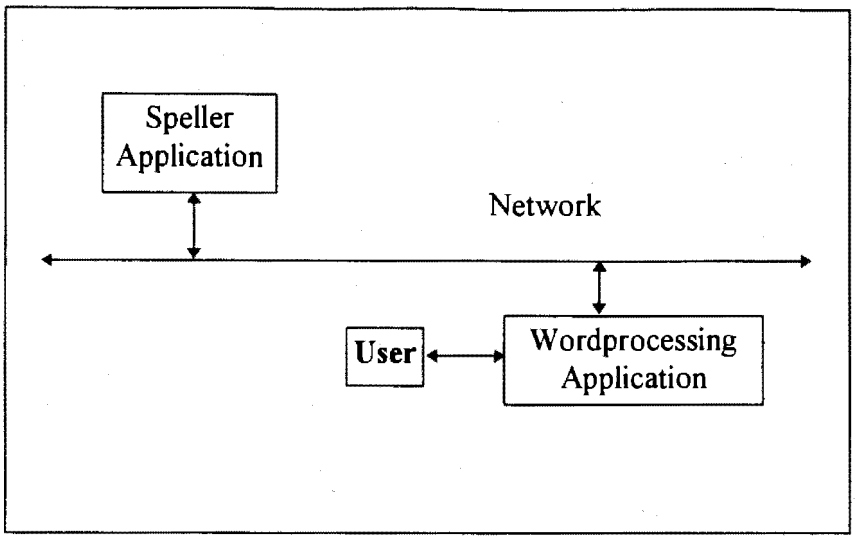


Figure 5.3 - Distributed Objects

5.7 Object Oriented Databases

This technology closely resembles the Distributed Component Software technology. The main difference is that the focus is on the persistent storage and subsequent retrieval of objects, rather than the sharing of application objects.

Object Oriented Databases aim to solve the problems experienced by Relational databases in the representation and storage of complex data, such as that found in engineering applications (CAD, CAE etc.)

5.8 Conclusion

Object Oriented Technology can provide the means to handle large, complex software projects, in a more efficient and productive way.

This technology, although very young and relatively immature, is continuing to gain wide acceptance in the software industry.

Object Oriented Programming Languages (Smalltalk and especially C++) are readily available on many hardware platforms and international standards for these are in the process of being introduced.

The chapters that follow provide a more detailed discussion of ODBMS's and the implementation of object persistence within the C++ programming language.

6. OBJECT ORIENTED DATABASE MANAGEMENT SYSTEMS

6.1 Introduction

As mentioned in chapter 2, during the past three decades the database technology has evolved from being pure file oriented to relational.

The relational model currently predominates, but there are certain applications that cannot use it effectively.

These applications include CAD, CAE, knowledge based systems, multimedia systems, graphic applications, statistical and scientific modelling and analysis programs.

All the above difficult database applications have one thing in common; they all demand the representation of complex structured data. The relational and other classical database models cannot express this demanded complexity.

The developers of these difficult applications, had therefore to resort to using their own developed systems in order to save and manage their data. This resulted in incompatibility, inefficiency and low productivity of such applications[9].

Hence, a new breed of databases have recently emerged to try and solve some of the problems of existing database technologies. These new databases have their roots in both object oriented concepts (analysis, design and programming) as well as classical database technology.

The Object Oriented Database Technology is still very young and is in the process of accelerated growth.

---

[9] This is similar to the state of commercial database applications somewhere in the 1960's prior to the development of the 'classic' standard data models.

The sections that follow first present the characteristics and salient features of this technology. A taxonomy of the different data model is then given, followed by some ODBMS implementation issues.

The current strengths and weaknesses of this technology are then presented as a conclusion.

6.2 Characteristics of ODBMS

In this section, the essential features of Object Oriented Database Management Systems will be presented. [ZDONIK, 90] [BROWN, 91]

The predominant requirement of an ODBMS is that it must be a Database Management System and as such it must provide the essential features and functionality expected of such systems.

Secondly, it must support the concepts of Object Orientation, as defined elsewhere in this dissertation.

The features and characteristics of a DBMS are given in chapter 2 of this dissertation. They are repeated below, as a summary and with particular reference to ODBMS's.

6.2.1 Essential Features

To qualify as a DBMS an ODBMS must have the following minimum features:

- **Model and Language:**

  A DBMS has a non-trivial model and language. That is, the DBMS understands some structure on the data it contains and provides a language for manipulating this structured data. This structure is often called 'data model'.

- **Relationships**

A DBMS can represent relationships between entities, the relationships can be named, and the language can query this relationship.

- **Permanence**

  A DBMS provides a persistent and stable store.

  Persistence, means that data can exist and is accessible past the end of the process that created it.

  Stable, means that data has some resilience in the face of process failure.

- **Sharing**

  A DBMS permits data to be shared amongst many users. This sharing does not have to be concurrent.

- **Arbitrary Size**

  The address space must not be constrained by limitations in the physical processor or the amount of processor memory.

6.2.2 Frequent Features

These features are not considered essential, but are highly desirable.

- **Integrity Constraints**

  A DBMS can help to ensure the correctness and consistency of the data it contains by enforcing integrity constraints. These are statements that must always be true for data items in the database.

- **Access Control**

  This provides security in terms of unauthorised database access.

- **Querying**

  A declarative query language is often provided such that easy access to data can be give to many users. In relational databases, the SQL query language is often given.

- **Report and Form Management**

  These are tools which are used to generate reports.

- **Distribution**

  Distributed databases have been developed such that data can be distributed over multiple computers, which could be geographically distant. In general, this provides improved availability, security and performance.

6.2.3 The ODBMS Threshold and Reference Models

The unique characteristics of an ODBMS are now presented with the aid of three models. [ZDONIK, 90]

The *Threshold Model* can be used as a yardstick to determine whether a DBMS can be considered to be Object Oriented. i.e. it presents the essential features of ODBMS's.

Although the Threshold Model presents the minimum features, a system with only these features would not be very useful. The *Reference Model* therefore is a proposal for a more 'complete' ODBMS.

These models are briefly presented below : -

6.2.3.1 The Threshold Model

As mentioned above, this model presents the minimum features required of a DBMS in order to be considered Object Oriented.

These features are:

- It must provide *database functionality*. This means it must comply with the essential DBMS features listed above.
- The database must support *Object Identity*.
- The database must provide *encapsulation*.
- It must support objects with *complex state*. The state of an object may refer to other objects, which in turn may have incoming references from elsewhere.

Inheritance is not considered to be essential, although it is a very useful property.

6.2.3.2 The Reference Model

A system satisfying only the threshold minimum requirements can be called ODBMS, but will not be able to give good support for most of the complex database applications.

The proposed reference model given below, attempts to define some more necessary properties which will enable an ODBMS to be applied successfully in many applications.

By definition, the Reference Model must include all the features of the Threshold Model including the following:

- **Structured Representation of Objects**

  The objects and their inheritance or class hierarchy must be represented and stored persistently

- **Polymorphism**

  The actual method executed at run-time for a message expression will depend on the type of the receiver of the message.

- **Collections**

The model has a collection of built-in types such as lists, sets and arrays.

- **Query Language**

  Ideally, this should be a declarative language. Since the object model, unlike the relational, does not have a mathematical basis, this feature is difficult to implement.

6.2.4 Other Threshold Models

Since the ODBMS is a recently conceived technology, complete standards are not available but many are proposed. The Object Management Group (OMG) has recently proposed an Object Model which could be used as a standard. This model is called the OMG Object Model and is presented in chapter 7. [OMG, 93] [CATTELL, 94]

Other proposed 'threshold models' that depict the minimum requirements for an ODBMS have been published in a paper during 1989 called "The ODBMS System Manifesto". [ATKINSON, 89]

6.3 Taxonomy of Database Data Models

6.3.1 Overview

All DBMS's are based on a specific data model. This data model provides the necessary abstraction and data independence.

Many data models have emerged to fulfil the need of a DBMS. These models can be categorised into two major groups (see Figure 6.1) [JOSEPH, 91]:

- Value-Oriented and
- Object Oriented

The sections that follow expand on these categories and specifically on the Object Oriented one.

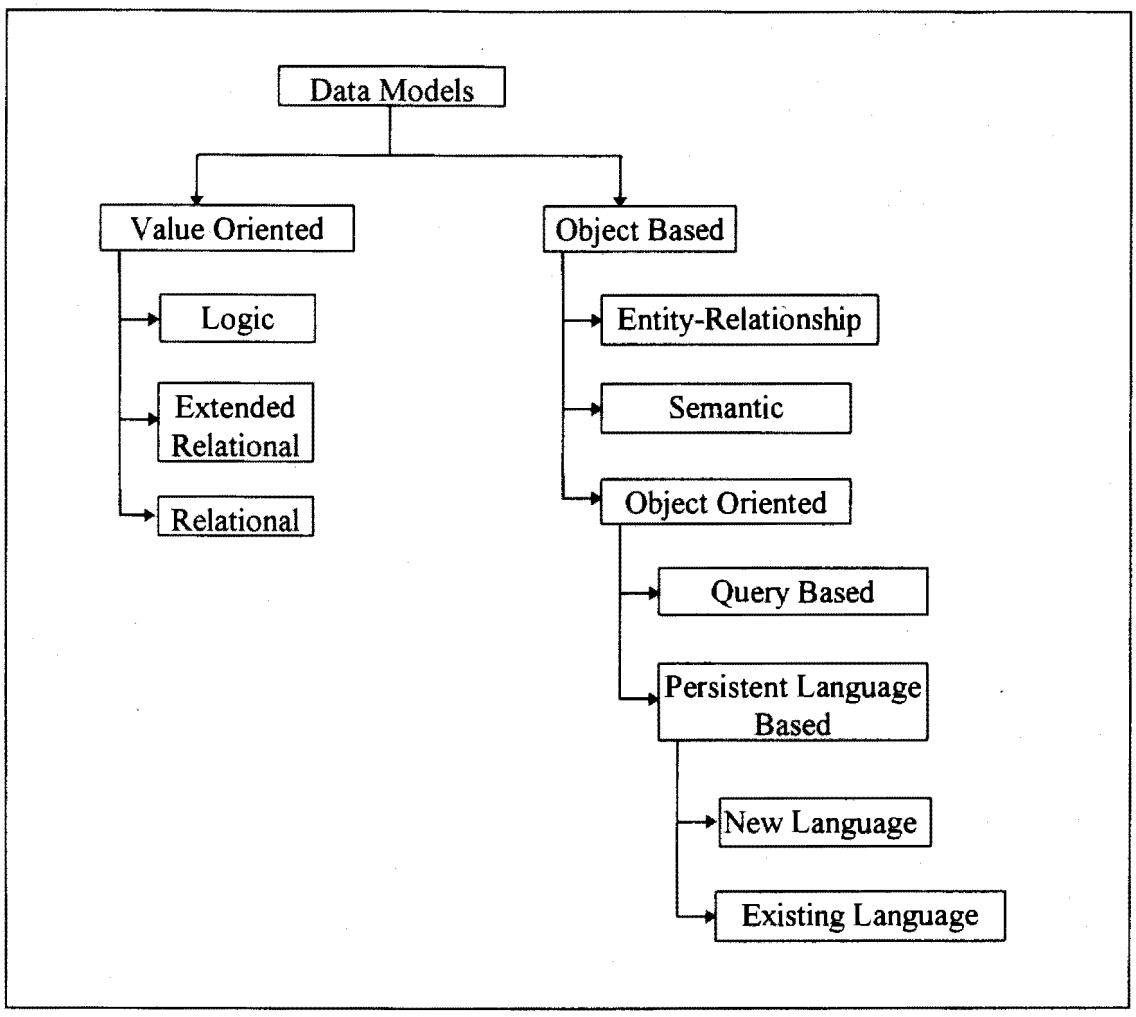


Figure 6.1 - A Taxonomy of Data Models

6.3.2 Value Oriented Models

In value-oriented data models, the relationships between objects are stored implicitly by comparison of values of attributes. [KIM, 90]

For example in the following relationships:

EMPLOYEE[id, name, ..,department] and

DEPARTMENT[department, name, ...]

a 'match' is made by comparing the *value* of attribute 'department' in the two relationships.

The Relational Model is an example of 'value-oriented' models. Other models include 'extended relational' and 'Logic'.

6.3.3 Object-Based Data Models

These data models have an inherent 'entity' or 'object' identity and hence relationships between two entities can be explicitly defined. e.g. in the above example, each employee object could reference the department object directly.

Object Based models can be further categorised as follows:

- Entity Relationship
- Semantic
- Object Oriented

6.3.4 Object Oriented Data Models

These models can be further split into two types:

- **Persistent Language Based .**

  The main objective of these models is to provide seamless integration with the application's programming language used. In order to achieve this integration, both the application language and the data model should use the same data types.

  The above can be achieved by either creating a new language with persistent object capability or augment an existing language with persistent storage capability.

  An example of the first method, that of creating a new language, is TRELLIS/OWL. [SCHAFFERT, 86]

  The method of providing object persistence by extending an existing language is more popular and has been used in the following commercial products:

  - ZEITGEIST, which is based on Lisp. [FORD, 88]

- GEMSTONE, which is based on Smalltalk. [GUPTA, 91]
- ONTOS, which is based on C++. [GUPTA,91]
- O2, which is based on C. [DEUX, 90]

- **Query Language Based.**

  In this approach, no attempt is made to provide a seamless integration with the application programming language. Instead, a set-oriented query language is provided to retrieve and update database objects.

  The main advantage of this approach is the provision of a declarative query language to perform database interactions. The disadvantage of this method is the 'impedance mismatch' obtained between application and database software.

  ODBMS which use the above approach are IRIS [FISHMAN,87] and POSTGRES. [STONEBRAKER, 90]

Recent ODBMS developments have incorporated the advantages of both schemes by providing a declarative query language within a persistent language-based ODBMS.

6.4 Object Programming and Database Management

One of the aspects of Object Technology is the way it blurs the boundaries between software disciplines within the SDLC. [LOOMIS, 93.5]

Object Technology is based on the use of a common, unified model in all phases of the SDLC. An object model developed during the analysis phase, is re-used in the design and implementation phases.

This synergy provided by the common model is one of the reasons that object technology can enhance and improve the efficiency of software development.

Object Database Management Systems (ODBMS) have their roots in both programming languages and database management systems. Programmers and database developers have different views and expectations of this technology. Some of these expectations are presented below: [LOOMIS, 93.5]

6.4.1 Programming Language Perspective

From the programmers perspective, the most important feature of an ODBMS is the provision of integrated support for persistent object storage.

In a programming language such as C++ , objects are created and destroyed in memory. This means that the lifetime of an object cannot be extended beyond the lifetime of the program. (unless they are stored on some other media)

The ODBMS provides the means of extending the life of objects beyond the termination of the program. Such 'permanent' objects are called 'persistent'.

There are several ways of achieving object persistent and they include the following:

- The use of files
- The use of a relational DBMS
- The use of Object Databases

1. File Systems

This solution requires the programmer to write code which transforms the objects to structures that can be written into files.

In addition, code must also be written to read these files and transform them back into objects. (See figure 6.2)

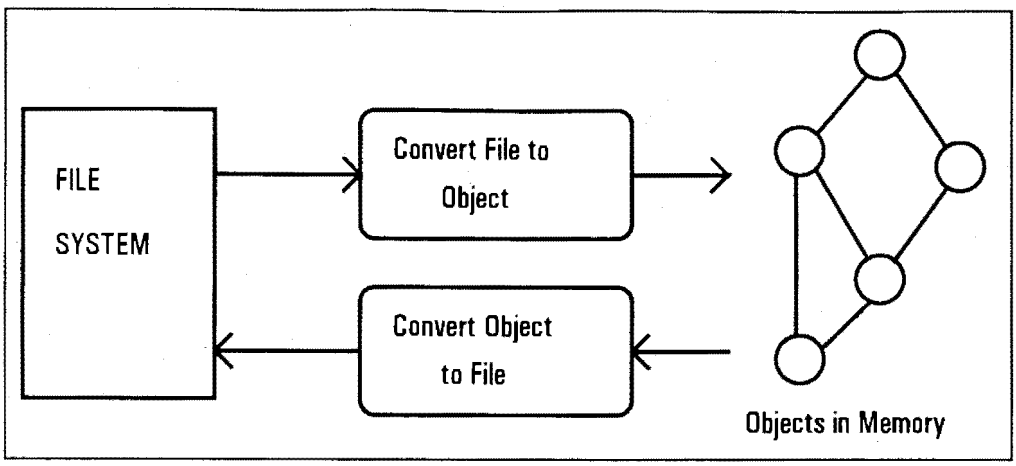


Figure 6.2 - Object Persistence using Files

The disadvantage of this persistence method is the extra code and hence cost and inefficiency that has to be implemented.

2. Relational Database

This persistence method involves the conversion of the object structure into a structure which is supported by the relational model. (see figure 6.3)

This is not a trivial task, as the relational model is far more constrained than the object model. Object have to be 'flattened' and normalised. This transformation will result in the loss of semantic information captured by the object model i.e. hierarchy etc.

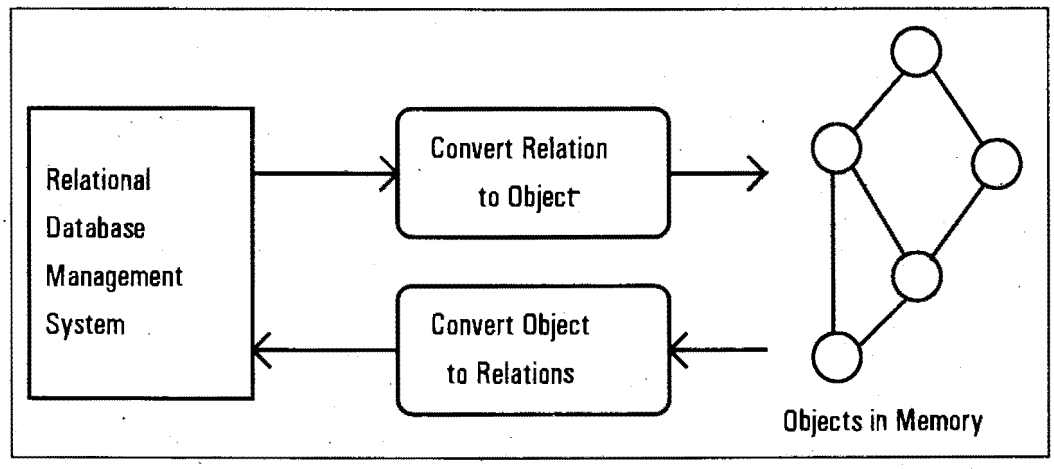


Figure 6.3 - Object Persistence using a Relational DBMS

3. Object Database

In this implementation, the programmer does not have to write any transformation code. This is because the ODBMS can understand the in-memory object & class structure.

The ODBMS uses the same representation as the programming language. Hence, not having to deal with SQL and object transformation to other structures, the productivity of the programmer is increased.

In addition, the developed system is bound to have less errors since errors resulting from transformations have been eliminated.

A further advantage of the Object Database approach is that type-checking can be performed across the system boundaries, whereas in the previous two methods, type information is not available once transformed and hence cannot by verified by the system.

4. Persistence with Seamless Integration

In order to enhance the use of persistent objects, an ODBMS interface must fit seamlessly into the programming language environment.

This means that to the programmer, the ODBMS must be 'invisible'.

ODBMS's achieve this tight integration by using the language of the environment as their sole interface. Hence Smalltalk is used as an interface to ODBMS's based on Smaltalk and C++ for databases based on C++.

This is in contrast to Relational Databases, which use SQL for data access and another procedural language, such as Cobol, as a host language.

This integration of object programming language and ODBMS's enables all objects to be treated with a single syntax , a unified model and one type/class system.

6.4.2 The Database Manager's Perspective

Database Managers usually deal with data abstractions only. Hence the most important feature they like to see in ODBMS's is the support of Object Models rather than Relational Models.

In addition, the database manager's perspective imposes the following requirements on ODBMS's:

- Concurrency Control
- Transaction Management
- Schema Management and Evolution
- Recovery
- Query Processing
- Access Control
- Database Administration Tools

Hence the database perspective, emphasises the need of ODBMS's to have the behaviour, functionality and resources of classical databases.

6.5 Persistent Database Architectures

6.5.1 Overview

There are two main methods of implementing object persistence [JOSEPH, 91]:

- Persistent Memory or
- Storage Server

Persistent Memory ODBMS's use virtual memory to store all the objects and rely on this memory to remain indefinitely in the execution environment of the application. This type of ODBMS will not be discusses further in this dissertation.

Storage Servers, which are also called '2-level storage environments', physically manage and store objects on secondary non-volatile storage facilities such as hard-disks.

If a network and a separate environment is used to execute the ODBMS, then the facility is also called 'client-server' , where the database is the server and the application software is the client.

6.5.2 Categories of Storage Servers

Storage Servers can be classified according to the following criteria[JOSEPH, 91] [WILCOX, 94]:

- Execution of Object Methods - client or server.
- Type of Interactions - Navigation or Query
- Unit of Transfer and Control
- Level of Semantics associated with Objects managed.

The following are the main categories of storage servers:

- **Passive** : Servers which do not execute the object's methods directly.
- **Active** : Servers which can execute the object's methods directly

The basic functions of these types of storage servers are depicted in figure 6.4.

- **Typeless Page Server**

  These servers manipulate pages of memory where instances of objects reside, instead of manipulating objects directly. In this way, they tend to be more efficient in their transfer speed.

  They do not understand any of the object's semantics and hence cannot execute methods directly.

- **Typeless Object Servers**

  These servers can manipulate objects directly. However, they only understand a minimum of the objects semantics, that of:

  - Object Identity (OID)
  - The fact that an object has a type
  - Inter-Object Relationships using embedded OID's

  This servers cannot execute the object's methods or access states directly.

- **Class Based Servers**

  These servers, in addition to object manipulation, can interpret and use the object's state to provide additional services i.e. queries.

  In order to address queries, they are usually built on top of relational databases.

- **Type Based Servers**

  These servers have the ability to manipulating individual objects and to execute the object's methods.

  This ability of accessing and executing methods allows the server to perform query processing and optimisation.

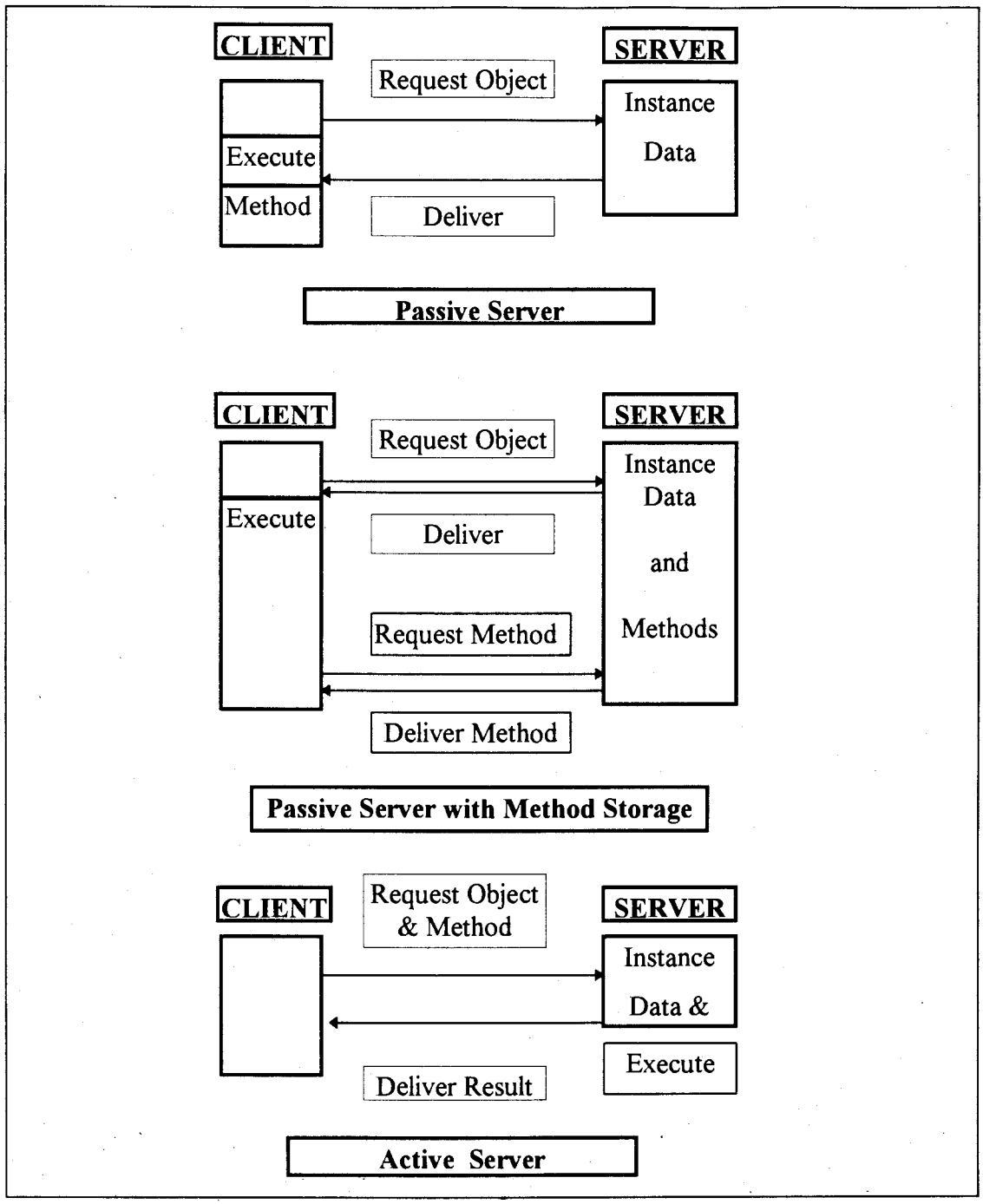


Figure 6.4 - Passive & Active Object Servers [WILCOX, 94]

6.6 Conclusions - ODBMS Strengths and Weaknesses

ODBMS's have the following Strengths and Weaknesses with respect to the 'Classical' DBMS's. [KIM, 93]

6.6.1 Strengths

- An object can encapsulate both state and behaviour.

  In contrast, a tuple in a relational database cannot hold 'behaviour'.

- Inheritance and classification can be used to express the structure of complex data.

  The relational model allows only 'flat' relationships, which completely eliminate data semantics.

- The domain of an attribute of a class may be any class. This allows complex objects to be represented in a similar manner to real-life objects.

  In relational databases, the domain of an attribute is restricted by definition, to a small set of primitive types and an attribute may have only one value.

- An ODBMS's language is the same as the host language.

  In contrast, the declarative query language of a relational database (SQL) is different to the host (procedural) language used. This is called an 'impedance mismatch'.

  Since the language used in ODBMS is homogeneous, the efficiency, quality and integrity of the software generated increases.

6.6.2 Weaknesses

- The ODBMS's are not based on formal mathematical theories or models.

  In contrast, the relational model is based on formal mathematical principles. This enables the use of declarative query languages that can be formally proved to be correct and can be optimised.

- The ODBMS's lack standardisation and are immature. (This is currently been rectified by the efforts of the Object Management Group. [OMG, 93])

  The relational model has been established as an industry standard and has matured to the degree whereby few experts will dispute its power.

- Object Models are inherently more complex than the simple table relational models.

  Their complexity, however, allows the representation of semantic information, which the relational model cannot represent.

Current indications are that the state of this technology and in particular the weaknesses, could change in the near future. Some of the emerging ODBMS standards are presented in the chapter that follows.

7. EMERGING ODBMS STANDARDS

7.1 Overview

One of the major weaknesses of Object Databases is the non-existence of standards. In order to rectify this deficiency, a number of authorities and organisations have undertaken the creation and proliferation of standards. Some of these standards are listed below:

Table 1 - Object Technology Standards

Abbreviation	Standard	Organisation	Description
COM	Common Object Model	Microsoft	Storage of Object
CORBA	Common Object Request Broker	Object Management Group (OMG)	Distributed Object Standard
OLE	Object Linking and Embedding	Microsoft	Component & Compound Document Software
ODMG-93	Object Database Management Group-93	Object Database Management Group & OMG	Standard of Object Database Management Systems
SOM	System Object Model	IBM	Object Model CORBA compliant
OMG Model	Object Model proposed by OMG	OMG	Object Model, modified by ODMG

7.2 The Object Management Group

The lack of standards in the Object Oriented Database Technology is one of the factors that retard its acceptance in the industry.

The Object Data Management Group was formed as an independent organisation to propose and promote ODBMS standards. [OMG, 93] [LOOMIS, 93.6]

This organisation has recently published a proposal for a standard to be adopted. This standard is called the 'OMG Object Model'.

The OMG Object model specifies a standard model for the semantics of database objects.

This standard model is important because it determines the built-in semantics that the ODBMS understands. Adopting a standard model ensures that the design of class libraries and applications that use the model are portable across the various ODBMS products.

The Object Database Management Group (ODMG), which is a working committee of the OMG, was formed during the year 1991 with the exclusive purpose of accelerating the development of ODBMS standards. They subsequent succeeded in publishing a standard during the year 1993. This standard is called ODBMS-93 and some of the details are presented below.

7.3 The ODMG-93 STANDARD

7.3.1 Overview

This standard was developed by the Object Database Management Group (ODMG) , which was formed for the exclusive purpose of developing a standard for Object Database Management Systems (ODBMS). [CATTELL, 94]

7.3.2 Architecture

The major components of the standard are as follows:

- **Object Model**

  The OMG Object Model was used as a basis for developing this Object Model.

- **Object Definition Language (ODL)**

  ODL is used to define the schema of the database. This language is implementation independent and can be transported into any programming language or development platform. Currently only C++ and Smalltalk bindings are supported.

- **Object Query Language (OQL)**

  This is a declarative language used for querying and updating database objects. The relational SQL standard was used as a basis for developing this language.

- **Object Manipulation Language (OML)**

  C++ and Smalltalk is available as an OML. The OML is in fact the language used for programming the application, hence offering a seamless integration of database and application programming.

7.3.3 Development

Figure 7.1 outlines the development process of an application using this database to manage the persistent storage and retrieval of objects. [CATTELL, 94]

First the programmer writes declarations for the database schema (both data and interface) and a source program for the application implementation.

The source program can be written in C++ or Smalltalk and the database schema in ODL.

The declarations and source code are then compiled and linked with the ODBMS to produce a running application.

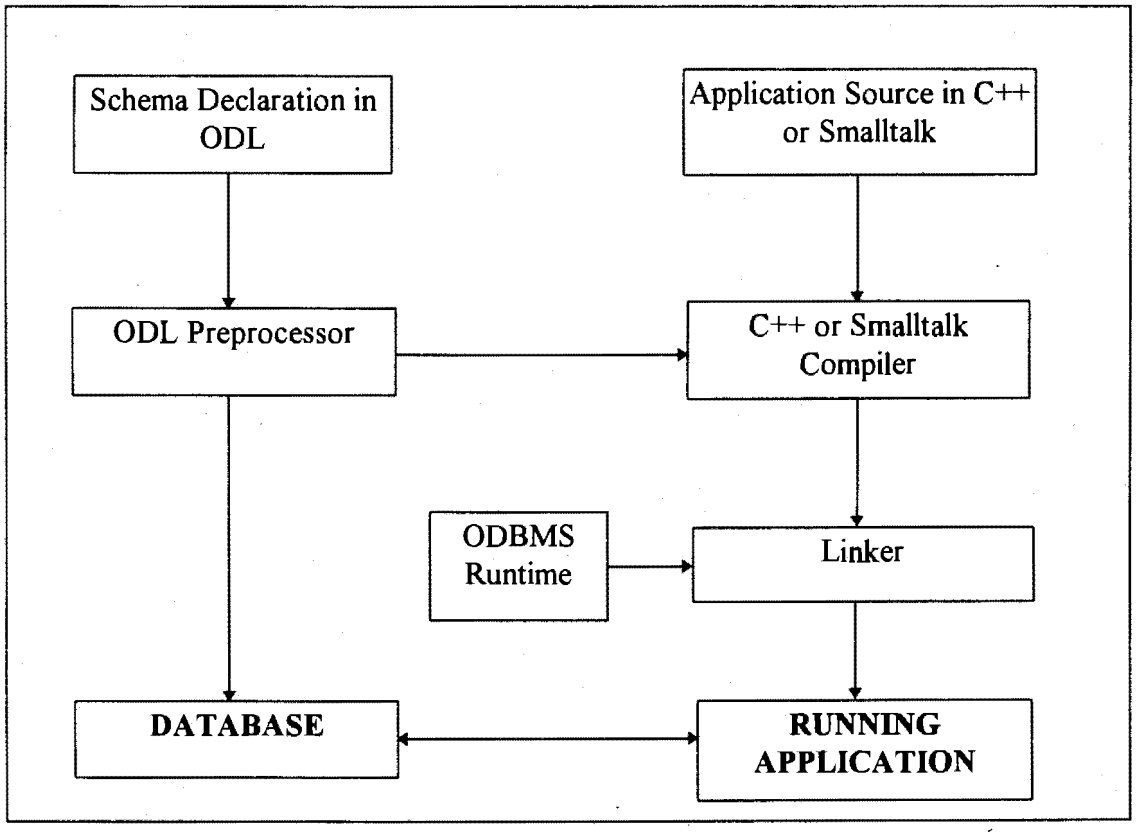


Figure 7.1 - Developing Applications with ODBMS [CATTELL, 94]

7.3.4 Basic Elements of the Object Model

The ODMG Object Model defines the following basic semantic elements:

- Objects : these are the basic modelling primitives.
- Types : these are categories of objects

- Behaviour : the set of operations that can be performed on objects
- State : these are the values objects carry for a set of properties

Two basic kinds of objects are distinguished:

- Mutable Objects and
- Immutable Objects

The value of a Mutable Object's property may change. In contrast, an Immutable Object's property value cannot change.

Immutable Objects are also called 'literals' and include objects of type integer, float, character etc.

Mutable Objects are also referred to as simply 'objects'.

When it is necessary to refer to both kinds of objects, then the term 'denotable object' is used.

All denotable objects have unique, immutable identity.

7.3.5 Object Lifetime

Every object has a lifetime which must be specified at creation time. The lifetime of the object cannot be changed.

The intent of the Object Model is to enable object programmers to deal with all objects in a manner consistent with the programming language conventions, regardless of the object's lifetime.

Three lifetimes are recognised by the object model as follows:

- **Coterminus with Procedure**

  This object exists only as long as the procedure that created it exists.

- **Coterminus with Process**

  The object exists as long as the program or process that created it exists.

- **Coterminus with Database**

  The object's storage is allocated from the segment, page or cluster managed by the ODBMS runtime.

7.3.6 Object Properties and Operations

The following properties are defined on type 'Object':

- has_name? which is either True or False
- names a set of strings
- type the name of the object's type/class

Type 'object' also has the following operations:

- create

  This operations takes a list of property_name, property_value pairs and allocates storage for the representation of the object.

- delete

  Deletes an object and frees the storage used by the representation.

- exists?

  Tests for an objects existence

- same_as?

  Tests to see if two names refer to the same object.

7.4 Conclusions

The wide acceptance of ODBMS standards by the industry is necessary for the success and proliferation of this technology. The OMG has been responsible for pursuing and supporting this task.

A working committee of the OMG, named ODMG, has published an ODBMS standard which is called the ODBMS-93.

8. PERSISTENT OBJECTS IN C++

8.1 Introduction

As discussed in chapter 6, one of the ways of achieving object persistence is to extend an existing object oriented language by means of pre-defined classes.

The C++ language does not support the ability to store objects created at run-time directly to a persistent store. The only mechanism supported is that of serialising data and storing it to disk.

The sections that follow describe different methods of extending the C++ language, such that object persistence can be achieved. The choice of method will depend on what needs to be achieved.

8.2 Extensions to C++ for Object Persistence

8.2.1 Overview

The method used will depend on the complexity of the objects to be saved. For example, if the objects contain only simple data types e.g. integers and characters, then the existing mechanisms will suffice, if however, the objects contain pointers to other objects, then more complex mechanisms have to be developed, as shown in the sections below.

8.2.2 Objects with Simple Data Structures

If the objects that are required to be made persistent contain only simple data structures, then persistence can be achieved simply by serialising and storing the object's data members to disk. [SHILLING, 94] [LAURENT, 93] [STEVENS, 94]

Simple data structures are integers, float values and characters.

As an example, in the following class:

```
class simpleClass
{
    public:
        int getNumb();
        int setNumb(int);
    private:
        int Numb;
};
```

simpleClass simpleObject;

In the 'simpleClass' above, persistence can be achieved by saving the private data member 'Numb' to disk before the program is terminated. The member can then be restored when the program is re-executed.

This disk 'read/write' operation can be further simplified if a base class (called 'persist') can be created which holds the relevant member functions. Any class that requires to be persistent can then inherit from this 'persist' class. For example:

```
class simpleClass public: persist
{..}
```

The object can then be written or read from disk as follows:

```
simpleObject.read(); or
simpleObject.write().
```

8.2.3 Persistent Complex Objects

Complexities arise in saving and restoring objects when these objects contain complex structures and pointers.

The types of problems that can be experienced are as follows:

- If the object contains a list of data values, then when this objects is read back, the memory space required needs to be dynamically obtained from the operating system, prior to re-initialisation of its data values.
- If the object contains pointers to other objects, then when the object is read from the store, the pointers need to be re-initialised such that they point to the relevant existing objects. Hence the system must provide pointer persistence as well as data persistence.

The various techniques of dealing with complex object are described below.[SHILLING, 94][LAURENT, 95]

- **Flatten Structure**

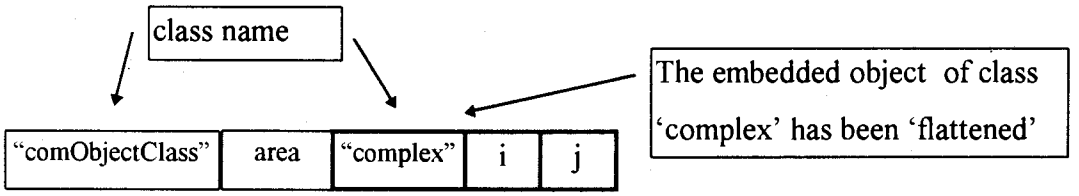
  This is a simple technique which can be used to avoid the problems mentioned above.

  The complex structure is 'flattened' such that it can be serialised and saved in the usual manner onto disk. For example, consider the following object which contains an embedded object.

```
class complex
{
    ..
  private:
      int    i;
      int    j;
};
```

```
class comObjectClass
{
    ..
    private:
        complex cNumber;      // embedded object of type complex
        float    area;        // float value
};

comObjectClass complexObject;
```

The object complexObject can be saved to disk as follows:



In the example above, the class name is also saved such that the system can identify and instantiate each class during the 'reading' phase.

- **Include the Object's size.**

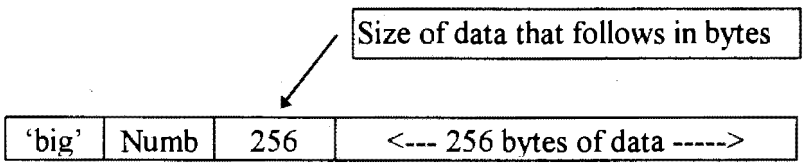
If data from the heap needs to be saved, then the memory must be re-allocated prior to performing a disk 'read' operation. This can be achieved by saving an additional integer value which indicated the number of bytes that are required to be allocated. As example follows:

```
ıs big

.

ɔrivate:
```

```
        int     Numb;
        char*  pHeap;  //pointer to heap
        ..
};

big BigObject;
```

The object 'BigObject' contains data which has been dynamically allocated form the heap.

Saving of this data can be achieved as follows:



When the system reads this back from disk, the correct memory allocation can be achieved before the object is re-initialised .

- **The Object Identifier (OID) Approach**

This technique can be used to handle pointer persistence. An OID is a system generated number that identifies each persistent object in the system uniquely. [VADAPARTY, 95]

In this approach, the system handles pointers to persistent objects differently to ordinary 'transient' pointers. When a persistent pointer is de-referenced, the system checks to see if this object exists. If the object referenced does not exist, then it is explicitly called from the persistent store. (see figure 8.1)

- **The Virtual Memory Mapping Approach**

  In this approach, the system treats all pointers as if they are transient. If the object does not exist in the system's memory pool, then a hardware interrupt is generated.

  The interrupt routine will then be responsible for loading that object to the memory pool.

Although the two persistent pointer methods above achieve the same objectives, their implementation is different. The OID approach is implemented using the C++ language features whereas the Virtual Memory Mapping approach depends on Operating System support, and hence is not fully transportable to other systems.
The OID approach is described in more detail below.

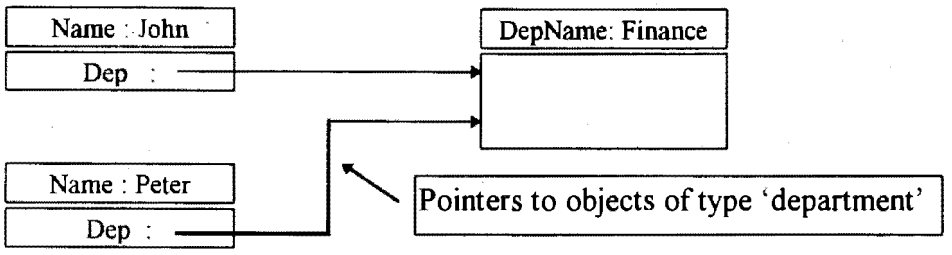
8.2.4 Pointer Persistence using the OID Approach

As described briefly above, this is a technique that can be used to achieve pointer persistence.[VADAPARTY, 95]

Before the details of this approach are presented, the requirements and objectives of pointer persistence are first given:

- When objects that contain pointers to other objects are recalled back from store, the system must initialise these pointers such that they point to the correct objects.
- If the referenced objects do not exist, then the system must take appropriate action. i.e. either inform the user appropriately or load the missing objects back into the memory pool. (also called object pool below)

The following 'employee-department' relationship highlights these requirements:



In the example above, two 'employee' objects are referencing the same 'department' object.

The issues to be considered with respect to the above example are the following:

- Flattening the structure of John and Peter will necessitate the saving of the department object 'Finance' twice. This introduces problems with updating the department object, that existed prior to the conception of databases.
- Saving the actual pointer contained in John and Peter is meaningless, since pointers refer to physical memory addresses that are different for every run-time environment. i.e. there is no guarantee that the next time this program will run, the physical addresses will be the same.
- How do we guarantee that the object 'Finance' exists prior to calling member function from John and Peter?

The method explained below addresses all the above issues.

1. The Object Identifier (OID)

Each persistent object is given a unique identifier, normally called the OID. This identifier is created by the system. (see figure 8.1) [VADAPARTY, 95]

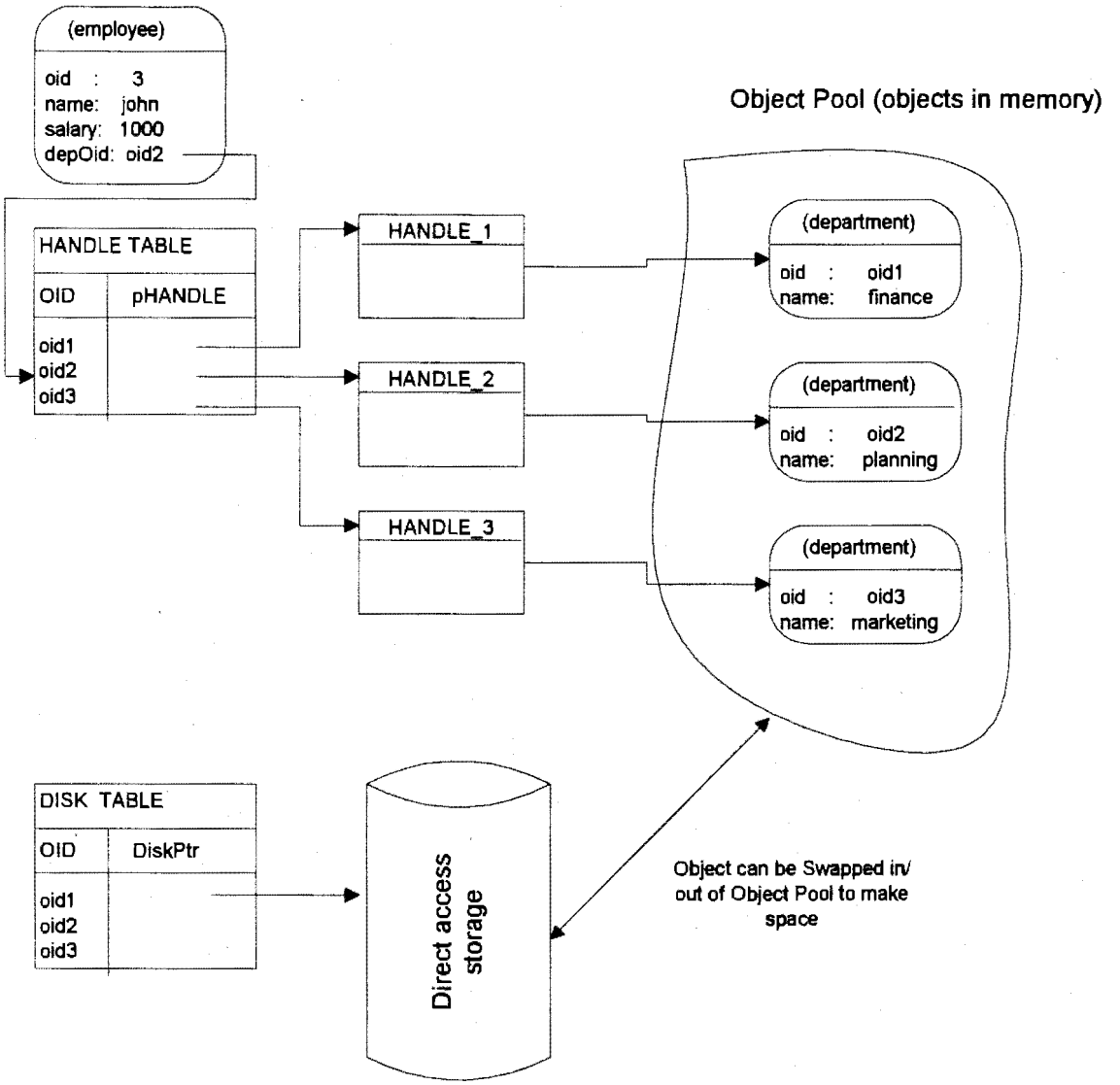


FIGURE 8.1 PERSISTENT POINTERS USING THE OID APPROACH

Once the OID is created, objects can use it to reference other objects. e.g. for the employee-department relationship above, an OID can be used in the employee object to reference a department object, instead of using a physical address.

The physical addresses to the actual object in the object pool can then be obtained indirectly by the use of the 'Handle Table' or the 'Disk Table', depending on whether the object is in the memory pool or in the persistent store.

The two tables created by the system are described below.

2. The Handle Table

This contains a pointer to a handle per OID that exists in the system. If the object does not exist, then the pointer value is a 'null'.

The 'handle to the Object' contains the following additional information:

- A pointer to the object that exists in the memory space. (also called object pool)
- Additional information such as a reference count.

3. The Disk Table

This contains the physical disk address of each object in the persistent store. It is used to access an object directly from store.

4. The Object Handle

Each persistent object that has been called into the object pool by the system has a handle. This handle is used by the system for the following reasons:

- **Managing the Object Pool**

As objects get called and created from the persistent store, the system's memory can become congested with no available space for more objects. These will necessitate the removal of objects that are not currently referenced back into the persistent store.

Instead of removing the 'handle', however, the physical disk location of the referenced object is stored as a data member of the handle.

When access to the object is again requested, the system can obtain the physical location directly, rather than searching the Disk Table.

- **Reference Counting**

  The handle keeps track of how many users are currently accessing this object. If the reference count is zero and memory from the pool is required, then this object is picked for transfer back to the persistent store.

The operation of the OID approach is then as follows:

Instead of saving direct pointers to object that need to be referenced, the system saves instead the OID of that object.

For example, in fig 8.1, the 'employee' object references the department by means storing the OID of that department in its private data.

When the pointed object needs to be accessed, first the 'Handle Table' is referenced to find the relevant pointer to the handle, and then the pointer to the actual object, existing in the object pool, is located.

If the pointer to the handle is nil, then a further search is made in the Disk Table. This will obtain the physical address of the object in the persistent store. The object is then loaded into the object pool for access and manipulation.

8.3 Persistence in C++ - Strengths and Weaknesses

This section has described a number of techniques in introducing object persistence in the C++ programming language.

Using an existing language, such as C++, for object persistence has the following strengths and weaknesses:[PATON, 96]

1. Strengths

- Existing programs can use persistent object techniques without major modifications.
- Programs and persistent data can be organised using the same language, hence avoiding the 'impedance mismatch' problem described in other sections of this dissertation.
- The low-level nature of such systems can result in very efficient code, especially for time-critical and real-time applications.

2. Weaknesses

- There is no support for 'query-languages', hence no access to data by non-programmers.
- The methods used, as is evident from the above sections, is low-level and hence time consuming for programmers to implement.
- There is limited support for data independence, since it is up to the programmer to define how the data will be physically stored on disk.

From the above, it is evident that using persistent objects in C++ is applicable for applications that contain embedded databases which can be completely hidden from the user and where data access must be fast and efficient.

Since the language provides very limited support for implementing persistent objects, it is highly advisable to implement and adhere to international standards. The Object Database Management Group (ODMG) has developed such a standard, which is called ODMG-93.

8.4 *The Employee-Department Application*

8.4.1 Overview

This is a simple application developed to illustrate the technique of using object persistence in storing objects directly to disk. The application was developed using the programming language C++.

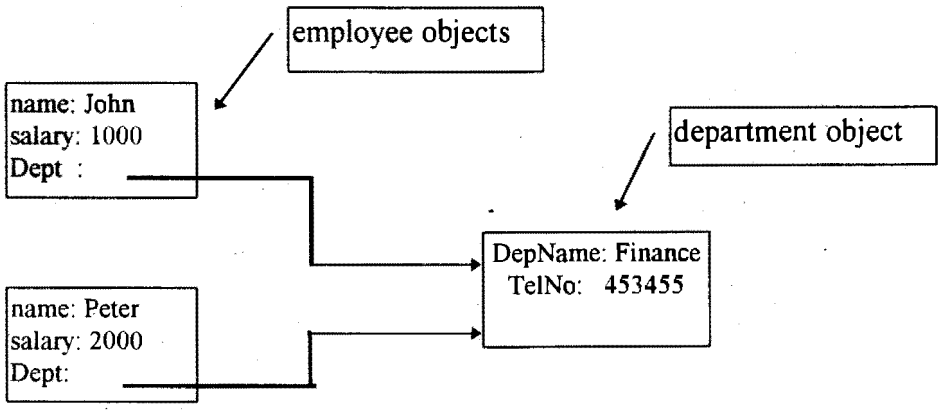
First the generic classes 'persist' and 'dbase' are developed to support object persistence.

The employee-department relationship is implemented using pointers to objects. Pointer persistence is implemented using system generated Object Identifiers (OID's).

The Borland C++ Version 4.0 running on the Windows-95 environment was used, although any advanced implementation of C++ can be used to compile the source code, provided the compiler supports 'run-time type identification' (RTTI).

8.4.2 The Employee-Department Relationship

The relationship is implemented using a pointer from the employee to the department that the employee is working in. This can be shown as follows:



The relationship is implemented by means of one pointer in each 'employee' object. The implementation of these persistent pointers is described in the sections that follow.

8.4.3 The Persist Class

All classes in the application that require to be persistent must be derived from the pre-defined 'persist' class.(see figure 8.4)

This class defines the following virtual functions[10]:

```
virtual void   write(opstream&)
virtual void   read(ipstream&)
virtual char*  className()
```

The above pure virtual functions need to be re-defined on every child class. The function must perform the following:

write()	This function must serialise the object's data members and store them to disk.
read()	This function must first read the data members from disk and then initialise the object's data appropriately.
className()	This function returns that object's class.

The read/write operations are performed by overloading the operators << and >>. [LAURENT,93]

The persist class holds a number, which is called the OID, as a private data member.[VADAPARTY, 95]

The OID can be read by any external function but no object has access to it except the 'dbase' object which is defined as a 'friend' to the persist class. A class defined as a

---

[10] An explanation of C++ virtual functions can be found in the following reference: [LIPPMAN, 91]

friend can gain access to the private members, whereas any other class has no access. Hence the class 'dbase' can access all the data members of the 'persist' class.

8.4.4 The Dbase Class

The following are the most important functions defined for this class:

openDbase()	opens the database with the given file name (DOS file name) and reads all the objects in it. The objects are then created dynamically.
saveDbase()	Saves all the persistent objects to disk.
insertObj(persist*)	Inserts an object of type persist to the database.
getPointer()	Returns a pointer to the object with the given OID.

The private data members are as follows:

fileName:	this is the name of the DOS file used for the database
maxObjects	this is an integer whose value is the number of persistent objects that currently exist in the object pool (or system memory)
persist* pPObj[]	This is an array of pointers to the class persist that relate the object's OID to the objects pointer in the object pool.

8.4.5 Implementation of Persistent Pointers

The OID approach is used to implement the persistent pointers in the employee class. This means that the physical pointer which points to the relevant department object is replaced with an identifier called OID. (see figure 8.2)

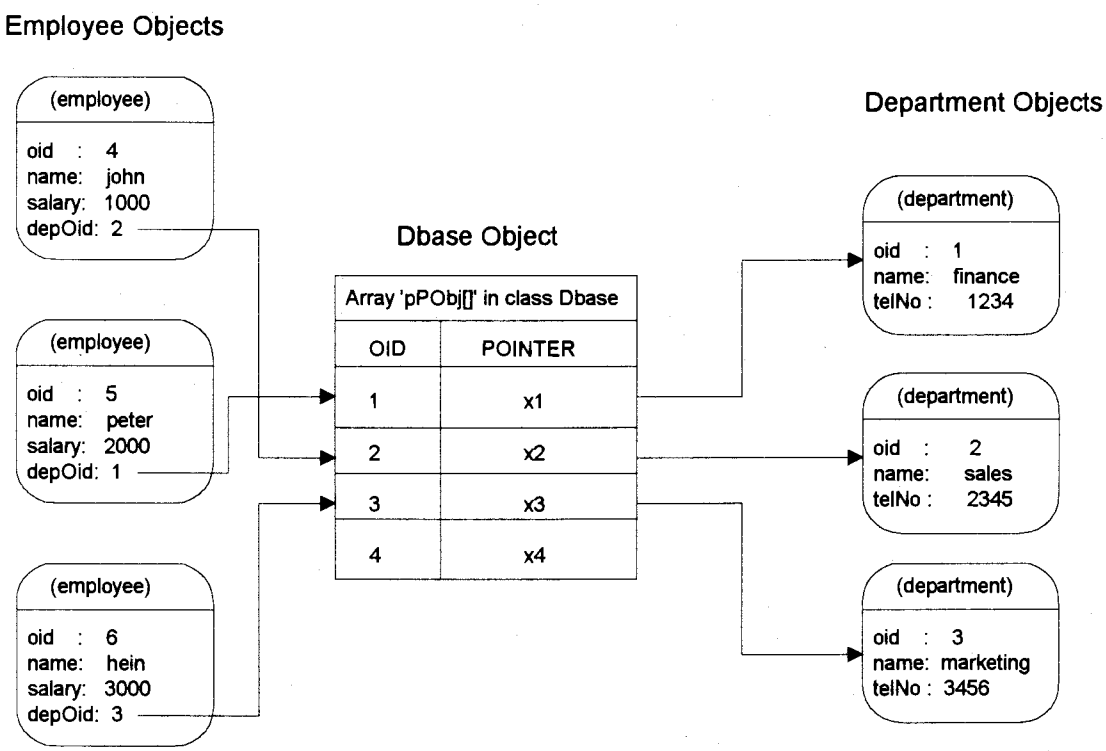


Figure 8.2 Instantiation of the Department and Employee Classes -Showing how the OID/Pointer table is used to define relationships

As described in section 8.2.4, a table is used to relate the OID to be referenced with its physical pointer (i.e. physical address in the object pool)

The technique, however, has been simplified by not implementing 'object handles' and 'disk tables'. Hence no memory optimisation has been implemented and the program assumes that all objects can reside in memory simultaneously.

The table that relates OID's to addresses is in the generic 'dbase' object and is called pPObj[].
This is simply an array of pointers to the persistent objects that exist in the object pool.

8.4.6 Physical Storage Structure

The storage structures are shown in figure 8.3. The class name as well as the OID of the object to be stored is saved on disk prior to saving the object's data members.

The class name and OID is saved automatically be the generic classes 'persist' and 'dbase'. i.e. this is hidden from the programmer.

When the object is read from disk, first the object is created dynamically using the class name read.

The OID table is then initialised with the physical address of the object created. The object's data members are then initialised according to the data read from disk.

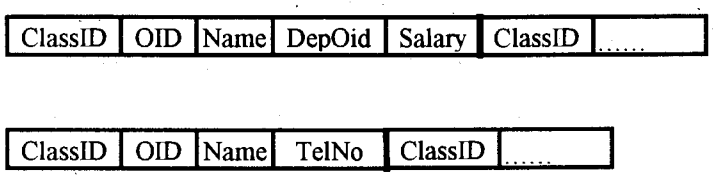


Figure 8.3 Employee and Department Object Physical Object Structure

8.4.7 Running the Application

In order to demonstrate the use of this limited application, a simple menu has been implemented. (see figure 8.5)

The user must first enter the department data prior to the employee data. When the user enters the employee's department, the data is rejected if the department object does not exist. This ensures referential integrity.

Once the data has been entered, the user can save this data for later usage.

Printing the Employee and Department data is also possible. In the example execution, the object's OID is also printed.

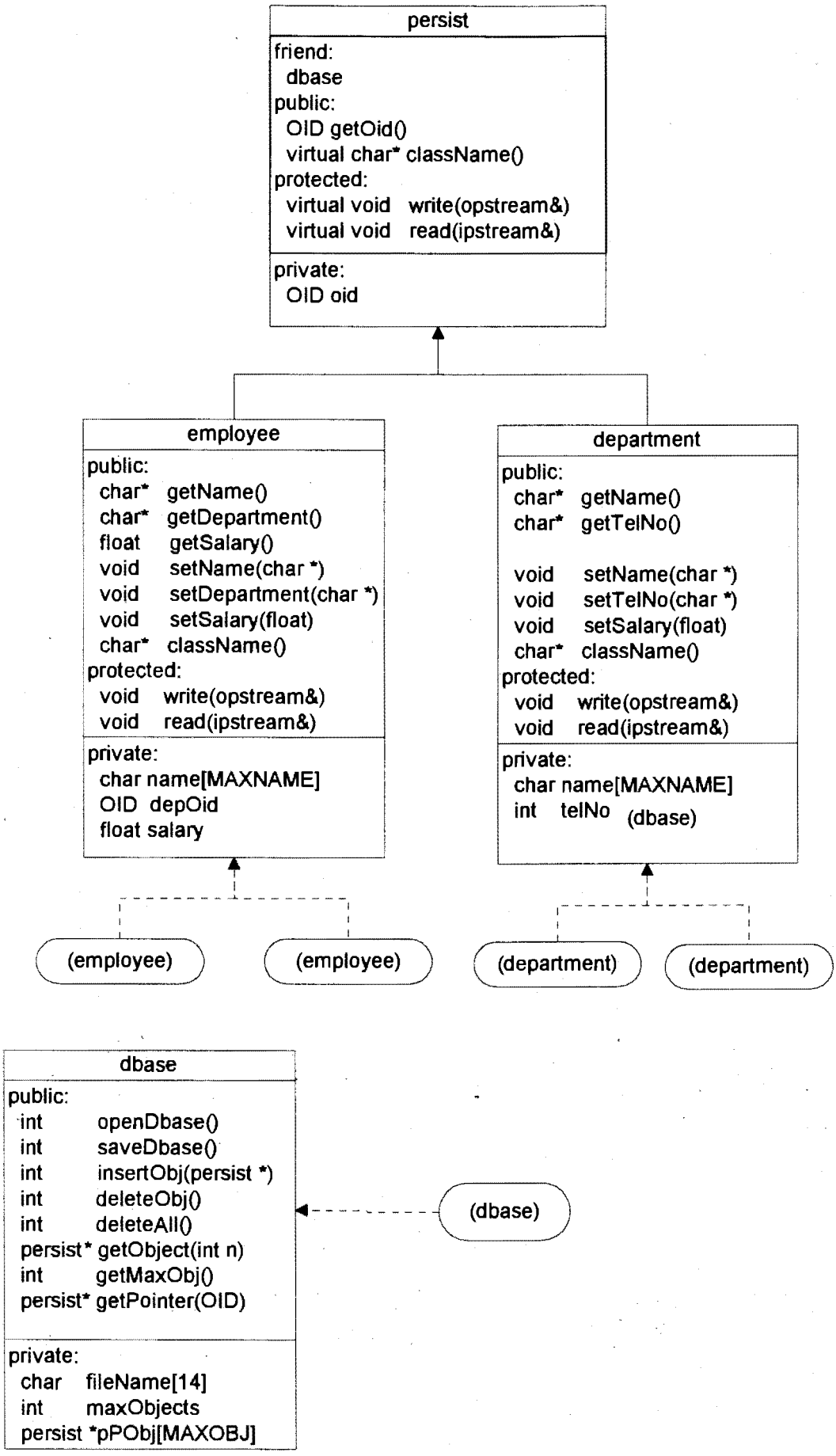


Figure 8.4 Employee and Department Application

```
C:\COS95\SIM\EMPL2.EXE
1. Enter Departments
2. Enter Employees
3. Save Database
4. Read Database
5. Print Employees
6. Print Department
10. Exit
6

        DEPARTMENTS
     Name        OID      TelNo
  finance          1       1234
    sales          2       2345
marketing          3       3456

     MAIN MENU
1. Enter Departments
2. Enter Employees
3. Save Database
4. Read Database
5. Print Employees
6. Print Department
10. Exit
```

```
C:\COS95\SIM\EMPL2.EXE
1. Enter Departments
2. Enter Employees
3. Save Database
4. Read Database
5. Print Employees
6. Print Department
10. Exit
5

            EMPLOYEES
     Name        OID      Depmt     Salary
     john          2      sales       1000
    peter          1    finance       2000
     hein          3  marketing       3000

     MAIN MENU
1. Enter Departments
2. Enter Employees
3. Save Database
4. Read Database
5. Print Employees
6. Print Department
10. Exit
```

Figure 8.5 The Employee Application

8.5 The Real-Time Simulation Application

8.5.1 Overview

An area where persistent objects and ODBMS's are useful is in engineering application. In these application, the data is more complicated than the typical commercial applications, such as the 'employee-department' example given above.

An example of such an application is that of real-time simulators. Here the objects represent complex structures such as conveyors, vibratory feeders and silos of specific geometric shape.

The application described below is a real-time simulator of limited scope.

8.5.2 The Simulator

The simulator implemented can be used to simulate the real-time operation of a typical materials handling plant, such as a coal supply system in a colliery.

The following pieces of equipment (objects of the simulation) can be created:

- **Source**

  This object can supply material at a fixed rate. Flow rates are expressed as kg/sec but any unit can be used.

- **Silo**

  Only a cylindrical silo is implemented, but by using inheritance, any other shape can be easily implemented by deriving it from the base class 'silo'.

  The silo has the following attributes:

  level : which is dynamically determined by the simulator
  diameter : which must be supplied by the user
  height : which must be supplied by the user

overflow : a Boolean attribute indicating an overflow condition

- **Feeder**

A feeder feeds material at a rate determined mathematically by the simulation

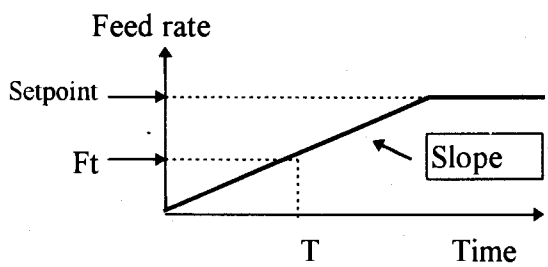
The feeder has the following attributes:

maxFlow : the maximum possible flowrate

slope : this determines the linear characteristic of the controller

setpoint : the setpoint of the controller determines the final desired feedrate

The actual feedrate is calculated as follows:



Ft is the calculated feedrate at time T.

- **Conveyor**

The conveyor has the following user defined attributes:

speed : the fixed speed of the conveyor in [m/s]

length : the length of the conveyor in [m]

The system calculates the delay and the feedrate at the end of the conveyor relative to time. This is implemented in the simulator by means of a shift register, which is a private array in its class.

8.5.3 The Simulator Implementation

A simple menu has been implemented as a user interface in order to demonstrate the application. (see figure 8.7 and 8.10)

First the user enters the objects to be simulated. The order of entry determines the linear flow of material. It is assumed the material flows from one device to the other. e.g. that a feeder feeds only one conveyor.

The above limitation implies that the relationships between the objects can be simply determined at run-time by following the sequence. Hence no complex relationships need be saved persistently on disk.

Hence, the application is simplified by not implementing OID's as in the employee-department application presented in the previous section.

The persist and dbase classes are used in a similar fashion to the one described in the previous section.

The class of its object is also saved on disk followed by the object's data members that need to be persistently stored. (see figure 8.6 and 8.9)

feeder	coke	100.0	20.0	0.5	silo	silo_a	0	10.0	50.0	....

Figure 8.6 Structure of disk file after the feeder and silo objects are saved

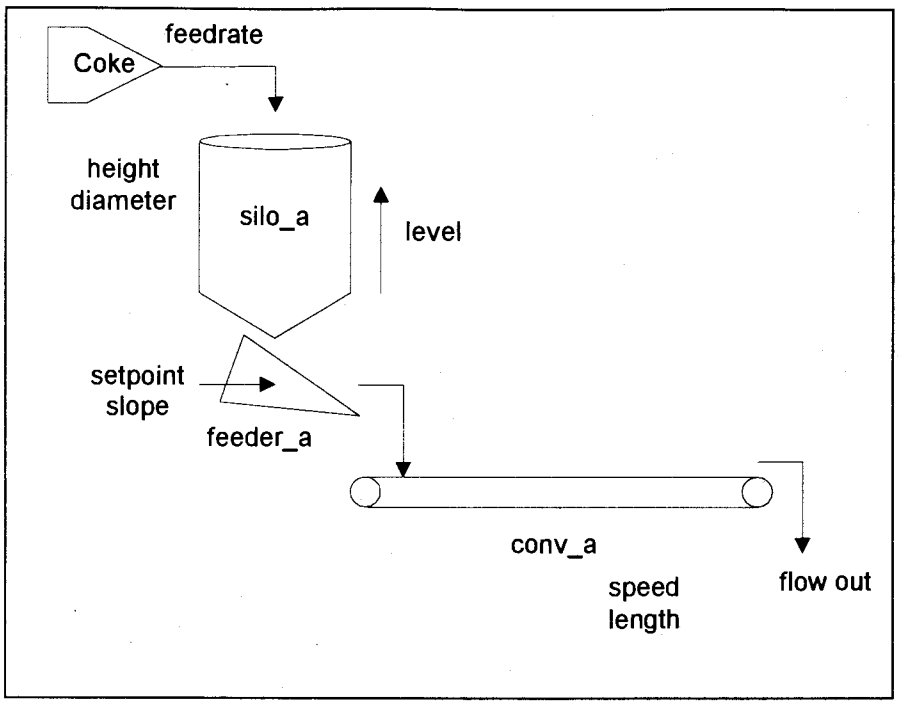


Figure 8.7 Simulation of a Coke Supply System

8.5.4 Conclusion

The simulator application represents a set of applications that can benefit from this persistent object technology. The data cannot easily be converted to relational tables, especially when the application is expanded to more complex objects by using inheritance hierarchies.

Furthermore, there is no need for non-programmers to query the database, as is the case with most commercial applications.

This is therefore a data intensive application, which requires the persistent storage of complex objects that do not have to be directly queried by the user.

For a large plant, the simulator might be required to handle thousands of objects which implies that efficiency in both memory space and speed is required.

From the above it is evident that applications of this category are particularly applicable to this persistent object technology.

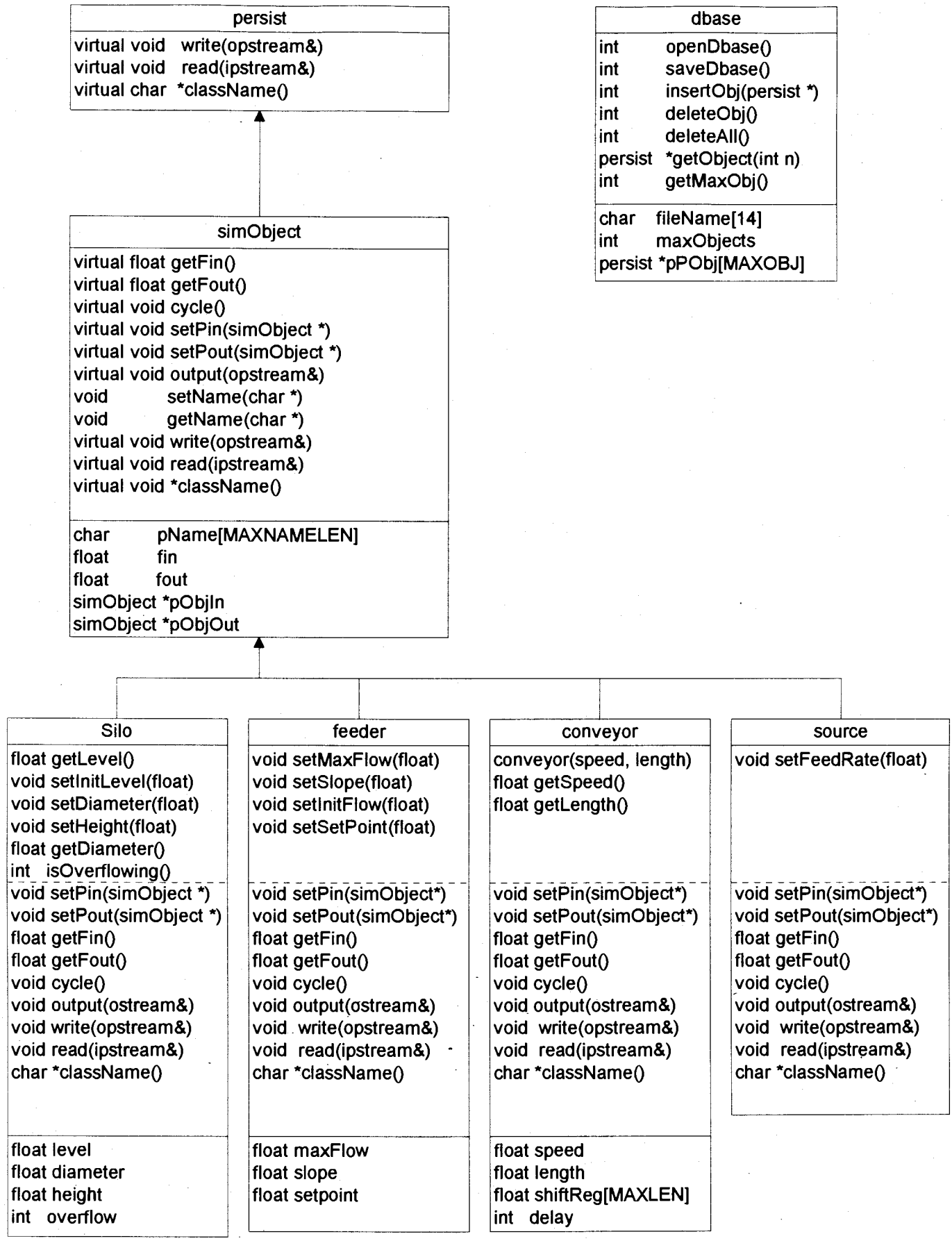


Figure 8.9 The Simulation Application Object Model

The Objects currently in Simulation are:

name : .....coke
feedrate: 100.00

name : .....silo_a
level : 0.00
diamt : 10.00
height: 50.00

name : ....feeder_a
maxFlow : 100.00
slope: 20.00
setpoint: 0.50

name : .....conv_a
speed : 1.00
delay: 10
length: 10.00

Simulation Results:

Time	Name	Fin	Fout	Level
1	coke	0.00	100.00	
1	silo_a	100.00	0.00	1.27
1	feeder_a	20.00	20.00	
1	conv_a	20.00	0.00	
2	coke	0.00	100.00	
2	silo_a	100.00	20.00	2.29
2	feeder_a	40.00	40.00	
2	conv_a	40.00	0.00	
3	coke	0.00	100.00	
3	silo_a	100.00	40.00	3.06
3	feeder_a	60.00	60.00	
3	conv_a	60.00	0.00	
4	coke	0.00	100.00	
4	silo_a	100.00	60.00	3.57
4	feeder_a	40.00	40.00	
4	conv_a	40.00	0.00	
5	coke	0.00	100.00	
5	silo_a	100.00	40.00	4.33
5	feeder_a	60.00	60.00	
5	conv_a	60.00	0.00	
6	coke	0.00	100.00	
6	silo_a	100.00	60.00	4.84
6	feeder_a	40.00	40.00	
6	conv_a	40.00	0.00	

7	coke	0.00	100.00	
7	silo_a	100.00	40.00	5.61
7	feeder_a	60.00	60.00	
7	conv_a	60.00	0.00	
8	coke	0.00	100.00	
8	silo_a	100.00	60.00	6.11
8	feeder_a	40.00	40.00	
8	conv_a	40.00	0.00	
9	coke	0.00	100.00	
9	silo_a	100.00	40.00	6.88
9	feeder_a	60.00	60.00	
9	conv_a	60.00	0.00	
10	coke	0.00	100.00	
10	silo_a	100.00	60.00	7.39
10	feeder_a	40.00	40.00	
10	conv_a	40.00	20.00	

Figure 8.10 Running the Simulation

8.6 Implementation of Persistent Objects in C++ - Conclusions

This chapter described various techniques which can be used to add object persistence in the programming language C++.

If the objects to be saved contain only simple data types, then this data can simply be saved by using the language standard serialisation facility. If, however, the objects contain pointers or other objects, then more complex techniques must be used.

The Object Identifier (OID) approach, which is presented in this chapter, can be used to save objects that contain pointers. Two examples are given in this chapter which demonstrate the OID technique.

The examples help to demonstrate that object persistence can be obtained by extending the C++ language using classes. These techniques are, however, only applicable for the use of programmers and not for the general database user. In contrast to the relational DBMS, no facilities are available for the average user to interrogate the database using declarative query languages.

9. CONCLUSIONS

Object Oriented Database Management Systems and Persistent Programming Languages, although in their infancy, will in the future play a major role in data intensive software applications.

One major weakness of ODBMS technology, is the lack of internationally accepted standards. There is evidence that this will, in the near future, be eliminated by the acceptance of the work performed by organisations such as the Object Management Group (OMG).

Object and pointer persistence can be introduced in the existing and well supported programming language C++. Although this method has the advantage of using a well known and accepted programming language, it has the disadvantage, however, that the data management support is low-level. This means that non-programmers cannot manipulate or interrogate the data directly.

Chapter 8 presented two complete examples which illustrate the support of persistent objects in C++, by developing classes that help to augment the language. These examples also show that such techniques are only applicable for the use of programmers. Relational DBMS's, however, can be interrogated by the non-programmer by means of external views and query languages.

Hence, the use of object persistence using C++ methods is applicable only to data intensive applications that require to store complex data structures, in the form of objects, and require the computational power of C++. If, in addition, users also require access to this data, then such access must be provided by means of a specially developed user interface.

10. REFERENCES

[ADLER, 95] Emerging Standards for Component Software. RM Adler. IEEE Computer, V28,N3, March 1995.

[AHMED, 92] Object Oriented Database Management Systems for Engineering: A Comparison, Ahmed, JOOP, V3 N3, June 1992.

[ATKINSON,89] Object-Oriented Database System Manifesto. Proceedings of the First International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, December 1989.

[ATWOOD, 91] Why the OMG Object Request Broker should be good news for Object Databases Atwood T, JOOP, V3 N4, July 1991.

[BERTINO, 93] Object Oriented Database Systems. Concepts & Architectures. Bertino E. Addison Wesley , 1993.

[BETZ, 94] Inter-Operable Objects. M.Betz. Dr Dobb's Journal, No220, Oct 1994.

[BLAIR, 94] Integrated Support for Complex Objects in a Distributed Multimedia Design Environment Blair G. JOOP, V5 N1, January 1994.

[BOOCH, 94] Object-Oriented Analysis and Design. 2nd Edition. G.Booch. The Benjamin/Cummings Publishing Co.,1994.

[BROWN, 91] Object Oriented Databases. AW Brown. McGrawhill Book Co., 1991.

[CATTELL, 94] The Object Database Standard: ODMG-93. Release1.1. Cattell RGB, Atwood T, Duhl J, Ferran G, Loomis M, Wade D, Morgan Kaufman Publishers, 1994.

[CHENG, 93] Distributed Object Database Management Systems. Cheng WK. JOOP, V4 N2, March 1993.

[DATE, 86] An Introduction to Database Systems. 4th Edition. CJ Date. Addison-Wesley Publishing Co.,1986.

[DEEN, 85] Principles and Practice of Database Systems. S.M.Deen. Mackmillan, 1985.

[DEUX, 90] The Story of O2. O.Deux. IEEE Transactions on Knowledge Data Engineering, Vol 2, March 1990.

[ELMASRI, 89] Fundamentals of Database Systems. 4th Edition. Elmasri R.Benjamin Cumming ,1989.

[FISHMAN, 87] IRIS: An Object Oriented Database Management System. D.Fishman, D.Beech, H.Cate. ACM Trans. Office Information Systems vol 5. January 1987.

[FORD, 88] Zeitgeist: Database Support of Object-Oriented Programming. S.Ford, J.Joseph. Proc. Second International Workshop on Object-Oriented Database Systems. 1988.

[GRAHAM, 91] Object Oriented Methods. I.Graham. Addison-Wesley. 1991.

[GUPTA, 91] Object Oriented Databases with Applications to CASE, Networks and VLSI CAD Gupta R .Prentice Hall, 1991.

[JOSEPH, 91] Object Oriented Databases: Design and Implementation. JV Joseph, SM Thatte, CW Thompson, DL Wells. Proc of IEEE, V79 N1, January 1991.

[KAPPEL, 94] TriGS: Making a Passive OODBS active. Kappel G .JOOP, V5 N4, July 1994.

[KHOSHAFIAN, 93] Object Oriented Databases. Khoshafian S. John Wiley & Sons,1993.

[KIM,90] Architectural Issues in Object Oriented Databases. Kim W. JOOP, V1 N2, March 1990.

[KIM, 93] Object Oriented Database Systems - Strengths & Weaknesses. Kim W JOOP, V4 N4, July 1993.

[LAURENT, 93] Persistence in C++ . Laurent P.JOOP, V4 N4, October 1993.

[LIPPMAN, 91] C++ Primer, 2nd Edition. SB Lippman. Addison Wesley Publishing Co.,1991.

[LOOMIS, 90.5] ODBMS - The Basics. Loomis MES. JOOP, 5/1/90, V1, N3, May 1990.

[LOOMIS, 90.7] ODBMS vs Relational. Loomis MES. JOOP, V1 N4, July 1990.

[LOOMIS, 90.9] Database Transactions. Loomis MES.JOOP, V1 N5, September 1990.

[LOOMIS, 91.1] More on Transactions. Loomis MES.JOOP , V2 N1, January 1991.

[LOOMIS, 93.2] Object Programming & Database Management. Loomis MES. JOOP, V4 N1, February 1993.

[LOOMIS, 93.3] Distibuted Object Databases. Loomis MES. JOOP, V4 N2, March 1993.

[LOOMIS, 93.5] Object Programming & Database management - Differences in Perspective. Loomis MES. JOOP, V4 N2, May 1993.

[LOOMIS, 93.6] The ODMG Object Model Loomis MES. JOOP, V4 N3, June 1993.

[LOOMIS, 93.7] Object Database Semantics Loomis MES.JOOP, V4 N3, July 1993.

[LOOMIS,93.10] Making Objects Persistent Loomis MES.JOOP, V4 N4, October 1993.

[LOOMIS, 94.1] Hitting the Relational Wall Loomis MES.JOOP, V5 N1, Janauary 1994.

[LOOMIS, 94.7] ODBMS - Myths and Realities Loomis MES.JOOP, V5 N4, July 1994.

[MILLER, 91] The Active KDL ODBMS & its Application to Simulation Support. Miller JOOP, V3 N4, July 1991.

[OMG-93] A Component Technology for the Future - An OMG White Paper, OMG Headquarters, 1993.

[PARSAYE, 89] Intelligent Databases, Object Oriented, Deductive Hypermedia Technologies Parsaye K .John Wiley & Sons, 1989.

[PATON, 96] Database Programming Languages. N.Paton, R.Cooper, H.Williams, P.Trinder. Prentice Hall, 1996.

[RUMBAUGH, 91] Object-Oriented Modeling and Design. J.Rumbaugh, Prentice-Hall. 1991.

[SCHAFFERT,86] An Introduction to Trellis/OWL. C.Schaffert,T.Cooper, B.Bullies. OOPSLA '86 Conference Proc., 1986.

[SEBESTA, 93] Concepts of Programming Languages. 2nd Edition. RW Sebesta. The Benjamin/Cummings Publishing Co. 1993.

[SHILLING, 94] How to roll your own Persistent Objects in C++ Shilling JJ. JOOP, V5, N4, July 1994.

[SOMMERVILLE, 89] Software Engineering, 3rd Edition. I.Sommerville. Addison-Wesley Publishing Company,1989.

[STEVENS,94] C++ Database Development . 2nd Edition. Stevens AL .MIS Press, 1994.

[STICKLAND, 94] The guts of an Object Database Strickland H C++ Report, June 1994.

[STONEBRAKER,90] The Implementation of Postgres. M.Stonebraker, L.Rowe, M.Hirohama. IEEE Transactions on Knowledge Data Engineering, Vol 2, March 1990.

[ULLMAN, 88] Principles of Database and Knowledge Base Systems - Vol 1 Ullman Computer Science Press, 1988.

[VADAPARTY, 95] Persistent Pointers 1. Kumar Vadaparty. JOOP, V6 N4, July-August 1995.

[WILCOX, 94] Object Databases Wilcox J. Dr Dobbs Journal, November 1994.

[ZDONIK, 90] Readings in ODBMS Zdonik , Morgan Kaufman Pub., 1990.

APPENDIX

APPENDIX A - THE EMPLOYEE-DEPARTMENT APPLICATION SOURCE CODE

```
/*********************************************************************
*  file     :  persist1.h
*
*  contains :  persist class
*           :  dbase   class
*
*  purpose  :  This file must be included in the applications
*           :  that require persistant objects.
*
*           :  All persistant classes are derived from the 'persist'
*           :  class.
*
*           :  The 'dbase' class provides additional database functionality
*           :  ie read,write, insert get objects from disk.
*
*  rev      :  1=>(20/9/95) => created separate .h file
*           :  2=>(25/9/95) => added macro definitions
*           :  3=>(1/11/95) => implemented OID's as an alternative to
*           :                  handling persistent pointers
*********************************************************************/
#include <iostream.h>
#include <fstream.h>
#include <string.h>

#define MAXCLASSNAME   30
#define MAXOBJECTS     30
typedef int            OID;

/*******************************************************************
*                 MACRO DEFINITIONS
*
*  macro :  PERSIST_OP(X)
*
*  use   :  macro to define persistent operators
*        :  used in derived persistent classes
*
*  X     :  is replaced with the derived class
*
*  macro :  PERSIST_FRIEND(X)
*
*  use   :  to define the persistent operators as friends to
*        :  the derived persist classes.
*
*  X     :  is replaced with the derived class
.....................................................................*/

#define PERSIST_OP(X) opstream& operator<<(opstream& os, X *c) \
                      { return os << (persist*)c;}; \
                      ipstream& operator>>(ipstream& is, X *&c) \
                      { return is >> (persist*)c;}

#define PERSIST_FRIEND(X)  friend opstream& operator <<(opstream&, X * );\
                           friend ipstream& operator >>(ipstream&, X *&)
//-------------------------------------------------------------------
class ipstream: public ifstream {}; // persistent streams
class opstream: public ofstream {}; // inherit form iosteams
class dbase;
```

```
/*****************************************************************
*  class : persist
*
*        : provides persistance to objects that inherit from it
*
*****************************************************************/
class persist
{
   // declare the persistant i/o operators as friends
   // such that they can have access to private data members
   // of the class
   friend ipstream& operator >>(ipstream&, persist* &);
   friend opstream& operator <<(opstream&, persist*  );
   friend dbase; // class dbase can set the objects oid

 public:
   OID          getOid(){return oid;}  //returns OID of object
   virtual char *className()     =0;   //returns class name

 protected:
   // the following virtual functions must be defined for all
   // derived persistant objects.
   virtual void write(opstream&) =0;  //write object to disk
   virtual void read(ipstream&)  =0;  //read  object from disk

 private:
   OID   oid;                         //holds the OID for all
                                      //persistent objects
};
/*************************************************************
*  function :  create(char *s)
*
*  purpose  :  function must be defined by the application
*           :  Its purpose is to create objects according to the
*           :  class name 'className' which is passed as a string.
.............................................................*/
persist *create(char *className);

/*************************************************************
* Persitent Operators
*
* <<  this operator will first write the class name to disk
*     and then the object.
*     To write the object, it calls the polymorpic function
*     'write(os)' which takes as a parameter a reference
*     to the output persistent stream.
*
* >>  This operator will read an object from the persistent stream
*     It first reads the class name
*     then it creates an object of that class
*     then it reads and initialises the data members of that object.
*
*************************************************************/
opstream& operator<<(opstream& os, persist *psObj)
{
   os << psObj->className();  //first write name of class
   os << ' ' << psObj->oid;   //then write OID
   psObj->write(os);          //then write data members
   return os;
};
ipstream& operator>>(ipstream& is, persist *&psObj)
{
   char  className[MAXCLASSNAME];
   OID   oidRead;

   is >> className;              //read class name from disk
   is.get(); is >> oidRead;      // read objects oid
   is.get();
   psObj = create(className);    //create object
   psObj->oid = oidRead;         //set objects oid
   psObj->read(is);              //read data members of object from disk
```

```
   return is;
};

/*******************************************************************
*  class : dbase
*
*        : This class is responsible for the general handling of
*        : the database.
*
*        : It contains member functions to open, read, write, and close
*        : the database.
*
*        : The database contains an array of pointers to objects,
*        : which is kept as a private data member.
*        : This array defines the objects that exist in the memory space
*        : of the running application.
*******************************************************************/
class dbase
{
  public:
            dbase(char *fname);        // constructor -> supply file name

   int      openDbase();               // opens file and reads all records
   int      saveDbase();               // save all records to disk
   int      insertObj(persist *pObj);  // inserts object in dbase
   void     deleteObj();               // deletes last Object
   void     deleteAll();               // deletes all objects
   persist* getObject(int n);          // get object n=1..last
   int      getMaxObj()
            {return maxObjects;}       // get number of objects
   persist* getPointer(OID objectOid); // get the pointer of object with
                                       //  given oid.
  private:
   char     fileName[14];         // contains dbase file name
   int      maxObjects;           // contains no. of objects
   persist  *pPObj[MAXOBJECTS];   // contains pointers to objects
                                  // that have been read from disk and
                                  // created in the memory space of the
                                  // application.
};
/*-----------------------------------------------
*  function : Constructor
*
*  params   : fName => the file name to be created
*
*  purpose  : Initialise the database
.................................................*/
dbase::dbase(char *fName)   // constructor
{
   strcpy(fileName, fName);
   maxObjects=0;
};
/*-----------------------------------------------
*  function : insertObj
*
*  params   :  pObj => pointer to the object that
*                      must be stored.
*
*  pupose   :  will store this pointer to the array
            :  which holds all pointer to persistent objects
.................................................*/
int dbase::insertObj(persist *pObj)
{
   if (maxObjects < MAXOBJECTS)
   {
      maxObjects++;               // this object's oid
      pPObj[maxObjects] = pObj;   // save the object's pointer to the array
      pObj->oid = maxObjects;     // define the object's OID
      return 1;                   // 1==> OK
   }
   else
      return 0;                   // 0==> not ok
```

```
};
/*-----------------------------------------------
*  functions : dateteObj    => deletes last object
*            : deleteAll    => deletes all objects
*.............................................*/
void dbase::deleteObj()
{
   pPObj[maxObjects] = 0;
   maxObjects--;
};
void dbase::deleteAll()
{
   for (int i=0; i<=maxObjects; i++) // make all object pointers nil
      pPObj[i] = 0;
   maxObjects =0;
};
/*-------------------------------------------------------
*  function :  openDbase()
*  returns  :  1 => all ok
*           :  0 => could not open file
*
*  purpose  : opens disk file and reads all objects
*
*           : the >> operator will create the objects in
*           : the applications memory space.
*
*           : the private array of pointers to existing objects
*           : pPObj[] is also initialised accordingly.
*
*           : Finally the disk file is closed
.......................................................*/

int dbase::openDbase()
{
   ipstream inFile;
   persist  *pObj;


   inFile.open(fileName);

   maxObjects = 0;
   while(inFile.peek() != EOF)
   {
      inFile >> pObj;
      maxObjects++;
      pPObj[maxObjects] = pObj;
   };

   inFile.close();
   return 1;

};
/*-------------------------------------------------------
*  function :  getObject(int n)
*
*  params   :  n =>object number (sequence number 1->last)
*
*  returns  :  pointer to nth object read from disk
*           :  0 => could not get the object
*
*  purpose  : to get a pointer to the nth object read.
.......................................................*/

persist* dbase::getObject(int n)
{
   if (n <= maxObjects)
      return pPObj[n];
   else
      return 0;
};
```

```
/*-------------------------------------------------------
*  function :  getPointer(OID oid)
*
*  params   :  oid =>object's oid
*
*  returns  :  pointer to object of given oid
*           :  0 => could not get the object
*
*  purpose  : to get a pointer to the object with given oid.
.......................................................*/

persist* dbase::getPointer(OID oid)
{
   if (oid <= maxObjects)
      return pPObj[oid];
   else
      return 0;
};

/*-------------------------------------------------------
*  function :  saveDbase()
*  returns  :  1 => all ok
*           :  0 => could not save file
*
*  purpose  : opens disk file and write all objects
*
*           : the << operator will read the objects in
*           : the applications memory space and save them to disk.
*
*           : Finally the disk file is closed
.......................................................*/

int dbase::saveDbase()
{
   opstream outFile;

   outFile.open(fileName);

   for(int i=1; i<=maxObjects; i++)
      outFile << pPObj[i];

   outFile.close();
   return 1;
};
//=========================================================
```

```
/*****************************************************
*  File  :  empl2.cpp
*
*        :  demonstrates the application of the
*        :  'persist' class by storing and retrieving
*        :  objects from disk storage.
*
*  date  :  5/10/95  => rev 1
*        :  1/11/95  => added support for OID's
*        :  1/11/95  => added the department class
*****************************************************/

#include <strstrea.h>
#include <stdlib.h>
#include <iomanip.h>
#include <typeinfo.h>

#include "persist1.h"    // contains the persistent class definition

#define  MAXNAME   30   // maximum employee name length
#define  MAXDEP    30   // maximum depertment name length

// ----- global variables ----------------------------
dbase emplDbase("emp.dat");  // create database object

class department;            // forward declarations
class employee;

/*****************************************************************
*  class :  department
*        :  contains all data relevent to the department:
*        :     name, telephone number
.................................................................*/
class department: public persist
{
   PERSIST_FRIEND(department);  // macro defined in file 'persist.h'

 public:
   // public function to gain access to the object's private data
   char  *getName()            {return name;}        //returns dep name
   int   getTelNo()            {return telNo;}       //returns tel number
   void  setName(char *pName)  {strcpy(name, pName);}
   void  setTelNo(int iTelNo)  {telNo = iTelNo;}

 protected:
   // definition of virtual function that provide object persistance
   void  write(opstream&);
   void  read(ipstream&);
   char  *className();

 private:
   // private data (attributes) stored per object
   char  name[MAXNAME];        //holds the department name
   int   telNo;                //holds the telephone number
};
PERSIST_OP(department); // macro defined in 'persist.h'
//-------------------------------------------------------------------
void department::write(opstream& os)
{
   // the 'write' virtual function defines the method of storing
   // this object to disk
   os <<' ' << name
      <<' ' << telNo
      <<endl;
};
void department::read(ipstream& is)
{
   // the 'read' virtual function defines the method of reading
   // the object. This method has to correspond to the way the object
   // has been serialised and stored on disk.
   is.get(name, MAXNAME, ' ');      is.get();
   is >> telNo;                     is.get();
```

```
};
char* department::className()
{
   // the virtual function reurns the class name
   return "department";
};
//-----------------------------------------------------------------------
/*****************************************************************
*  class :  employee
*        :  contains all data relevent to the employee:
*        :     name, depertment & salary
.................................................................*/
class employee: public persist
{
   PERSIST_FRIEND(employee);  // macro defined in file 'persist.h'

 public:
   // public function to gain access to the object's private data
   char* getName()                {return name;}     //name
   OID   getDepOid()              {return depOid;}   //depart oid
   char* getDepartment();                            //depart name
   float getSalary()              {return salary;}   //empl salary
   void  setName(char *pName)     {strcpy(name, pName);}
   void  setDepOid(OID oid)       {depOid = oid;}
   void  setSalary(float fSalary){salary = fSalary;}

 protected:
   // definition of virtual function that provide object persistance
   void  write(opstream&);
   void  read(ipstream&);
   char  *className();

 private:
   // private data (attributes) stored per object
   char  name[MAXNAME];     //holds the employee name
   OID   depOid ;           //holds the department OID
   float salary;            //holds the salary
};
PERSIST_OP(employee); // macro defined in 'persist.h'
//-----------------------------------------------------------------------

char* employee::getDepartment()
{
   // function to return the department name as a string
   //   the function first gets a pointer from the dbase object
   //   using the OID that it contains

   persist*    pPObj;
   department* pDepObj;

   pPObj   = emplDbase.getPointer(depOid); // get pointer to persist object
   pDepObj = dynamic_cast<department*> (pPObj); //cast to department

   return pDepObj->getName();              //return the dap. name

};
//-----------------------------------------------------------------------
void employee::write(opstream& os)
{
   // the 'write' virtual function defines the method of storing
   // this object to disk
   os <<' ' << name
      <<' ' << depOid
      <<' ' << salary
      <<endl;
};
void employee::read(ipstream& is)
{
   // the 'read' virtual function defines the method of reading
   // the object. This method has to correspond to the way the object
   // has been serialised and stored on disk.
   is.get(name, MAXNAME, ' ');      is.get();
```

```
   is >> depOid;                    is.get();
   is >> salary;                    is.get();
};
char* employee::className()
{
   // the virtual function reurns the class name
   return "employee";
};
//-----------------------------------------------------------------------

/*********************************************************************
*  function :  create
*     in    :  char *s: s is a class name
*     return:  pointer to the dynamic object created
*           :  the pointer return is of type 'persist'
*           :
*           :  creates an object dynamically according to the supplied
*           :     class name
.....................................................................*/
persist *create(char *s)
{
   if (strcmp(s, "employee") == 0)
      return new employee;
   else if (strcmp(s, "department") == 0)
      return new department;
   else
      return 0;
};
//-------------------------------------------------------------------

//*********************************************************************
void depMenu()
{
   // menu to obtain the department details and

   char  pName[MAXNAME];
   int   telNo;
   char  ch='y';

   while (ch == 'y')
   {
      // get data from user
      cout <<endl<< "enter department's name: "; cin >> pName;
      cout <<endl<< "enter telephone no: "     ; cin >> telNo;

      department *pdepartment = new department; // create new object
      pdepartment->setName(pName);              // and set its private data
      pdepartment->setTelNo(telNo);

      emplDbase.insertObj(pdepartment);    // insert this object in the
                                           // database

      cout <<endl<< " More departments? <y/n>: "; cin >> ch;
   };

};
//-------------------------------------------------------------------
void emplMenu()
{
   // Menu to obtain the employee data
   // The function also checks for referntial integrity of the
   // department entered. If Department does not exist, the data is
discarded

   char        pName[MAXNAME];
   char        pDepName[MAXDEP];
   char        pDepTemp[MAXDEP];
   int         fSalary;
   OID         depOid;
   char        ch = 'y';
   persist*    pPObj;
   employee*   pEmpl;
```

```
    department* pDep;

     while (ch == 'y')
    {
       // get data from user
       cout <<endl<< "enter employee's name: "; cin >> pName;
       cout <<endl<< "enter department : "    ; cin >> pDepName;
       cout <<endl<< "enter salary: "         ; cin >> fSalary;

       // search the dapartments for exist dep. name
       depOid = 0;
       int n=1;

       while(pPObj = emplDbase.getObject(n))
       {
          if (strcmp(pPObj->className(),"department") == 0)
          {  //if pointing to a department object
             pDep = dynamic_cast<department*> (pPObj);

             strcpy(pDepTemp, pDep->getName());
             if(strcmp(pDepName,pDepTemp)== 0) // if department exists
             {
                 depOid = pDep->getOid(); // get department's oid
                 break;
             };
          };
          n++;
       };
       if (depOid != 0)
       {
          employee *pEmployee = new employee; // create new object
          pEmployee->setName(pName);          // and set its private data
          pEmployee->setDepOid(depOid);
          pEmployee->setSalary(fSalary);

          emplDbase.insertObj(pEmployee);     // insert this object in the
        }                                     // database
        else
          cout<< "Department does not exist!" <<endl;

       cout <<endl<< " More Employees? <y/n>: "; cin >> ch;
    };

};
//-----------------------------------------------------------------
void printEmployees()
{
   //Print the employees data obtained from the database

   persist  *pPObj; // pointer to persist objects
   employee *pEmpl; // pointer to employee objects
   int      n=1;

   cout << endl  <<setw(20) << "EMPLOYEES" << endl;
   cout  << setw(10) << "Name"
         << setw(10) << "OID"
         << setw(10) << "Depmt"
         << setw(10) << "Salary"
         << endl;
```

```
   while(pPObj = emplDbase.getObject(n))  // read object 'n'
   {
      if (strcmp(pPObj->className(),"employee") == 0)
      {  //if pointing to an employee object
         pEmpl = dynamic_cast<employee*> (pPObj);

         // print data read from disk
         cout  <<setw(10)<< pEmpl->getName()
               <<setw(10)<< pEmpl->getDepOid()
               <<setw(10)<< pEmpl->getDepartment()
               <<setw(10)<< pEmpl->getSalary()
               << endl;
      };
      n++;
   };
};
//-----------------------------------------------------------------

void printDepartments()
{
   // Print the department's data from the database

   persist     *pPObj; // pointer to persist objects
   department  *pDep;  // pointer to department objects
   int         n=1;

   cout  << endl <<setw(20) << "DEPARTMENTS" << endl;
   cout  << setw(10) << "Name"
         << setw(10) << "OID"
         << setw(10) << "TelNo"
         << endl;

   while(pPObj = emplDbase.getObject(n))  // read object 'n'
   {
      if (strcmp(pPObj->className(),"department") == 0)
      {  //if pointing to an department object
         pDep = dynamic_cast<department*> (pPObj);

         // print data read from disk
         cout  << setw(10)<< pDep->getName()
               << setw(10)<< pDep->getOid()
               << setw(10)<< pDep->getTelNo()
               << endl;
      };
      n++;
   };
};
//-----------------------------------------------------------------


void mainMenu()
{
   // This is the main menu
   int ans=0;

   while(ans != 10)
   {
      cout <<endl <<setw(15) <<"MAIN MENU" <<endl;

      cout << "1. Enter Departments" << endl;
      cout << "2. Enter Employees"   << endl;
      cout << "3. Save Database"     << endl;
      cout << "4. Read Database"     << endl;
      cout << "5. Print Employees"   << endl;
      cout << "6. Print Department"  << endl;endl;
      cout << "10. Exit"             << endl;

      cin >> ans;
```

```
      switch (ans)
      {
         case 1:
            depMenu();
            break;
         case 2:
            emplMenu();
            break;
         case 3: //save database
            emplDbase.saveDbase(); // save data to disc and close file
            break;
         case 4: //read database
            emplDbase.deleteAll(); // delete database from RAM.
            emplDbase.openDbase(); // open database & read all objects from
                                   // disk
            break;
         case 5:
            printEmployees();
            break;
         case 6:
            printDepartments();
            break;
         default:;
      };
   };
};
//-------------------------------------------------


/*****************************************************************
*                        MAIN PROGRAM
*
*  This program perform the following tasks:
*
*     - get employee's and department's data from user
*     - create an object per employee and department and
*        initialise the objects private data
*     - save these objects to disk
*     - detete all data from memory space
*     - open database and read all objects
*     - display all objects private data
*
*  Relationships between employee and department are represented by
pointers
.................................................................*/

void main()
{
   char  pName[MAXNAME];
   char  pDep[MAXDEP];
   float fSalary;

   mainMenu();

};
//*****************************************************************
```

APPENDIX B - THE REAL-TIME SIMULATOR APPLICATION SOURCE CODE

```
/*****************************************************************
*  file     :  persist.h
*
*  contains :  persist class
*           :  dbase   class
*
*  purpose  :  This file must be included in the applications
*           :  that require persistant objects.
*
*           :  All persistant classes are derived from the 'persist'
*           :  class.
*
*           :  The 'dbase' class provides additional database functionality
*           :  ie read,write, insert get objects from disk.
*
*  rev      :  1=>(20/9/95) => created separate .h file
*           :  2=>(25/9/95) => added macro definitions
*****************************************************************/
#include <iostream.h>
#include <fstream.h>
#include <string.h>

#define MAXCLASSNAME   30
#define MAXOBJECTS     30

/*****************************************************************
*                MACRO DEFINITIONS
*
*  macro :  PERSIST_OP(X)
*
*  use   :  macro to define persistent operators
*        :  used in derived persistent classes
*
*  X     :  is replaced with the derived class
*
*  macro :  PERSIST_FRIEND(X)
*
*  use   :  to define the persistent operators as friends to
*        :  the derived persist classes.
*
*  X     :  is replaced with the derived class
.................................................................*/

#define PERSIST_OP(X) opstream& operator<<(opstream& os, X *c) \
                      { return os << (persist*)c;}; \
                      ipstream& operator>>(ipstream& is, X *&c) \
                      { return is >> (persist*)c;}

#define PERSIST_FRIEND(X)  friend opstream& operator <<(opstream&, X * );\
                           friend ipstream& operator >>(ipstream&, X *&)
//-----------------------------------------------------------------
class ipstream: public ifstream {}; // persistent streams
class opstream: public ofstream {}; // inherit form iosteams

/*****************************************************************
*  class : persist
*
*        : provides persistance to objects that inherit from it
*
*****************************************************************/
class persist
{
   // declare the persistant i/o operators as friends
   // such that they can have access to private data members
   // of the class
   friend ipstream& operator >>(ipstream&, persist* &);
```

```
   friend opstream& operator <<(opstream&, persist*  );

 protected:
   // the following virtual functions must be defined for all
   // derived persistant objects.
   virtual void write(opstream&) =0;  //write object to disk
   virtual void read(ipstream&)  =0;  //read  object from disk
   virtual char *className()     =0;  //returns class name
};
/*************************************************************
*  function :  create(char *s)
*
*  purpose  :  function must be defined by the application
*           :  Its purpose is to create objects according to the
*           :  class name 'className' which is passed as a string.
.............................................................*/
persist *create(char *className);

/*************************************************************
* Persitent Operators
*
* <<  this operator will first write the class name to disk
*     and then the object.
*     To write the object, it calls the polymorpic function
*     'write(os)' which takes as a parameter a reference
*     to the output persistent stream.
*
* >>  This operator will read an object from the persistent stream
*     It first reads the class name
*     then it creates an object of that class
*     then it reads and initialises the data members of that object.
*
*************************************************************/
opstream& operator<<(opstream& os, persist *psObj)
{
   os << psObj->className();  //first write name of class
   psObj->write(os);          //then write data members
   return os;
};
ipstream& operator>>(ipstream& is, persist *&psObj)
{
   char className[MAXCLASSNAME];
   is >> className;              //read class name from disk
   is.get();
   psObj = create(className);    //create object
   psObj->read(is);              //read object from disk
   return is;
};

/*******************************************************************
*  class : dbase
*
*        : This class is responsible for the general handling of
*        : the database.
*
*        : It contains member functions to open, read, write, and close
*        : the database.
*
*        : The database contains an array of pointers to objects,
*        : which is kept as a private data member.
*        : This array defines the objects that exist in the memory space
*        : of the running application.
*******************************************************************/
class dbase
{
  public:
            dbase(char *fname);        // constructor -> supply file name

   int      openDbase();               // opens file and reads all records
   int      saveDbase();               // save all records to disk
   int      insertObj(persist *pObj);  // inserts object in dbase
   void     deleteObj();               // deletes last Object
```

```
   void     deleteAll();                // deletes all objects
   persist  *getObject(int n);          // get object n=1..last
   int      getMaxObj()
            {return maxObjects;}        // get number of objects

  private:
   char     fileName[14];          // contains dbase file name
   int      maxObjects;            // contains no. of objects
   persist  *pPObj[MAXOBJECTS];    // contains pointers to objects
                                   // that have been read from disk and
                                   // created in the memory space of the
                                   // application.
};
/*-----------------------------------------------
*  function : Constructor
*
*  params   : fName => the file name to be created
*
*  purpose  : Initialise the database
.................................................*/
dbase::dbase(char *fName)   // constructor
{
   strcpy(fileName, fName);
   maxObjects=0;
};
/*-----------------------------------------------
*  function : insertObj
*
*  params   :  pObj => pointer to the object that
*                      must be stored.
*
*  pupose   :  will store this pointer to the array
            :  which holds all pointer to persistent objects
.................................................*/
int dbase::insertObj(persist *pObj)
{
   pPObj[maxObjects] = pObj;
   maxObjects++;
   return 1;
};
/*-----------------------------------------------
*  functions : dateteObj   => deletes last object
*            : deleteAll   => deletes all objects
*...............................................*/
void dbase::deleteObj()
{
   maxObjects--;
};
void dbase::deleteAll()
{
   maxObjects =0;
};
/*-----------------------------------------------
*  function :  openDbase()
*  returns  :  1 => all ok
*           :  0 => could not open file
*
*  purpose  : opens disk file and reads all objects
*
*           : the >> operator will create the objects in
*           : the applications memory space.
*
*           : the private array of pointers to existing objects
*           : pPObj[] is also initialised accordingly.
*
*           : Finally the disk file is closed
.................................................*/

int dbase::openDbase()
{
   ipstream inFile;
   persist  *pObj;
```

```
   inFile.open(fileName);

   maxObjects = 0;
   while(inFile.peek() != EOF)
   {
      inFile >> pObj;
      pPObj[maxObjects] = pObj;
      maxObjects++;
   };

   inFile.close();
   return 1;

};
/*-----------------------------------------------
*  function :  getObject(int n)
*
*  params   :  n =>object number (sequence number 1->last)
*
*  returns  :  pointer to nth object read from disk
*           :  0 => could not get the object
*
*  purpose  : to get a pointer to the nth object read.
.......................................................*/

persist* dbase::getObject(int n)
{
   if (n <= maxObjects)
      return pPObj[n-1];
   else
      return 0;
};
/*-----------------------------------------------
*  function :  saveDbase()
*  returns  :  1 => all ok
*           :  0 => could not save file
*
*  purpose  : opens disk file and write all objects
*
*           : the << operator will read the objects in
*           : the applications memory space and save them to disk.
*
*           : Finally the disk file is closed
.......................................................*/

int dbase::saveDbase()
{
   opstream outFile;

   outFile.open(fileName);

   for(int i=0; i<maxObjects; i++)
      outFile << pPObj[i];

   outFile.close();
   return 1;
};
//=========================================================
```

```
/*************************************************
*  file: simio4.cpp
*      : dynamic simulation of silo and feeder
*      : added user interface to enter simulation params
*      : added object persistance
*      : added persist class
*      : added dbase class
*      : added print file generation
*      : added macro processing
*************************************************/
#include <strstrea.h>
#include <stdlib.h>

#include <iomanip.h>
#include <typeinfo.h>
#include "persist.h"

#define MAXCONVEYORLEN 30
#define MAXNAMELEN     30

class simObject: public persist
{
   PERSIST_FRIEND(simObject);

 public:
   virtual float getFin()                  = 0;
   virtual float getFout()                 = 0;
   virtual void  cycle()                   = 0;
   virtual void  setPin(simObject *pIn)    = 0;
   virtual void  setPout(simObject *pOut)  = 0;
   virtual void  displayDyn(ostream&, int time)     = 0; //displays dynamic
vars
   virtual void  output(ostream&)          = 0;

   void setName(char *pBuf){strcpy(pName, pBuf);}
   void getName(char *pBuf){strcpy(pBuf, pName);}

 protected:
   virtual void write(opstream&) {;}  //write to disk
   virtual void read(ipstream&)  {;}  //read from disk
   virtual char *className()     {;}  //returns class name

   char      pName[MAXNAMELEN];   // name of object
   float     fin;
   float     fout;
   simObject *pObjIn;
   simObject *pObjOut;
};
//-------------------------------------------------------------

PERSIST_OP(simObject);

class silo :public simObject
{
   PERSIST_FRIEND(silo);
 public:
         silo(){fin=0; fout=0; level=0;}
   float getLevel()            {return(level);}
   void  setInitLevel(float l) {level =l;}
   void  setDiameter(float d)  {diameter = d;}
   void  setHeight(float h)    {height = h;}
   float getDiameter()         {return(diameter);}
   int   isOverflowing()       {return(overflow);}

   void  setPin(simObject *pIn)   {pObjIn  = pIn; }
   void  setPout(simObject *pOut) {pObjOut = pOut;}

   float getFin()                 {return(fin);}
   float getFout()                {return(fout);}
   void  cycle();
   virtual void displayDyn(ostream& os, int time);
```

```
  virtual void output(ostream& os)
  {
    os  << setprecision(2)
        << setfill('.')
        << setw(10)  << setiosflags(ios::left)     << "name  : "
        << setw(10)  << resetiosflags(ios::left)   << pName       <<endl
        << setw(10)  << setiosflags(ios::left)     << "level : "
        << setw(10)  << resetiosflags(ios::left)   << level       <<endl
        << setw(10)  << setiosflags(ios::left)     << "diamt : "
        << setw(10)  << resetiosflags(ios::left)   << diameter    <<endl
        << setw(10)  << setiosflags(ios::left)     << "height: "
        << setw(10)  << resetiosflags(ios::left)   << height      <<endl;
  };

 protected:
  virtual void write(opstream& os)
  {
     os <<' ' << pName
        <<' ' << level
        <<' ' << diameter
        <<' ' << height   << endl;
  };
  virtual void read(ipstream& is)
  {
     is.get(pName, MAXNAMELEN, ' ');
     is.get(); is >> level;
     is.get(); is >> diameter;
     is.get(); is >> height;
     is.get();
  };
  virtual char* className(){return "silo";}

 private:
  float level;
  float diameter;
  float height;
  int   overflow;
};
void silo::displayDyn(ostream& os, int time)
{
  os                    // << resetiosflags(ios::left)
     << setw(5) << time
     << setw(10) << pName
     << setprecision(2)
     << setw(10) << fin
     << setw(10) << fout
     << setw(10) << level
     << endl;
};
void silo::cycle()
{
  if (pObjIn)
     fin  = pObjIn->getFout();
  else
     fin = 0;
  if (pObjOut)
     fout = pObjOut->getFin();
  else
     fout = 0;
  //cout << "fin: " << fin << " fout: " << fout << endl;

  level = (fin-fout)*4.0/(3.14*diameter*diameter) + level;

  // check for overflowing silo
  if (level > height)
  {
     level = height;
     overflow = 1;
  }
  else
     overflow = 0;
}
```

```
PERSIST_OP(silo);

/*********************************************************
*  class: feeder
.........................................................*/

class feeder : public simObject
{
  PERSIST_FRIEND(feeder);
 public:
        feeder(){fin=0; fout=0;}
   void  setMaxFlow(float f)     {maxFlow = f;}
   void  setSlope(float s)       {slope   = s;}
   void  setInitFlow(float f)    {fout = f; fin=f;}

   void  setPin (simObject *pIn)  {pObjIn  = pIn; }
   void  setPout(simObject *pOut) {pObjOut = pOut;}
   void  setSetpoint(float sp)    {setpoint = sp;}

   float getFin()               {return(fin);}
   float getFout()              {return(fout);}
   void  cycle();
   virtual void displayDyn(ostream& os, int time);
   virtual void output(ostream& os)
   {
     os  << setprecision(2)
         << setfill('.')
         << setw(10)  << setiosflags(ios::left)     << "name  : "
         << setw(10)  << resetiosflags(ios::left)   << pName         <<endl
         << setw(10)  << setiosflags(ios::left)     << "maxFlow : "
         << setw(10)  << resetiosflags(ios::left)   << maxFlow       <<endl
         << setw(10)  << setiosflags(ios::left)     << "slope: "
         << setw(10)  << resetiosflags(ios::left)   << slope         <<endl
         << setw(10)  << setiosflags(ios::left)     << "setpoint: "
         << setw(10)  << resetiosflags(ios::left)   << setpoint
<<endl;

   };

 protected:
   virtual void write(opstream& os)
   {
      os <<' ' << pName
         <<' ' << maxFlow
         <<' ' << slope
         <<' ' << setpoint   << endl;
   };
   virtual void read(ipstream& is)
   {
      is.get(pName, MAXNAMELEN, ' ');
      is.get(); is >> maxFlow;
      is.get(); is >> slope;
      is.get(); is >> setpoint;
      is.get();
   };
   virtual char* className(){return "feeder";}


 private:
   float maxFlow;
   float slope;
   float setpoint;
};
void feeder::displayDyn(ostream& os, int time)
{
   os << resetiosflags(ios::left)
      << setw(5) << time
      << setw(10) << pName
      << setprecision(2)
      << setw(10) << fin
      << setw(10) << fout
      << endl;
```

```
};
void feeder::cycle()
{
   float desiredFlow;

   desiredFlow = setpoint * maxFlow;

   if (fout < desiredFlow)
      fout = fout + slope;
   else if (fout > desiredFlow)
      fout = fout - slope;

   if (fout > maxFlow)     //check for max/min values
      fout = maxFlow;
   if (fout < 0)
      fout = 0;

   fin = fout;

};
PERSIST_OP(feeder);

//-----------------------------------------------
class conveyor : public simObject
{

   PERSIST_FRIEND(conveyor);

 public:
   conveyor(float s, float l) // constructor
   {
      if(s!=0)
         delay  = (int)l/s; // set private vars
      else
         delay = 0;

      speed  = s;
      length = l;
      fin    = 0; fout = 0;

      // clear shift register
      for(int i=delay; i>= 1; i--)
         shiftReg[i] = 0;
   }

   float  getSpeed()       {return(speed);}
   float  getLength()      {return(length);}
   int    getDelay()       {return(delay);}

   void  setPin (simObject *pIn)  {pObjIn  = pIn; }
   void  setPout(simObject *pOut) {pObjOut = pOut;}

   float getFin()              {return(fin);}
   float getFout()             {return(fout);}
   void  cycle();
   virtual void displayDyn(ostream& os, int time);
   virtual void output(ostream& os)
   {
     os  << setprecision(2)
         << setfill('.')
         << setw(10)  << setiosflags(ios::left)     << "name  : "
         << setw(10)  << resetiosflags(ios::left)   << pName        <<endl
         << setw(10)  << setiosflags(ios::left)     << "speed : "
         << setw(10)  << resetiosflags(ios::left)   << speed        <<endl
         << setw(10)  << setiosflags(ios::left)     << "delay: "
         << setw(10)  << resetiosflags(ios::left)   << delay        <<endl
         << setw(10)  << setiosflags(ios::left)     << "length: "
         << setw(10)  << resetiosflags(ios::left)   << length       <<endl;

   };

 protected:
```

```
   virtual void write(opstream& os)
   {
      os <<' ' << pName
         <<' ' << speed
         <<' ' << delay
         <<' ' << length   << endl;
   };
   virtual void read(ipstream& is)
   {
      is.get(pName, MAXNAMELEN, ' ');
      is.get(); is >> speed;
      is.get(); is >> delay;
      is.get(); is >> length;
      is.get();
   };
   virtual char* className(){return "conveyor";}

 private:

   float speed;
   int   delay;
   float length;
   float shiftReg[MAXCONVEYORLEN];
};
void conveyor::displayDyn(ostream& os, int time)
{
   os << resetiosflags(ios::left)
      << setw(5) << time
      << setw(10) << pName
      << setprecision(2)
      << setw(10) << fin
      << setw(10) << fout
      << endl;
};
void conveyor::cycle()
{
   // get input flow
   if (pObjIn)
      fin  = pObjIn->getFout();
   else
      fin = 0;

   shiftReg[0] = fin;
   for(int n=delay; n >= 1; n--)      // shift register
      shiftReg[n] = shiftReg[n-1];

   fout = shiftReg[delay];

};
PERSIST_OP(conveyor);

class source : public simObject
{
  PERSIST_FRIEND(source);
 public:
         source(){fin=0; fout=0;}
   float getFin()  (return(0.0);}
   float getFout() {return(fout);}
   void  cycle(){;} // do nothing

   void  setPin (simObject *pIn)  (pObjIn  = pIn; }
   void  setPout(simObject *pOut) (pObjOut = pOut;}
   void  setFeedrate(float fr)    {fout = fr;}
   virtual void displayDyn(ostream& os, int time);
   virtual void output(ostream& os)
   {
     os  << setprecision(2)
         << setfill('.')
         << setw(10)  << setiosflags(ios::left)     << "name  : "
         << setw(10)  << resetiosflags(ios::left)   << pName        <<endl
         << setw(10)  << setiosflags(ios::left)     << "feedrate: "
         << setw(10)  << resetiosflags(ios::left)   << fout         <<endl;
```

```
   };

 protected:
   virtual void write(opstream& os)
   {
      os <<' ' << pName
         <<' ' << fout <<endl;
   };
   virtual void read(ipstream& is)
   {
      is.get(pName, MAXNAMELEN, ' ');
      is.get(); is >> fout;
      is.get();
   };
   virtual char* className(){return "source";}
};
PERSIST_OP(source);


void source::displayDyn(ostream& os, int time)
{
   os << resetiosflags(ios::left)
      << setw(5) << time
      << setw(10) << pName
      << setw(10) << fin
      << setw(10) << fout
      << endl;
};
//-----------------------------------------------------
class sink : public simObject
{
  PERSIST_FRIEND(sink);

 public:
         sink(){fin=0; fout=0;}
   float getFin()  {return(0.0);}
   float getFout() {return(0.0);}
   void  cycle()   {;} // do nothing

   void  setPin (simObject *pIn)  (pObjIn  = pIn; }
   void  setPout(simObject *pOut) {pObjOut = pOut;}
  virtual void displayDyn(ostream& os, int time);
  virtual void output(ostream& os)
   {
      os  << "name: " << pName     << endl;
   };

 protected:
   virtual void write(opstream& os)
   {
      os <<' ' << pName << endl;

   };
   virtual void read(ipstream& is)
   {
      is.get(pName, MAXNAMELEN, ' ');
      is.get();
   };
   virtual char* className(){return "sink";}

};
PERSIST_OP(sink);

void sink::displayDyn(ostream& os, int time)
{
   os << resetiosflags(ios::left)
      << setw(5) << time
      << setw(10) << pName
      << setprecision(2)
      << setw(10) << fin
      << setw(10) << fout
```

```
      << endl;
};
//************* Simulation Global Vars ********************************

simObject   *pSimObj[MAXOBJECTS];    // array of pointers to sim objects
int         maxSimObj= 0;           // no of sim objects in simulation
dbase       simDbase("sobj.dat"); // create the database object
ofstream    prtFile("prt.txt");

void        appendMenu();
void        createNewMenu();
void        createSource();
void        createSilo();
void        createFeeder();
void        createConveyor();
void        createSink();
void        simulate();
void        saveObjects();
void        readObjects();
void        printObjects();

//---------- Set Object Pointers to point according to
//            array pSimObj[] which defines relationships

void setObjPointers()
 {
   pSimObj[0]->setPin(0);           // set first object
   pSimObj[0]->setPout(pSimObj[1]);

   for( int n = 1; n < maxSimObj-1; n++)
   {
      pSimObj[n]->setPin(pSimObj[n-1]);
      pSimObj[n]->setPout(pSimObj[n+1]);
   };
   pSimObj[maxSimObj-1]->setPin(pSimObj[maxSimObj-2]);   // set last object
   pSimObj[maxSimObj-1]->setPout(0);
 };
//-------------------------------------------------------------
void mainMenu()
{
   int ans=0;
   while (ans != 7)
   {  cout <<endl<<endl<< "************ Main Menu ****************"<<endl;
      cout << "1. Create New Simulation System"    <<endl;
      cout << "2. Append New Simulation Objects"   <<endl;
      cout << "3. Save  Simulation Objects"  <<endl;
      cout << "4. Read  Simulation Objects"  <<endl;
      cout << "5. Print Simulation Objects"  <<endl<<endl;
      cout << "6. Run Simulation"            <<endl;

      cout << "7. Quit" << endl<<endl;

      cout << "Enter Choice: ";

      cin >> ans;

      switch (ans)
      {
         case 1:
            createNewMenu();
            break;
         case 2:
            appendMenu();
            break;
         case 3:
            saveObjects();
            break;
         case 4:
            readObjects();
            break;
         case 5:
            printObjects();
```

```
            break;
         case 6:
            simulate();
            break;
         default:;
      };
   };
};
//-----------------------------------------------------
void createNewMenu()
{
   for(int i=0; i < maxSimObj; i++) // clear ram object space
      delete pSimObj[i];

   maxSimObj = 0;                   // reset object counter
   appendMenu();
}
//-----------------------------------------------------
void appendMenu()
{
   int ans=0;

   while (ans != 6)
   {
      cout << "1. Create Source"    << endl;
      cout << "2. Create Silo"      << endl;
      cout << "3. Create Feeder"    << endl;
      cout << "4. Create Conveyor"  << endl;
      cout << "5. Create Sink"      << endl;


      cout << endl << endl << "6. Quit back to Main Menu" << endl;
      cout << endl << endl << "Enter Choice: ";

      cin >> ans;

      switch (ans)
      {
         case 1:
            createSource();
            break;
         case 2:
            createSilo();
            break;
         case 3:
            createFeeder();
            break;
         case 4:
            createConveyor();
            break;
         case 5:
            createSink();
            break;

         default:;
      };
   };

};
//-----------------------------------------------------------
void createSource()
{
   char  pBuf[MAXNAMELEN];
   float feedrate;

   cout << "enter source name: ";
   cin  >> pBuf;

   cout << "enter feedrate [m3/s]: ";
   cin >> feedrate;

   source *pSourceObj = new source;
```

```
   pSourceObj->setFeedrate(feedrate);
   pSimObj[maxSimObj] = pSourceObj;

   pSourceObj->setName(pBuf);
   char pBufRead[MAXNAMELEN];
   pSourceObj->getName(pBufRead);

   cout << "name is " << pBufRead <<endl;

   maxSimObj++;
};
//-------------------------------------------------------
void createSilo()
{
   char  pBuf[MAXNAMELEN];
   float initLevel;
   float diameter;
   float height;

   cout << "enter source name: ";
   cin  >> pBuf;
   cout << "enter diameter of silo [m]: ";
   cin  >> diameter;
   cout << "enter height of silo [m]: ";
   cin  >> height;
   cout << "enter initial level of silo [m]: ";
   cin  >> initLevel;

   silo *pSiloObj = new silo;
   pSiloObj->setDiameter(diameter);
   pSiloObj->setHeight(height);
   pSiloObj->setInitLevel(initLevel);
   pSimObj[maxSimObj] = pSiloObj;
   pSiloObj->setName(pBuf);

   maxSimObj++;
};
//-------------------------------------------------------
void createFeeder()
{
   char  pBuf[MAXNAMELEN];
   float initFlow;
   float gain;
   float maxFeed;
   float setpoint;

   cout << "enter Feeder name: ";
   cin  >> pBuf;

   cout << "enter feeder acceleration [m3/s2]: ";
   cin  >> gain;
   cout << "enter maximum feedrate [m3/s]: ";
   cin  >> maxFeed;
   cout << "enter initial flowrate [m3/s]: ";
   cin  >> initFlow;
   cout << "enter feeder setpoint [0..1]: ";
   cin  >> setpoint;

   feeder *pFeederObj = new feeder;
   pFeederObj->setSlope(gain);
   pFeederObj->setMaxFlow(maxFeed);
   pFeederObj->setInitFlow(initFlow);
   pFeederObj->setSetpoint(setpoint);
   pFeederObj->setName(pBuf);

   pSimObj[maxSimObj] = pFeederObj;

   maxSimObj++;
};
//-------------------------------------------------------
void createConveyor()
{
```

```
   char  pBuf[MAXNAMELEN];

   float length;
   float speed;


   cout << "Enter Conveyor Name: ";
   cin  >> pBuf;
   cout << "Enter Conveyor Length [m]: ";
   cin  >> length;
   cout << "Enter Conveyor Speed [m/s]: ";
   cin  >> speed;

   conveyor *pConveyorObj = new conveyor(speed, length);

   pSimObj[maxSimObj] = pConveyorObj;
   pConveyorObj->setName(pBuf);

   maxSimObj++;
};
//-------------------------------------------------------------
void createSink()
{
   char  pBuf[MAXNAMELEN];

   cout << "enter sink name: ";
   cin  >> pBuf;

   sink *pSinkObj = new sink;
   pSimObj[maxSimObj] = pSinkObj;
   pSinkObj->setName(pBuf);

   maxSimObj++;
};
//-------------------------------------------------------------
void simulate()
{
   int tPeriod;
   char temp;

   if (maxSimObj)
      setObjPointers(); // set sim object pointers

   cout << "enter time period [s]: ";   // get number of simulation secs
   cin  >> tPeriod;

   cout    << setfill(' '); prtFile << setfill(' ');

   cout    <<endl << "Simulation Results:" << endl;
   prtFile <<endl << "Simulation Results:" << endl;

   cout    << setw(5)  << "Time"   // print title
           << setw(10) << "Name"
           << setw(10) << "Fin"
           << setw(10) << "Fout"
           << setw(10) << "Level"
           << endl;
   prtFile << setw(5)  << "Time"
           << setw(10) << "Name"
           << setw(10) << "Fin"
           << setw(10) << "Fout"
           << setw(10) << "Level"
           << endl;

   for (int t =1; t <=tPeriod; t++)
   {
     for(int i=0; i < maxSimObj; i++) //cycle all sim objects
         pSimObj[i]->cycle();
      cout <<endl; prtFile <<endl;
      for (int n=0; n<maxSimObj; n++)       // display dyn vars for
      {                                     // all objects
         pSimObj[n]->displayDyn(cout, t);
```

```
            pSimObj[n]->displayDyn(prtFile, t);
         };

        cout <<endl << "Press <enter> to continue: ";
        cin.get();
      };
};
/******* Create Function ***************************
*  creates an object dynamically according to what the read
*  name is
..........................................................*/
persist *create(char *s)
{
   if     (strcmp(s, "silo")     == 0)
      return new silo;
   else if(strcmp(s, "feeder")   == 0)
      return new feeder;
   else if(strcmp(s, "conveyor") == 0)
      return new conveyor(0,0);
   else if(strcmp(s, "source")   == 0)
      return new source;
   else if(strcmp(s, "sink")     == 0)
      return new sink;

   else return 0;
};
//-------------------------------------------------------
void saveObjects()
{
   simDbase.deleteAll(); // clear database

   for(int i=0; i<maxSimObj; i++)   // insert all new objects
      simDbase.insertObj(pSimObj[i]);

  simDbase.saveDbase();
};
//-------------------------------------------------------
void readObjects()
{
      persist     *psObj;
      int         n;

      // open and read all records from disk
      simDbase.openDbase();

      //-- get all the object pointers & store them on an array
      //   pSimObj[]
      //   use dynamic casting to cast dbase persist pointers to
      //   simObject pointer types - This allows caaling all functions
      //   defined in simObject, even the ones that all not declared as
      //   virtual in persist.
      n=1;
      while(psObj = simDbase.getObject(n))
      {
         pSimObj[n-1] = dynamic_cast<simObject*> (psObj);
         n++;
      };
      maxSimObj = n-1; // contains max number of objects in simulation

      cout <<endl <<"Number of Objects Read: "<<maxSimObj <<endl;

};
//-------------------------------------------------------
void printObjects()
{
   cout    << endl << "The Objects currently in Simulation are: " <<endl;
   prtFile << endl << "The Objects currently in Simulation are: " <<endl;

   for(int i=0; i<maxSimObj; i++)
   {
      cout <<endl; prtFile <<endl;
      pSimObj[i]->output(cout);
```

```
      pSimObj[i]->output(prtFile);
      cout  << endl << "Press <enter> to continue: ";
      cin.get();

   };

};
//=========================================================================

void main()
{
   // set floating point number precision
   // cout for console output
   // prtFile for log file

   cout    << setprecision(2) << setiosflags(ios::fixed);
   prtFile << setprecision(2) << setiosflags(ios::fixed);

   // call main menu
   mainMenu();
}
//==============================================================
```